λ*DNN:* Achieving Predictable Distributed DNN Training with Serverless Architectures

Fei Xu, *Member, IEEE*, Yiling Qin, Li Chen, *Member, IEEE*, Zhi Zhou, *Member, IEEE*, Fangming Liu, *Senior Member, IEEE*

Abstract—Serverless computing is becoming a promising paradigm for Distributed Deep Neural Network (DDNN) training in the cloud, as it allows users to decompose complex model training into a number of *functions* without managing virtual machines or servers. Though provided with a simpler resource interface (*i.e.*, function number and memory size), inadequate function resource provisioning (either under-provisioning or over-provisioning) easily leads to *unpredictable* DDNN training performance in serverless platforms. Our empirical studies on AWS Lambda indicate that, such *unpredictable performance* of serverless DDNN training is mainly caused by the resource bottleneck of Parameter Servers (PS) and small local batch size. In this paper, we design and implement λ*DNN*, a cost-efficient function resource provisioning framework to provide predictable performance for serverless DDNN training workloads, while saving the budget of provisioned functions. Leveraging the PS network bandwidth and function CPU utilization, we build a *lightweight* analytical DDNN training performance model to enable our design of λ*DNN* resource provisioning strategy, so as to guarantee DDNN training performance with serverless functions. Extensive prototype experiments on AWS Lambda and complementary trace-driven simulations demonstrate that, λ*DNN* can deliver predictable DDNN training performance and save the monetary cost of function resources by up to 66.7%, compared with the state-of-the-art resource provisioning strategies, yet with an acceptable runtime overhead.

Index Terms—Distributed DNN training, serverless computing, predictable performance, function resource provisioning.

1 Introduction

Distributed Deep Neural Network (DDNN) training is becoming an increasingly important workload in the cloud [1]. As the DNN models get complex and datasets grow large, DDNN training workloads require a considerable amount of cloud resources and thus become computationally expensive [2]. To reduce the budget and ease the management of cloud resources, the serverless architecture has recently emerged as a promising paradigm for training DNN models in a distributed manner, as it allows cloud users to decompose complex model training into a number of *functions*, without managing Virtual Machines (VMs) or servers [3]. Most cloud providers have launched serverless platforms (*e.g.*, AWS Lambda [4], Google Cloud Functions [5], and Microsoft Azure Functions [6]) for commercial use such as machine learning, data processing, and Web applications. They offer cloud users a simple resource interface to specify the number and memory size of functions and then charge only for the consumed function resources [7]. As a result, it becomes increasingly compelling to deploy DDNN training workloads in serverless platforms.

Though users are offered a simple and flexible resource model in serverless platforms, serverless DDNN training workloads easily suffer from *unpredictable performance* [8], due to inadequate function resource provisioning (*under-provisioning* or *over-provisioning*). While under-provisioning (*e.g.*, insufficient function memory size) will automatically trigger several rounds of workload re-executions [9], over-provisioning can severely degrade the resource utilization of functions. The intrinsic function resource limitations (*e.g.*, function timeout) as listed in Table 1 can further deteriorate the training performance of large DNN models. As evidenced by our motivation experiment in Sec. 2.2, the CPU utilization of functions for training the MobileNet model can be drastically degraded from 92.1% to 25.7%. Such severe function CPU under-utilization mainly originates from: (1) *resource bottleneck of Parameter Servers (PS)* [2] caused by aggregating model gradients and pushing updated model parameters to workers, and (2) *small local batch size* [10] for DDNN training workloads. Surprisingly, we have also identified that the functions have been well isolated (and thus little performance interference exists among functions) in public serverless platforms [11] (as evidenced by Sec. 2.2). Accordingly, the unpredictable performance elaborated above undoubtedly hinders cloud users from migrating DDNN training workloads to serverless platforms.

To solve such performance issues, many efforts have been devoted to optimizing the function performance *from the provider's perspective*. While most of them focus on

- *Fei Xu, Yiling Qin are with the Shanghai Key Laboratory of Multi-dimensional Information Processing, School of Computer Science and Technology, East China Normal University, 3663 N. Zhongshan Road, Shanghai 200062, China. Email:* `fxu@cs.ecnu.edu.cn`.
- *Li Chen is with the School of Computing and Informatics, University of Louisiana at Lafayette, 301 East Lewis Street, Lafayette, LA 70504, USA. E-mail:* `li.chen@louisiana.edu`.
- *Zhi Zhou is with the Guangdong Key Laboratory of Big Data Analysis and Processing, School of Computer Science and Engineering, Sun Yat-sen University, 132 E. Waihuan Road, Guangzhou 510006, China. E-mail:* `zhouzhi9@mail.sysu.edu.cn`.
- *Fangming Liu is with the National Engineering Research Center for Big Data Technology and System, the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan 430074, China. E-mail:* `fmliu@hust.edu.cn`.

Manuscript received July XX, 2020; revised November XX, 2020.

enforcing isolation among functions (*e.g.*, FAASM [12]) or mitigating the cold-start latency (*e.g.*, Catalyzer [13]), comparatively little attention has been paid to *guaranteeing* the performance [14] of serverless applications. There have also been several recent works on improving the cost efficiency of function resources for various serverless applications *from the user's perspective*, ranging from machine learning inference [15] and training workloads [16] to linear algebra algorithms [17] and Directed Acyclic Graph (DAG) jobs [18]. However, they mainly leverage existing machine learning algorithms like Deep Reinforcement Learning (DRL) [16] and Bayesian Optimization [19] to find a feasible amount of function resources. Such *black-box* approaches still require efforts (*i.e.*, around hundreds of iterations and high-quality profiled performance data samples) to train the model and thus bring non-negligible performance overhead. Moreover, the existing methods are not readily available for serverless DDNN training especially with large models (*e.g.*, ResNet50) due to the resource limitations of functions (*e.g.*, function timeout) listed in Table 1. As a result, there has been a paucity of research attention paid to delivering predictable performance to *long-running* DDNN training workloads in a *lightweight* manner in serverless platforms.

To fill this gap, in this paper, we present *λDNN*, a cost-efficient function resource provisioning framework to minimize the monetary cost and guarantee the performance for DDNN training workloads in serverless platforms. To the best of our knowledge, *λDNN* is the first attempt to demonstrate how to *achieve predictable performance for DDNN training workloads with serverless functions, while saving the training budget for cloud users. λDNN* can also reduce the resource cost for serverless computing providers. Specifically, we make the following contributions in *λDNN* as below.

▷ *First*, we build a serverless DDNN training framework with the PS architecture [20], and we design a *lightweight* analytical model to predict the DDNN training performance in serverless platforms. To capture the performance degradation caused by the resource bottleneck of PS and small local batch size, our model explicitly takes the PS network bandwidth and function CPU utilization into account.

▷ *Second*, we devise a *cost-efficient* function resource provisioning strategy to guarantee the performance of serverless DDNN training workloads. Given a DNN model with the training dataset and objective training time, *λDNN* first calculates the upper and lower bounds of provisioned functions, and then identifies an appropriate function resource provisioning plan (*i.e.*, the number and memory size of functions) with the minimal cost for DDNN training workloads.

▷ *Finally*, we implement a prototype of *λDNN* on AWS Lambda [4] and conduct extensive prototype experiments and large-scale trace-driven simulations with four representative DNN models. Experiment results show that *λDNN* can provide predictable performance for serverless DDNN training workloads, while reducing the monetary cost by up to 66.7%, in comparison to the state-of-the-art function resource provisioning strategies (*e.g.*, Siren [16]).

The rest of the paper is organized as follows. Sec. 2 empirically analyzes the key factors that cause unpredictable DDNN training performance in serverless platforms, which motivate the design of our analytical performance model for serverless DDNN training workloads in Sec. 3. Sec. 4 further presents the design and implementation of our *λDNN* function resource provisioning strategy. Sec. 5 evaluates the effectiveness and runtime overhead of *λDNN*. Sec. 6 discusses related work and Sec. 7 concludes this paper.

TABLE 1: Resource limitations of functions in the three representative serverless computing platforms (*Data is retrieved on Nov. 26th, 2020*).

Resources	Limitations[1]		
	AWS	Google	Azure
#maximum memory (MB)	$3,008$	$2,048$	$1,536$
Timeout (seconds)	900	540	600
Local storage (MB)	512	> 512	1024

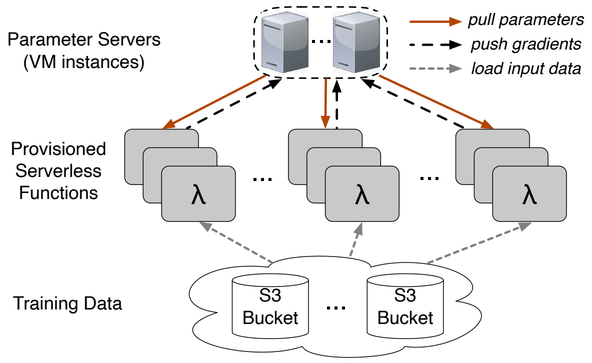


Fig. 1: A serverless DDNN training framework with the PS architecture in AWS Lambda [4].

2 Background and Motivation

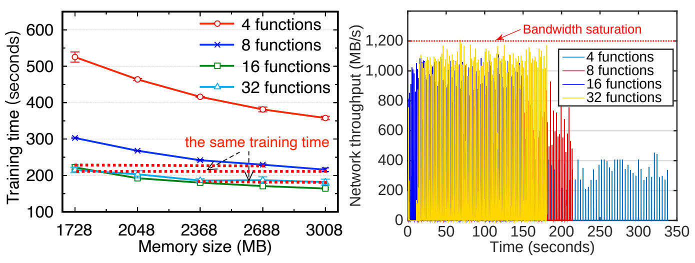
In this section, we seek to understand the following questions: *how can we effectively train DNN models in a distributed manner in serverless platforms, and what are the key factors that impact the performance of serverless DDNN training workloads?*

2.1 DDNN Training with Serverless Functions

To reduce the complexity of DDNN training and ease the resource management in the cloud [21], deploying DDNN training workloads on serverless functions becomes increasingly compelling. However, DDNN training with serverless functions is challenging, mainly because the functions have stringent limitations on a set of resources such as memory size, lifetime, local storage. As listed in Table 1, users are allowed to allocate each function with a flexible amount of memory within a maximum size (*e.g.*, $3,008$ MB), and AWS Lambda [4] allocates *proportional* computing resources (*i.e.*, CPU capability, disk I/O bandwidth). The functions are invoked and scaled automatically by requests, and each function is terminated when it completes the user request or reaches the maximum execution time (*i.e.*, timeout). In addition, the function is "stateless" and the temporary local storage will be deleted when the function is terminated.

To effectively train the DNN model, we build a serverless DDNN training framework in a public cloud platform (*i.e.*,

1. Resource limitations can be increased over time. The maximum function memory size is increased to $10,240$ MB and $4,096$ MB for AWS Lambda and Google Cloud Functions, respectively, when our paper is accepted on January 17th, 2021.



(a) (b)

Fig. 2: (a) Training time of the MobileNet model and (b) network throughput of PS over time, under different provisioning plans of function resources (*i.e.*, number and memory size of functions).

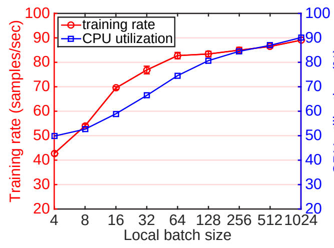


Fig. 3: Characterizing the relationship between the training rate, CPU utilization of a *single* function and *local* batch size for MobileNet.

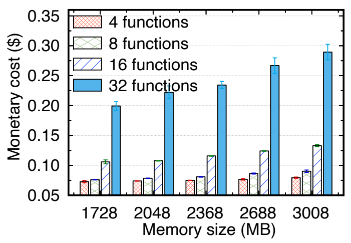


Fig. 4: Monetary cost of functions under different resource provisioning plans for training the MobileNet model.

TABLE 2: Average CPU utilization and network throughput of a function (*i.e.*, worker) during the training period of the MobileNet model under different resource provisioning plans.

Resource provisioning plans		CPU	Network
#functions (#batch)	#memory	utilization	throughput
4 functions (*local batch size:* 256)	1,728 MB	92.1%	3.85 MB/s
	2,368 MB	84.4%	3.95 MB/s
	3,008 MB	79.7%	4.08 MB/s
16 functions (*local batch size:* 64)	1,728 MB	70.0%	9.03 MB/s
	2,368 MB	60.8%	9.08 MB/s
	3,008 MB	53.0%	8.98 MB/s
32 functions (*local batch size:* 32)	1,728 MB	46.6%	7.56 MB/s
	2,368 MB	38.1%	7.59 MB/s
	3,008 MB	25.7%	7.65 MB/s

AWS Lambda [4]) as shown in Fig. 1. We adopt the PS architecture due to two facts. *First,* the PS architecture has been widely used in production machine learning clusters [20]. *Second,* serverless functions are not allowed to communicate with each other directly, which makes another widely-adopted Ring-AllReduce training architecture nontrivial to implement in serverless platforms [21]. Accordingly, to efficiently aggregate the gradients collected from functions (*i.e.,* workers), we simply use VM instances with sufficient resources as the PS. In more detail, the training input data are initially stored in the distributed storage (*e.g.,* Amazon S3 [22]). The data are evenly partitioned and dispatched to the provisioned functions when the training of DNN model starts. For each iteration, the functions first calculate and push the model gradients to the PS for aggregation. The PS then updates the model parameters once it has received all the model gradients. Finally, the functions pull the updated model parameters from the PS for the next training iteration. In particular, the workers communicate with the PS through TCP connections. The training process above is iteratively executed, until the training loss reaches an objective value or a given number of training epochs have been processed [23].

Though serverless computing can ease the resource management and scaling, *how to provision the number and memory size of functions to guarantee the performance of serverless DDNN training* still remains challenging, even for a cloud expert. As DDNN training workloads commonly consume lots of computing resources, cloud users rely on their own experience to provision an adequate number of functions always with the largest-size memory [21] to serverless deep learning workloads. Nevertheless, such a *naive* provisioning strategy [21] is likely to *underutilize* the function resources (*i.e.,* CPU, memory, I/O), thereby leading to *cost inefficiency* of function resource provisioning.

2.2 Characterizing Performance of Serverless DDNN Training Workloads

To explore the key factors that influence the performance of serverless DDNN training workloads, we conduct our motivation experiments by training representative DNN models (*e.g.,* MobileNet [24], ResNet50 [25]) on TensorFlow 1.3.1 using AWS Lambda functions [4]. Specifically, we launch a cluster of functions in the AWS region of N. Virginia (*i.e.,* `us-east-1`). We vary the number of provisioned functions from 4 to 32, and the function memory size from 1,728 MB (*i.e.,* the minimum memory size to train the MobileNet model) to 3,008 MB. To achieve fast convergence of DNN model training, we fix the *global batch size* as 1,024, and the number of training iterations as 100 (*i.e.,* 2 epochs).

Performance of serverless DDNN training: As shown in Fig. 2(a), we observe that: *First,* the DDNN training time decreases as a larger amount of memory is allocated to functions. This is simply because a larger function memory size indicates more CPU cycles to process DDNN training workloads [4]. *Second,* provisioning more functions can expectedly speed up the DDNN training process. *Interestingly*, 16 and 32 functions have a *slower* improvement in DDNN training performance compared with 4 and 8 functions. Moreover, 32 functions slightly increase the DDNN training time compared to 16 functions when allocated with over 2,048 MB memory. We conjecture that: as more functions (*i.e.,* 16, 32 functions) are provisioned, **(a)** the network I/O bandwidth of PS becomes *bottleneck*, and **(b)** a *smaller local batch size* of training data is processed on each function, as well as **(c)** the performance interference of functions *might* become severe [8]. The three factors above inevitably *underutilize* the CPU resource of provisioned functions (as shown in Table 2), and thus *offset* the performance gains of increased function resources.

Resource bottleneck of PS: To verify our analysis above, we further examine the system-level metrics including CPU utilization and network throughput of functions and PS with different resource provisioning plans. Specifically, we normalize the CPU utilization of functions by dividing the measured utilization data (*i.e.,* executing the `top` command every 0.5 seconds) to the allocated proportion of CPU capacity (*e.g.,* 3,008 MB memory corresponds to $\frac{3{,}008}{1{,}792\times 2} = 0.839$ of CPU capacity). Also, we measure the network throughput of PS and functions by tracing the network I/O statistics in the `proc` file system. As listed in Table 2, we observe that the function CPU utilization gradually decreases from 92.1% to 25.7% as more function resources are provisioned. This implies that the PS network bandwidth becomes *bottleneck* as more functions are provisioned, which is evidenced by Fig. 2(b) that the PS bandwidth becomes saturated at around $1,200$ MBps for over 16 functions. Moreover, our measurement study shows that the network throughput of functions *surprisingly remains unchanged* as the amount of allocated function memory increases (from $1,152$ to $3,008$ MB in Table 2). As a result, the data communication becomes dominating the serverless DDNN training process as the more function resources are provisioned, which in turn degrades the CPU utilization of functions.

Small local batch size: As more functions are provisioned, the *fixed global batch size* can lead to a *smaller local batch size* per function, thereby degrading the training rate on each function [10]. To verify that, we examine the function training rate and CPU utilization of the MobileNet model trained on a single function which connects to a PS node. As shown in Fig. 3, the function training rate gradually converges to a peak value (*i.e.,* 90 samples/sec) as the local batch size increases, which is highly related to the CPU utilization of a function. As the local batch size gets smaller (*i.e.,* less than 64), it *negatively* degrades the function CPU utilization below 80%. Accordingly, it would be challenging to find local and global batch sizes *good* enough to achieve both fast training rate and model convergence.

Performance interference of functions: We experimentally examine the severity of performance interference [8] among functions. Specifically, we launch 200 different AWS Lambda functions with 128 MB memory, and then use the `sysbench` benchmark to measure the CPU, memory and disk I/O performance of functions. *Surprisingly,* we identify stable function performance and thus the performance variation can be negligible among functions (*e.g.,* the average function written throughput is 12.78 MB/s with a negligible standard deviation of 0.60). We further run the `uname` command in functions to obtain the underlying host VM's IP [26]. We find that all these 200 Lambda functions have different host VM's IPs, which implies that these functions are distributed among different VMs (*i.e.,* not colocated). Our experimental results above are consistent with a latest work [11], which reports that AWS Lambda [4] has recently launched a lightweight VM monitor named Firecracker [11] to significantly mitigate the performance interference and achieve isolation among functions.

Monetary cost of serverless DDNN training: As analyzed above, we conclude that *blindly over-provisioning more function resources can be cost-inefficient*. As shown in Fig. 4, the largest memory allocation (*i.e.,* $3,008$ MB) increases the monetary cost by 45.3% compared to the lowest memory allocation (*i.e.,* $1,728$ MB) when allocated with 32 functions, while the training time is reduced only by 18.4%. This is mainly because the function resources are *underutilized* (*i.e.,* wasted) as analyzed in Table 2. *Interestingly*, 8 functions with $2,688$ memory and $3,008$ memory have almost the same DDNN training performance with 16 functions and 32 functions with $1,728$ memory (as shown in Fig. 2(a)), respectively, but the former resource provisioning plans can reduce 22.5% – 121.1% of monetary cost over the latter ones. This brings us an opportunity to judiciously provision function resources to serverless DDNN training workloads, with the aim of *guaranteeing the objective DDNN training performance while minimizing the monetary cost.*

TABLE 3: Comparison of the ResNet50 model training time and monetary cost of resources with different resource provisioning approaches. The first four strategies provision *function* resources while the last two DDNN training architectures are used for *GPU instance* resources.

Approaches	Provisioning plans		Time	Cost
	#functions (#batch)	#memory	(seconds)	(10^{-2}\$)
Random	4 (*local batch:* 64)	$1,536$ MB	Failure	N/A
Naive [21]	8 (*local batch:* 64)	$3,008$ MB	703.95	29.40
Siren [16]	9 (*local batch:* 128)	$2,816$ MB	528.96	23.19
λ*DNN*	8 (*local batch:* 128)	$2,560$ MB	**573.34**	**20.36**
PS	two ml.p3.8xlarge instances		189.67	154.77
Horovod	two ml.p3.8xlarge instances		243.33	198.56

An illustrative example: To achieve predictable serverless DDNN training performance and *cost-efficient* resource provisioning of functions, we propose λ*DNN* in Sec. 4 and illustrate its effectiveness by conducting another motivation experiment with the ResNet50 model [25]. We train the model with 1 epoch and set the objective DDNN training time as 600 seconds. As listed in Table 3, the random strategy randomly selects 4 functions allocated with $1,536$ MB memory and it fails to train the ResNet50 model. This is because the allocated function resources of $1,536$ MB are too small to process the training of ResNet50. Though the naive strategy [21] chooses the largest memory size, it exceeds the objective training time by 17.3% with the highest budget, as it chooses an inadequate small local batch size. Siren [16] and λ*DNN* both can train the workload within the objective training time, but Siren [16] causes a higher monetary cost by 13.9% compared with our λ*DNN* strategy. As a result, our λ*DNN* strategy can finish the DDNN training within the objective training time in a cost-efficient manner.

To obtain complementary insights, we have compared the DDNN training performance and monetary cost achieved on GPU instances with that on serverless functions. Specifically, we deploy two ml.p3.8xlarge instances (each is equipped with 4 GPUs and 32 vCPUs) to train the ResNet50 model with Horovod and the PS architecture on Amazon SageMaker [27]. As listed in Table 3, the DDNN training time with Horovod and the PS architecture (*i.e.,* GPU clusters) is 243.33 seconds and 189.67 seconds, respectively, which are much (*i.e.,* 1.35 – 2.02$\times$) faster than that with serverless functions. However, the serverless functions can cut down a significant amount (*i.e.,* 86.84% – 89.75%) of monetary cost as compared with the GPU instances, while guaranteeing the objective DDNN training performance. Accordingly, provisioning serverless functions is a cost-effective approach to execute DDNN training workloads.

Discussion: As the maximum memory size of functions has recently increased to $10,240$ MB for AWS Lambda, we have also re-conducted our motivation experiments by allocating function memory with $4,816$ MB, $6,624$ MB, $8,432$ MB, and $10,240$ MB, respectively. As expected, our experiment results confirm that the resource bottleneck of PS still exists and the monetary cost is significantly increased as large function memory is allocated, which are consistent with our findings in Fig. 2, Fig. 4, and Table 2 above. Though the increase in function memory limitations allows the resource-intensive workloads to run in serverless platforms, it in turn exacerbates resource underutilization of functions and expands the search space of the optimal function resource provisioning plan. Fortunately, our λ*DNN* strategy is effective and promising to provide cost-efficient function resource provisioning plans to serverless DDNN training workloads.

Summary: *First*, the serverless DDNN training performance is essentially determined by the PS network bandwidth and function CPU utilization. In particular, the resource bottleneck of PS and small local batch size are the main factors that underutilize the function resources and thus slow down the training rate. *Second*, judiciously optimizing the resource provisioning of serverless functions can achieve significant cost saving while delivering predictable performance for DDNN training workloads.

3 Modeling DDNN Training Performance in Serverless Platforms

In this section, we first build an *analytical* model to predict the serverless DDNN training performance, by explicitly considering the network bandwidth of PS and CPU utilization of provisioned functions. Next, we formulate the function resource provisioning problem to minimize the monetary cost with an objective DDNN training time, and then analyze the problem complexity. The key notations in our performance model are summarized in Table 4.

3.1 Predicting DDNN Training Time with Serverless Function Resources

Serverless DDNN training can be divided into *data loading* process and *model training* process, as illustrated in Fig. 1. Specifically, each function (*i.e.*, worker) first fetches a batch of training data samples from Amazon S3 storage. Then, the functions *iteratively* calculate the model gradients based on a batch of fetched data samples, and upload the computed model gradients to the PS where the gradients from all functions are aggregated. Finally, the functions download the updated model parameters from the PS to complete one training iteration. In general, the DNN model requires a number of iterations (denoted by k) to converge to an objective training loss value. Accordingly, the DDNN training time T can be calculated by summing up the loading time t_{load} of data samples, and the computation time t_{comp} of model gradients, as well as the communication time t_{comm} of model parameters and gradients [2], which is given by

$$T = t_{load} + k \cdot (t_{comp} + t_{comm}). \tag{1}$$

Given the number of training data samples n_t, the number of training epochs e (where one epoch denotes the training of the whole n_t data samples), and the global batch size b_g, the number of training iterations k can be calculated by

$$k = \frac{n_t \cdot e}{b_g}. \tag{2}$$

Data loading process. To achieve fast convergence, we consider the data communication of functions follows Bulk Synchronous Parallel (BSP) protocol, which has been widely used in production machine learning clusters [20]. To effectively train the DNN model, we also simply assume that the training data samples are *evenly* partitioned across the provisioned functions. Accordingly, t_{load} is calculated as

TABLE 4: Key notations in our serverless function-based DDNN training performance model.

Notation	Definition
B_f^p	Available network bandwidth between a function and PS
B_f^s	Available network bandwidth between a function and S3
B_p	Network bandwidth of the PS nodes
B_s	Disk I/O bandwidth of the S3 storage
R	Training rate of a single function
m	Allocated function memory size
n	Number of provisioned functions
b_l, b_g	Local, global batch size of training data samples
d_t, n_t	Size, number of training data samples
d_m	Size of model parameters and model gradients
k, e	Number of training iterations, epochs
t_{comp}	Computation time for each training iteration
t_{comm}	Data communication time for each training iteration
t_{load}	Data loading time of training data samples

$$t_{load} = \frac{d_t}{n \cdot B_f^s}, \tag{3}$$

where B_f^s denotes the available network bandwidth between the function and S3 storage. d_t is the size of training dataset and n denotes the number of provisioned functions.

Model training process. For each iteration, each provisioned function trains a set of data samples with the *local* batch size (denoted by b_l). Given a local batch size, the total number of data samples processed in one iteration (*i.e.*, *global* batch size $b_g = b_l \cdot n$) gets larger as more functions are provisioned. Moreover, the training rate (*i.e.*, the processing rate of data samples, denoted by R) of a function is considered as the same over the iterations with the BSP protocol. Accordingly, given n provisioned functions, we estimate the computation time t_{comp} of model gradients as

$$t_{comp} = \frac{b_g}{n \cdot R} = \frac{b_l}{R}. \tag{4}$$

In addition, the data communication time t_{comm} of each iteration includes the network transfer time to upload (push) model gradients to the PS and download (pull) the model parameters from the PS, as shown in Fig. 5. Given the size of model parameters d_m, which is practically the same as the size of model gradients [23], we can calculate the data communication time t_{comm} as

$$t_{comm} = \frac{2 \cdot d_m}{B_f^p}, \tag{5}$$

where B_f^p denotes the available network bandwidth between a function and PS nodes.

We proceed to model the training rate R of a single function and the network bandwidth (*i.e.*, B_f^p, B_f^s). As discussed in Sec. 2.1, the CPU capacity of functions is actually proportional to the allocated memory of functions [4]. Accordingly, the computation time is highly related to (*i.e.*,

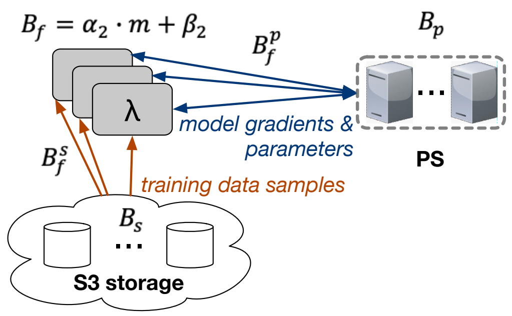

$B_f = \alpha_2 \cdot m + \beta_2$
B_f^p
B_p
λ
model gradients & parameters
PS
B_f^s
training data samples
B_s
S3 storage

Fig. 5: Data communication between functions and PS nodes, as well as the training data loading between functions and S3.

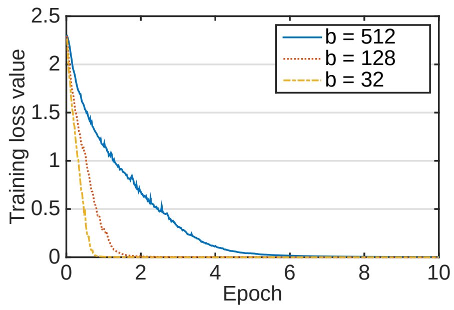


Fig. 6: Fitting training loss of `cifar10` DNN model with different global batch sizes (*e.g.*, $b_g = 512, 128, 32$).

in proportion to) the function memory size m, as evidenced by Sec. 2.2. We formulate the training rate of a function as

$$R = \alpha_1 \cdot m + \beta_1, \tag{6}$$

where α_1 and β_1 are model coefficients. In particular, the network bandwidth B_f^p between a function and PS (*i.e.*, EC2 instances) is actually *independent* of the function memory as evidenced by our experiment results in Table 2. Accordingly, it is bounded by the PS network bandwidth for each function $\frac{B_p}{n}$, as the network bandwidth of PS can become *bottleneck* (as analyzed by Sec. 2.2). Meanwhile, the network bandwidth B_f^s between a function and S3 is linear to the function memory, and it is bounded by the I/O bandwidth of S3 storage B_s. As a result, we formulate B_f^p and B_f^s as

$$B_f^p = \min\left(B_f^{max}, \frac{B_p}{n}\right), \tag{7}$$

$$B_f^s = \min(B_f, B_s) = \min\left(\alpha_2 \cdot m + \beta_2, B_s\right), \tag{8}$$

where α_2 and β_2 are model coefficients, and B_f^{max} denotes the *fixed* network bandwidth between a function and an EC2 instance. By substituting Eq. (2) – Eq. (8) into Eq. (1), we find that the DDNN training time T is actually decided by the number n and memory size m of provisioned functions as well as the local batch size b_l, which is denoted by a triple $\langle n,\ m,\ b_l\rangle$. While simple, our model is *effective enough* to predict the serverless DDNN training performance. We will evaluate the model accuracy in Sec. 5.2.

Obtaining model parameters: Based on the above, the parameters of our DDNN training performance model include five *workload-specific* parameters d_t, n_t, d_m, α_i, β_i, and three *platform-specific* parameters B_p, B_f^{max}, B_s. Specifically, the input dataset size d_t, the number of data samples n_t, and the model size d_m can be easily obtained, once the DDNN training workload (*i.e.*, the DNN model and training dataset) is submitted to the serverless platform. Furthermore, the model coefficients α_i, β_i in Eq. (6) and Eq. (8) can be obtained by workload profiling. Specifically, we profile (*i.e.*, train) the DNN model on a single one function with a small number (*i.e.*, 50) of iterations, and we record the training rates and the network bandwidth between the function and S3 under different function memory sizes. To acquire model coefficients, we fit the recorded data including the training rates and network bandwidth using the linear regression method [28]. In addition, the network bandwidth B_p and B_f^{max} can be measured running the `netperf` tool in an EC2 instance (*i.e.*, the PS node) and a function, respectively. The disk I/O bandwidth B_s of S3 storage can be obtained by transferring data from an EC2 instance to S3 by `AWS CLI`.

Identifying local and global batch sizes: To examine the relationship between DDNN training loss and global batch size, we collect the training loss values of `cifar10` DNN model[2] with six different global batch sizes (*i.e.*, 16, 32, 64, 128, 256, and 512). As shown in Fig. 6, the training loss highly depends on the number of epochs e and the global batch size b_g [23]. The model requires more epochs to reach an objective training loss value (*i.e.*, converges slowly), as b_g gets larger (when b_l is fixed and more functions are provisioned). To guarantee fast model convergence, we simply set a maximum global batch size b_g^{max} (*e.g.*, $1,024$) [10] for DDNN training. Meanwhile, we identify a local batch size b_l^f for the DNN model to achieve relatively high function CPU utilization (as evidenced by Sec. 2.2), as long as the global batch size (*i.e.*, $b_l^f \cdot n$) is below b_g^{max}. Accordingly, the local batch size b_l can be set as

$$b_l = \begin{cases} b_l^f & b_l^f \cdot n \le b_g^{max}, \\ \frac{b_g^{max}}{n} & \text{otherwise}. \end{cases} \tag{9}$$

We obtain b_l^f for each DNN model to achieve high function CPU utilization (*e.g.*, $\ge 80\%$) during the workload profiling on a single function.

3.2 Analyzing Resource Provisioning Optimization Problem of Serverless Functions

Based on our DDNN training performance model above, we further formulate the resource provisioning optimization problem of serverless functions. Such an optimization problem is defined as follows: *Given the unit price of functions p and the objective DDNN training time T_o, how can we provision the number n and memory size m of functions to guarantee the performance of DDNN training workloads, while minimizing the monetary cost of provisioned function resources.* Accordingly, our optimization problem can be formally defined as

$$\min_{m,n} \quad C = m \cdot n \cdot p \cdot T \tag{10}$$

$$\text{s.t.} \quad T \le T_o, \tag{11}$$

$$m \in \{m_{step} \cdot j \mid j \le \frac{m_{max}}{m_{step}},\ j \in \mathbb{Z}^+\}, \tag{12}$$

$$n \in \mathbb{Z}^+, \tag{13}$$

where Eq. (10) defines our objective function which minimizes the monetary cost of function resource provisioning,

2. The tutorial DNN model [29] defined in the project path "/models/tutorials/image/cifar10/" of Tensorflow.

subject to the following three constraints. Specifically, Constraint (11) guarantees the DDNN training time below the objective time T_o. Constraint (12) denotes that the allocated function memory is below the maximum size m_{max} with the memory increment of m_{step} (*e.g.*, 64 MB). Constraint (13) indicates that the number of provisioned functions requires to be a positive integer.

Problem analysis: By substituting Eq. (1) – Eq. (9) into Eq. (10), the monetary cost C is actually affected by m and n. Obviously, the training time T in Constraint (11) and the monetary cost C is non-linear with m and n. Accordingly, our optimization problem turns out to be in the form of *non-linear integer programming*, which is NP-hard to solve [30]. We turn to designing a heuristic algorithm in Sec. 4 to solve such a resource provisioning optimization problem.

To improve the algorithm efficiency, we proceed to analyze the lower bound n_{lower} and upper bound n_{upper} of provisioned functions. According to Eq. (7), the network bandwidth of functions can be *underutilized* as more functions are provisioned and thus saturates the PS network bandwidth. To avoid the resource bottleneck of PS and fully utilize the function resources (*i.e.*, CPU, network bandwidth), we have $\frac{B^p}{n} \geq B_f^{max}$. Accordingly, the upper bound of provisioned functions n_{upper} is calculated as

$$n_{upper} = \left\lfloor \frac{B^p}{B_f^{max}} \right\rfloor. \tag{14}$$

According to Constraint (11) (*i.e.*, $T \leq T_o$), we substitute Eq. (3) – Eq. (5) into Eq. (1), and calculate the lower bound of provisioned functions n_{lower} as follows:

$$n_{lower} = \begin{cases} \left\lceil \frac{\lambda}{\kappa} + \frac{\mu}{b_l^f \cdot \kappa} \right\rceil & b_l^f \cdot n \leq b_g^{max}, \\ \max\left(\left\lfloor \frac{b_g^{max}}{b_l^f} \right\rfloor, \left\lceil \frac{b_g^{max} \cdot \lambda}{b_g^{max} \cdot \kappa - \mu} \right\rceil \right) & \text{otherwise}, \end{cases} \tag{15}$$

where $\lambda = B_f^p \cdot (R \cdot d_t + n_t \cdot e \cdot B_f^s)$, $\kappa = T_o \cdot R \cdot B_f^s \cdot B_f^p$, and $\mu = 2 \cdot d_m \cdot n_t \cdot e \cdot B_f^s \cdot R$ denote the coefficients to calculate n_{lower}. In fact, two model parameters (*i.e.*, B_f^s, R) are the functions of m, according to Eq. (6) and Eq. (8). In more detail, when $n_{lower} > \frac{b_g^{max}}{b_l^f}$, it indicates that the objective training time T_o can only be achieved by fixing the global size as b_g^{max}, so that the model training convergence can be guaranteed. When $n_{upper} \leq \frac{b_g^{max}}{b_l^f}$, it implies that we can meet the objective training time by fixing the local batch size as b_l^f, while achieving high function utilization. As a result, we are able to narrow down the search space of provisioned functions within the range from n_{lower} to n_{upper} by Eq. (14) and Eq. (15), given an allocated function memory size m.

4 Design of λDNN: Guaranteeing Performance of Serverless DDNN Training Workloads

Based on our serverless DDNN performance model and resource provisioning optimization problem defined in Sec. 3, we further present *λDNN* in Alg. 1, a *simple yet effective* function resource provisioning strategy in the serverless platform (*e.g.*, AWS Lambda [4]). Our *λDNN* strategy aims to provide predictable performance for serverless DDNN training workloads, while minimizing the monetary cost of function resource provisioning.

Algorithm 1: *λDNN:* Cost-efficient function resource provisioning strategy for predictable performance of serverless DDNN training workloads.

```
Input: DDNN training workload with its input dataset size $d_t$,
    number of data samples $n_t$, model size $d_m$, and the
    objective training time $T_o$ and number of epochs $e$.
Output: Cost-efficient function resource provisioning plan
    (i.e., the number $n$ and memory size $m$ of functions).
1: Initialize: $C_{min} \leftarrow \infty$; $m \leftarrow 0$; $n \leftarrow 0$;
2: Acquire platform-specific parameters $B_p$, $B_f^{max}$, and $B_s$;
3: Obtain model-specific parameters $\alpha_i$, $\beta_i$, and $b_l^f$, as well as
   the minimum memory size $m_{min}$ to train the DNN model,
   through workload profiling on a single function;
4: Set $m_i \leftarrow m_{min}$;
5: while $m_i \leq$ the maximum memory size $m_{max}$ do
6:    Calculate the DDNN training rate $R \leftarrow$ Eq. (6), and
      function network bandwidth $B_f^s \leftarrow$ Eq. (8);
7:    Calculate the lower bound $n_{lower} \leftarrow$ Eq. (15), and the
      upper bound $n_{upper} \leftarrow$ Eq. (14);
8:    for all $n_i$ in $[n_{lower}, n_{upper}]$ do
9:       Calculate $T \leftarrow$ Eq. (1), and $C \leftarrow$ Eq. (10);
10:      if $T \leq T_o$ && $C < C_{min}$ then
11:         Record the resource provisioning plan $m \leftarrow m_i$,
            $n \leftarrow n_i$, and its monetary cost $C_{min} \leftarrow C$;
12:         break;
13:      end if
14:   end for
15:   Set $m_i \leftarrow m_i + m_{step}$;
16: end while
```

4.1 Algorithm Design

How does *λDNN* work? Given a DDNN training workload (*i.e.*, the DNN model and training dataset) with the objective training time T_o and the number of training epochs e, *λDNN* first initializes the number and allocated memory size of provisioned functions as well as the monetary cost. According to the parameter acquisition method elaborated in Sec. 3.1, *λDNN* then obtains the *platform-specific* parameters (*i.e.*, B_p, B_f^{max}, B_s) and *model-specific* parameters (*i.e.*, α_i, β_i, b_l^f) through the workload profiling on a single function (lines 1-3). To speedup the algorithm execution, *λDNN* also identifies the minimum function memory size m_{min} to train the DNN model, and it iteratively allocates the function memory size to the maximum value m_{max} with the memory increment of m_{step} (*e.g.*, 64 MB) (lines 4-5). For each allocated function memory size m_i, *λDNN* further calculates the DDNN training rate R by Eq. (6), and the network bandwidth B_f^s between a function and S3 by Eq. (8). To narrow down the search space of the number of provisioned serverless functions, *λDNN* proceeds to calculate the upper and lower bounds of n (lines 6-7). By iterating all the possible numbers of provisioned functions in the range from n_{lower} to n_{upper}, *λDNN* is able to calculate the DDNN training time T by Eq. (1) and the monetary cost C using Eq. (10) (lines 8-9). Finally, *λDNN* identifies the cost-efficient function resource provisioning plan including the number of serverless functions n and the allocated memory size m, with the guaranteed training time ($T \leq T_o$) and the minimum monetary cost (lines 10-13).

Remark: The complexity of Alg. 1 is in the order of $\mathcal{O}(p \cdot q)$, where $p = n_{upper} - n_{lower} + 1$ denotes the cardinality of search space of n, and q denotes the number of possible function memory sizes, which is equal to $\frac{m_{max} - m_{min}}{m_{step}} + 1$. The minimum function memory size m_{min} can further improve the algorithm efficiency, as the DNN model size is large in common (*e.g.*, m_{min} is $2,432$ MB for ResNet50 [25]). Accordingly, the computation overhead of our λ*DNN* strategy can be well controlled, even when the maximum memory size m_{max} is increased to $10,240$ MB. In particular, λ*DNN* identifies a *sub-optimal* function resource provisioning plan, as we narrow down the search space of function numbers and memory sizes. We will validate the lightweight runtime overhead of λ*DNN* in Sec. 5.4.

4.2 Implementation of λ*DNN*

We implement a prototype of λ*DNN* framework running on AWS Lambda [4], with over $1,500$ lines of Python, C++, and Linux Shell codes which are publicly available on GitHub[3]. To enable efficient network communication between the functions and PS, we adopt an open-source asynchronous messaging library ZeroMQ[4]. Specifically, λ*DNN* comprises two pieces of modules: a *training performance predictor* and a *function resource provisioner* as illustrated in Fig. 7. Specifically, users first submit a DDNN training workload (*i.e.*, the DNN model and training dataset) and the objective training time to the λ*DNN portal*, which can be deployed on either an EC2 instance or a Lambda function. With the model parameters obtained by λ*DNN portal*, the *performance predictor* then predicts the DDNN training time using our performance model designed in Sec. 3. To guarantee the objective DDNN training time, the *resource provisioner* further identifies the cost-efficient serverless function resource provisioning plan using Alg. 1 devised in Sec. 4. Once the cost-efficient resource provisioning plan is determined, the *function allocator* finally sets up a number of functions with an appropriate amount of memory using the command-line tools for serverless platforms (*e.g.*, `AWS CLI`). In particular, we provision the serverless functions within one Amazon Virtual Private Cloud (VPC)[5] to avoid network traffic across different Availability Zones.

Why to deploy λ*DNN*? The benefit of deploying λ*DNN* in serverless platforms is *twofold*. From the user's perspective, λ*DNN* can guarantee the DDNN training performance while saving the training budget. Accordingly, deploying λ*DNN* can attract the cloud users to migrating their DDNN training workloads to serverless platforms. From the provider's perspective, λ*DNN* can achieve the objective DDNN training time by provisioning less function resources compared with the state-of-the-art function resource provisioning strategies (*i.e.*, Siren [16]). As a result, λ*DNN* can reduce the resource cost to complete a DDNN training workload, thereby serving more DDNN training requests for a serverless computing platform. We will evaluate the benefits of λ*DNN* from both the perspectives of users and providers in Sec. 5.3.

3. https://github.com/icloud-ecnu/lambdadnn
4. https://zeromq.org
5. https://aws.amazon.com/vpc/

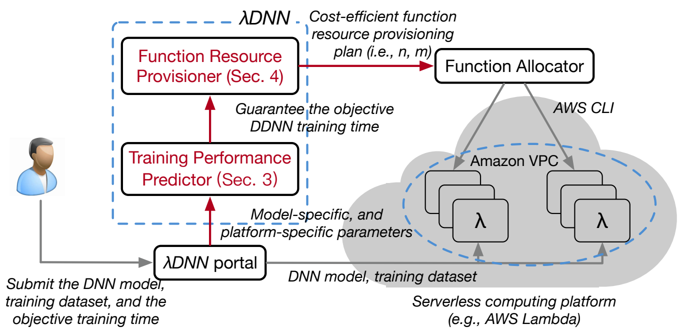

λDNN
Cost-efficient function resource provisioning plan (i.e., n, m)
Function Resource Provisioner (Sec. 4)
Function Allocator
Guarantee the objective DDNN training time
AWS CLI
Training Performance Predictor (Sec. 3)
Amazon VPC
λ
λ
Model-specific, and platform-specific parameters
λDNN portal
Submit the DNN model, training dataset, and the objective training time
DNN model, training dataset
Serverless computing platform (e.g., AWS Lambda)

Fig. 7: λ*DNN* prototype in serverless computing platforms.

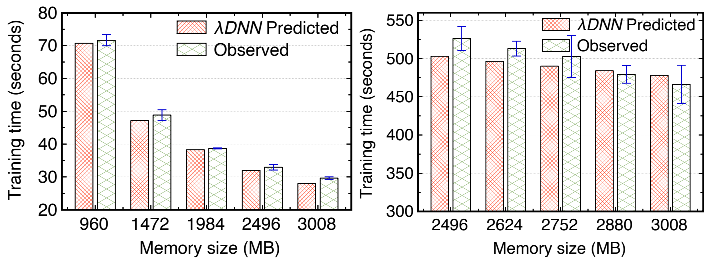
We discuss two practical issues related to the implementation of λ*DNN* as follows. *First is how to deal with the timeout of serverless functions.* As illustrated in Table 1, the serverless functions are terminated once the function expires at a certain amount of time (*e.g.*, 900 seconds for Lambda functions), which brings unpredictable performance overhead to the *long-running* DDNN training workloads. λ*DNN* simply leverages the *proactive checkpointing* technique to mitigate such performance overhead in a lightweight manner. Specifically, λ*DNN* first leverages the performance model in Sec. 3 to predict the number of iterations k_{check} that can be successfully trained before the expiration of functions (*e.g.*, 30 seconds ahead of function expirations). After k_{check} iterations, λ*DNN* then proactively checkpoints the function state (*e.g.*, the current iteration number, local batch size, and model parameters) to the S3 storage [22]. Finally, the *function allocator* of λ*DNN* relaunches a set of functions to download the function states and training dataset from S3, and restores the DDNN training process. In particular, the function relaunch process is lightweight (*i.e.*, *warm start* [31]) as the relaunched functions can be online within several seconds in our experiments.

Second is how to train large DNN models in λDNN. λ*DNN* currently supports *data parallelism*, and it requires to be extended to support *model parallelism* when *extremely large* DNN models are trained in serverless functions. For instance, the VGG19 model (with the model size of 576 MB) generates too large temporary files that require to be stored in the `/tmp` directory of functions (with the storage limitation of 512 MB as listed in Table 1). Accordingly, the large DNN model requires splitting into small partitions (*i.e.*, model parallelism) so that each partition of the DNN model can be fitted and trained in the function.

5 Performance Evaluation

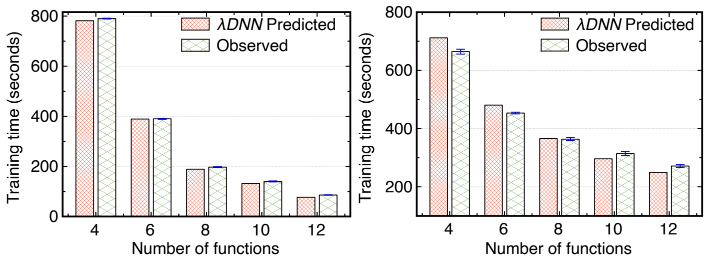
In this section, we evaluate λ*DNN* by carrying out a set of prototype experiments with four representative DNN models (as listed in Table 5) trained on AWS Lambda [4], as well as large-scale simulations driven by the Microsoft Azure trace [32]. Our prototype experiments and trace-driven simulations seek to answer the following questions:

- **Accuracy:** Can our λ*DNN* performance model accurately predict the performance of serverless DDNN training workloads? (Sec. 5.2)
- **Effectiveness:** Can our λ*DNN* resource provisioning strategy deliver predictable performance to server-



(a) (b)

Fig. 8: λDNN prediction for the training time of (a) 1DCNN model and (b) ResNet50 model with different function memory sizes.



(a) (b)

Fig. 9: λDNN prediction for the training time of (a) ESPCN model and (b) MobileNet model with different numbers of provisioned functions.

TABLE 5: Configurations and model-specific parameters of four representative DDNN training workloads.

Model	ESPCN	1DCNN	MobileNet	ResNet50
Dataset	`BSDS500`	IMDB	`cifar10`	`cifar10`
#dataset (MB)	128.7	41.1	148	148
#model (MB)	0.34	2	18	98
b_l^f	16	32	64	128
b_g^{max}	256	512	512	$1,024$
α_1, β_1	$0.001, -2.3$	$0.1, -8.8$	$0.03, 5.6$	$0.003, 11.3$

less DDNN training workloads, while saving the monetary cost? (Sec. 5.2 & Sec. 5.3)

- **Overhead:** How much runtime overhead does λDNN practically bring? (Sec. 5.4)

5.1 Experimental Setup

Configurations of serverless DDNN training cluster: We set up a serverless training cluster according to Fig. 1 in the `us-east-1` region. Specifically, we use an `m5.large` EC2 instance (equipped with 2 vCPUs, 8 GB memory) to serve as the PS, and the Lambda functions are served as the workers. We set up an S3 bucket to store the training dataset in `us-east-1` to save the budget. In particular, we measure the three *platform-specific* parameters (*i.e.*, B_p, B_f^{max}, B_s) using the `netperf` tool and `Boto3` SDK[6] according to Sec. 3.1. The network bandwidth of PS B_p and a function B_f^{max} is set as 1.2 GBps and 84 MBps, respectively, and the disk I/O bandwidth of S3 B_s is set as 115 MBps.

DDNN training workloads and datasets: We select four representative DNN models as listed in Table 5, which includes (1) the ESPCN model [33] trained on the BSDS500 dataset for super-resolution image reconstruction, (2) the 1DCNN model [34] trained on the IMDB dataset for text classification, and (3) the MobileNet model [24] trained on the `cifar10` dataset for image classification, as well as (4) the ResNet50 model [25] trained on the `cifar10` dataset for image classification. Through workload profiling on a single function, we are able to obtain the key *model-specific* parameters as elaborated in Table 5.

Comparable function provisioning strategies and metrics: We compare λDNN with the following two strategies: (1) Naive provisioning [21], which always provisions the largest memory size to functions and randomly chooses the number of functions for DDNN training workloads; (2) Modified Siren [16], which leverages the DRL method to select the adequate number and memory size of functions for achieving predictable performance while minimizing the monetary cost of DDNN training workloads, as the stock Siren strategy [16] aims to reduce the DDNN training time given a budget. We focus on two key metrics including the DDNN training time and the monetary cost for each resource provisioning plan. We illustrate the DDNN training performance with error bars of standard deviation by repeating the DDNN training workload for three times.

5.2 Effectiveness of λDNN

Can λDNN accurately predict the serverless DDNN training time? We first examine our λDNN predicted training time of 1DCNN and ResNet50 by varying the function memory size with a fixed number (*i.e.*, 8) of functions. Specifically, we train 1DCNN and ResNet50 for 3 epochs and 1 epoch, respectively. We start the minimum function memory size m_{min} as 960 MB for 1DCNN and $2,496$ MB for ResNet50, respectively. As shown in Fig 8, our performance model can well predict the DDNN training time with a prediction error of 0.98% – 6.0%, as the function memory increases to $3,008$ MB[7]. Fig 8(a) depicts that our predicted training time of 1DCNN is basically faster than the observed time. This is because small DNN models ($d_m = 2$ MB for 1DCNN) cannot fully utilize the function network bandwidth, which makes our performance model overestimate the training performance especially for small DNN models. Moreover, Fig. 8(b) shows that λDNN first overestimates and then underestimates the training performance for ResNet50, which implies that our performance model is insensitive to the allocated function memory for ResNet50. This is because the serverless training performance of large DNN models is unstable with a large standard deviation (*i.e.*, up to 34.98 seconds), which inevitably makes our workload profiling inaccurate for ResNet50. To effectively train large DNN models in functions, we will extend λDNN to support model parallelism (*i.e.*, splitting models into several parts) of DDNN training, as discussed in Sec. 4.2.

We further examine the λDNN predicted training time for ESPCN and MobileNet by varying the number of functions from 4 to 12 with a fixed amount of function memory (*i.e.*, $3,008$ MB). We train ESPCN and MobileNet for 5 epochs and 3 epochs, respectively. As depicted in Fig. 9, λDNN can basically predict the DDNN training

6. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

7. Our experiments are conducted on July 2020 ($m_{max} = 3,008$ MB), and λDNN can still work well in the scenario of $m_{max} = 10,240$ MB when our paper has been accepted on January 17th, 2021.

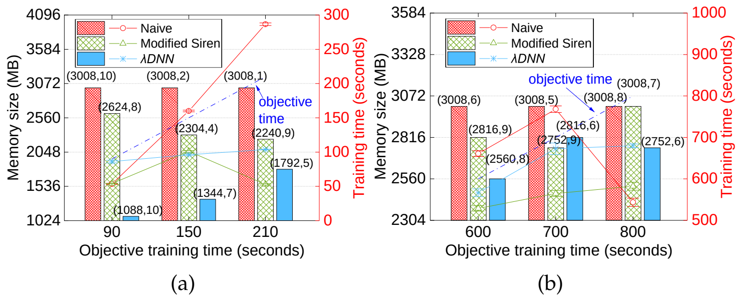


Fig. 10: Comparison of provisioned function resources and the obtained DDNN training performance achieved by the naive, modified Siren, and λ*DNN* strategies, for training (a) the 1DCNN model and (b) the ResNet50 model.

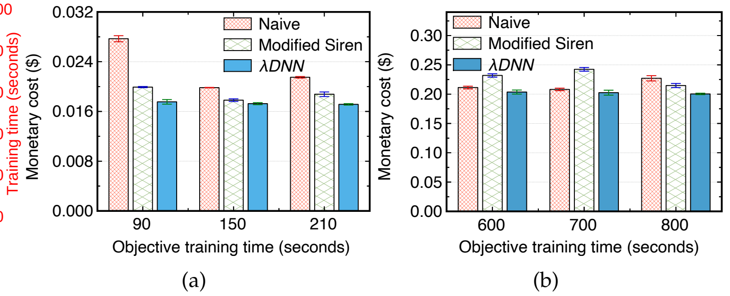


Fig. 11: Comparison of monetary cost of provisioned function resources by the naive, modified Siren, and λ*DNN* strategies for training (a) the 1DCNN model and (b) the ResNet50 model.

performance with a prediction error of 0.20% – 10.27%, as the number of provisioned functions increases. Specifically, Fig. 9(a) shows that λ*DNN* slightly overestimates the DDNN training performance for ESPCN simply because its model size is small (*i.e.*, 348 KB) as discussed in Fig. 8(a). In particular, Fig. 9(b) illustrates that the prediction error of our performance model gets large (from 0.43% to 8.6%) as the function number increases. This is because the PS has to wait for all functions to perform the gradient aggregation, and such a parameter synchronization [20] overhead cannot be overlooked as more functions are provisioned.

Can λ*DNN* guarantee the objective DDNN training time? We evaluate the predictability of DDNN training performance under the naive [21], modified Siren [16], and λ*DNN* resource provisioning strategies with two typical DNN models (*i.e.*, 1DCNN and ResNet50). Specifically, we train 5 epochs for 1DCNN and 1 epoch for ResNet50. We set three different objective training time as 90, 150, 210 seconds for 1DCNN and 600, 700, 800 seconds for ResNet50, respectively. As shown in Fig. 10, we observe that our λ*DNN* strategy can leverage less provisioned function resources to guarantee the objective training time, as compared with the other two strategies. Though the modified Siren strategy can achieve the least DDNN training time, it over-provisions the function resources and thus degrades the function resource utilization as analyzed in Sec. 2.2. In addition, the naive strategy is likely to violate the objective training time as it randomly selects provisioned function resources.

In more detail, with the objective time of 90 seconds for 1DCNN in Fig 10(a), the naive strategy always allocates the largest memory size (*i.e.*, $3,008$ MB) and randomly provisions 10 functions. The observed training time of such a provisioning plan is 53.6 seconds which is almost the same as that of another $(2,624,\ 8)$ provisioning plan (*i.e.*, $2,624$ MB memory and 8 functions) obtained by the modified Siren strategy. Though λ*DNN* achieves the largest model training time (*i.e.*, 86.2 seconds), it provisions the least amount of function resources $(1,088,\ 10)$ while completing the job before the objective training time. Our experiment results obtained on ResNet50 in Fig. 10(b) are similar to that on 1DCNN. As a result, over-provisioning function resources is likely to degrade the DDNN training performance and incur large monetary cost of provisioned function resources, which will be analyzed as follows.

Can λ*DNN* minimize the monetary cost of serverless functions? As shown in Fig. 11, we observe that λ*DNN* always achieves the minimum monetary cost of resource provisioning for DDNN training. Specifically, λ*DNN* can save the monetary cost by up to 19.7% and 57.9%, as compared with the naive [21] and modified Siren [16] provisioning strategies, respectively. Fig. 11(a) shows that the naive strategy has the largest monetary cost because it incurs resource over-provisioning $(3,008,\ 10)$ for 90 seconds or under-provisioning $(3,008,\ 1)$ for 210 seconds. As an example, though the naive strategy can save 38.7% training time under the objective training time of 90 seconds, its monetary cost is 57.9% higher than λ*DNN*. As shown in Fig. 11(b), the modified Siren still has higher monetary cost than λ*DNN* simply because it always over-provisions function resources, as we have illustrated in Fig 10(b). The rationale is that the modified Siren [16] adopts the DRL method, which highly depends on the quality of training data samples of DRL model. To achieve good performance, the modified Siren is likely to train DRL model for each DNN model using a number of data samples, while λ*DNN* builds a lightweight analytical performance model based on the workload profiling for each DNN model only once.

Can λ*DNN* handle the timeout of serverless functions? To extend the DDNN training time to exceeding 15 minutes, we set the training epochs of MobileNet model as 6, and its objective training time T_o as $1,200$ seconds. To guarantee the training performance, our λ*DNN* strategy provisions 7 functions with $1,728$ memory size, while the modified Siren [16] and naive [21] strategies provision 10 functions with $2,752$ memory size and 4 functions with $3,008$ memory size, respectively. The training time of MobileNet under λ*DNN* and modified Siren is 967.5 seconds and 738.9 seconds (both are within $1,200$ seconds), respectively, while the naive strategy cannot complete the DDNN training process because it does not handle the function timeout. In particular, the modified Siren over-provisions function resources compared with λ*DNN* so that the training process of MobileNet does not encounter the function timeout (within 900 seconds). Nevertheless, the modified Siren slightly incurs more monetary cost than λ*DNN* by 66.7%, and also the stock Siren [16] does not reveal the implementation details on dealing with the function timeout. In contrast, λ*DNN* leverages *proactive checkpointing* to deal with the function timeout, as elaborated in Sec. 4.2. As shown in Fig. 12, we observe that the function training rate is slightly impacted by our proactive checkpointing for around 6 seconds (*i.e.*, three iterations from 840 seconds to 846 seconds). As a result, λ*DNN* brings an acceptable checkpointing and restoration overhead to the DDNN training performance.

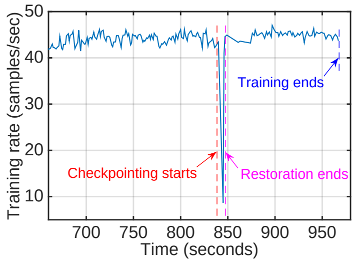


Fig. 12: Function training rate over time during MobileNet training with λ*DNN* as function timeout occurs.

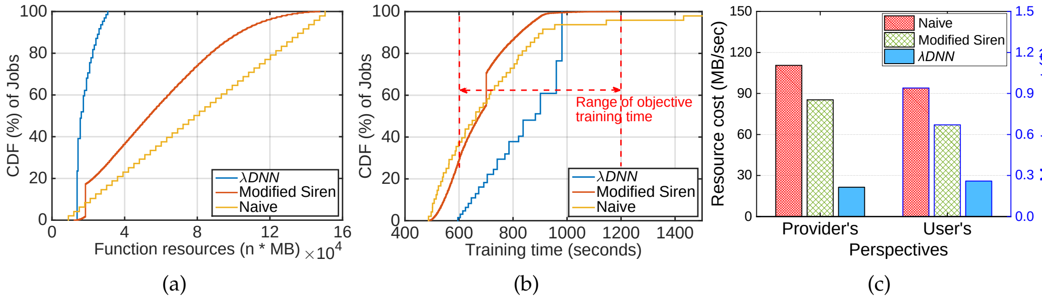


Fig. 13: CDF of (a) provisioned function resources and (b) DDNN training time of $117,325$ production jobs from Microsoft Azure trace, and (c) the benefits of λ*DNN* from the perspectives of both users and providers.

5.3 Large-scale Simulations Driven by Microsoft Azure Trace

To illustrate the benefits of λ*DNN from the perspectives of both cloud users and providers* and obtain complementary insights, we conduct large-scale simulations driven by the real-world deep learning job trace [32] from Microsoft Azure. Specifically, we set up our simulation environment using $117,325$ production deep learning jobs from the real-world trace, and $50,000$ functions with a maximum amount of $3,008$ memory. We randomly set the objective DDNN training time in the range from 600 to $1,200$ seconds, and we extract the `submitted_time` field in the production job trace [32] as the start time for each job in our simulation.

As shown in Fig. 13(a), we observe that the modified Siren [16] and naive [21] strategies significantly *over-provision* function resources (*i.e.,* the product of function number and memory size) to the jobs, while λ*DNN* provisions adequate function resources to guarantee the objective DDNN training time. Specifically, our simulation results reveal that λ*DNN* can cut down the amount of provisioned function resources by 68.5% and 77.2% on average for each job, compared with the modified Siren and naive strategies, respectively. Though the average training time for each job under λ*DNN* strategy is increased by 17.3% – 25.4% than the other two strategies, λ*DNN* can guarantee the objective DDNN training time (*i.e.,* the training time achieved by λ*DNN* is within the range from 600 to $1,200$ seconds), while the naive strategy violates the objective time goals for a number of jobs, as shown in Fig. 13(b). As revealed by our simulation results, the modified Siren and naive strategies complete the DDNN training before the objective time for 91.3% and 79.5% of jobs, respectively. In contrast, λ*DNN* can guarantee the objective training time for 99.98% of jobs. This is because the DRL method [16] relies on the quality of model training samples and its resource provisioning plans cannot guarantee the given objective time. In particular, λ*DNN* fails to guarantee the performance of 0.02% of training jobs in the trace, simply because the function resources are totally occupied by the current jobs and thus there lacks available resources for these jobs.

Based on our simulation analysis above, we further illustrate the benefits of λ*DNN* in Fig. 13(c) from the *perspectives of both cloud users and providers*. Specifically, we use the resource cost to represent the *unit cost* for serverless computing providers, which indicates the average amount of resources required to execute DDNN training workloads for one second. Under the circumstances of guaranteeing the objective training time, the less resource cost will make the provider save more function resources and serve more DDNN training requests. As an example in Fig. 10(b), to train ResNet50 within 800 seconds, the resource cost for the providers is 44.3 MB/sec, 36.2 MB/sec, and 24.3 MB/sec with the naive, modified Siren, and λ*DNN* strategies, respectively. Similarly in our trace-driven simulation, λ*DNN* can save the resource cost for the providers by 74.9% and 80.6% on average for each job, compared with the modified Siren and naive strategies, respectively. Meanwhile, we calculate the average monetary cost for serverless computing users, λ*DNN* can save the monetary cost by 60.7% and 72.1% on average for each job, compared with the modified Siren and naive strategies, respectively. Such a simulation result is consistent with our prototype experiments in Sec. 5.2.

5.4 Runtime Overhead of λ*DNN*

We evaluate the runtime overhead of λ*DNN* in terms of the profiling overhead of serverless DDNN training workloads and the computation time of λ*DNN* resource provisioning strategy (*i.e.,* Alg. 1). Specifically, we launch a Lambda function allocated with $3,008$ MB memory to profile the DDNN training workload. With a small set (*e.g.,* 2%) of data samples, we train each DNN model on the single launched function for 50 iterations. The profiling time for the training of ESPCN [33], 1DCNN [34], MobileNet [24], and ResNet50 [25] models is 114.37, 15.72, 128.36, and 310.34 seconds, respectively. The result above shows that the profiling overhead highly depends on the type of DNN models and can be within several minutes. After obtaining the performance model parameters, we also run our λ*DNN* strategy in Alg. 1 on the single launched function. The computation overhead of λ*DNN* for ESPCN, 1DCNN, MobileNet, and ResNet50 models is 0.99, 1.38, 1.15, and 0.61 milliseconds, respectively. Even when the maximum function memory size m_{max} is increased to $10,240$ MB, the computation overhead of λ*DNN* is within 5.27 milliseconds for the four workloads. This is because the computation time of Alg. 1 is linear to the lower and upper bounds of the number and memory size of provisioned functions, as analyzed in Sec. 4.1. As a result, the runtime overhead of our λ*DNN* strategy is well controlled and practically negligible.

6 Related Work

Performance characterization and optimization of serverless applications: There have been works on characterizing the function performance from the *user's perspective* (*i.e.,* outside of serverless platforms) [35]. For instance, two empirical studies are conducted on public serverless platforms such as AWS Lambda [4] and Azure Functions [6], in order to evaluate the performance of distributed data processing [36] and DNN inference workloads [31], respectively. To particularly optimize DDNN training performance, Feng *et al.* [37] design a multi-layer PS architecture with serverless functions to reduce the latency of transferring gradients. As functions cannot directly communicate with each other, however, such a hierarchical PS structure tends to bring much communication overhead to DDNN training. To tackle such function communication barriers, a recent work named Cirrus [21] deploys a cloud VM as the data store (*i.e.,* the PS) to facilitate the execution of end-to-end serverless machine learning workloads. λ*DNN* differs from prior works above in that: (1) We characterize serverless DDNN training performance in terms of PS network bandwidth, function CPU utilization, and local batch size. (2) We focus on guaranteeing DDNN training performance in serverless platforms, through the cost-efficient provisioning of function resources.

To particularly optimize the performance of serverless applications from the *provider's perspective* (*i.e.,* inside the serverless platforms), Shahrad *et al.* [38] fully identify a set of system-level performance overheads such as containerization, cold-start latency, and inter-function interference. To mitigate the performance overhead of function cold starts, Catalyzer [13] designs a generic serverless sandbox system to restore functions from the checkpoint image. Shahrad *et al.* [39] further optimize the keep-live value and pre-warm functions based on the invocation patterns of real-world function workloads. To provide resource isolation among functions, EMARS [40] designs a predictive model to limit the function memory size according to the workload usage, and FAASM [12] develops a lightweight isolation abstraction for data-intensive serverless computing. To achieve predictable performance for serverless applications, several works (*e.g.,* [41]) propose the centralized resource scheduler that can balance the load of functions [42], and dynamically allocates adequate amounts of CPU and memory resources to functions [43]. Orthogonal to prior works above, λ*DNN* focuses on optimizing the function resource provisioning plans to deliver predictable performance to serverless DDNN training workloads from the user's perspective, while reducing the resource cost from the provider's perspective. Besides, we experimentally identify *negligible* function performance interference in AWS Lambda [4].

Resource provisioning of serverless functions: There have been several recent studies devoted to provisioning adequate function resources to serverless applications. For example, an open-source resource provisioning tool named AWS Lambda Power Tuning [44] leverages Step Functions[8] to help users fine-tune the optimal memory allocation of Lambda functions. However, it requires running the user's workload with all the possible memory configurations, which inevitably brings non-negligible profiling overhead. To improve the cost efficiency of resource provisioning, MArk [15] and SplitServe [45] jointly provision VM instances and serverless functions for machine learning inference workloads and complex workloads (*e.g.,* Spark jobs), respectively. To meet the workload delay constraint and minimize user budget, COSE [19] leverages Bayesian Optimization to find the optimal memory size for serverless functions, while λ*DNN* relies on a simple analytical performance model to find cost-efficient serverless resources solely for DDNN training workloads. To optimize the performance of machine learning training workloads under a given budget in serverless platforms, a more recent work named Siren [16] adopts the DRL method to dynamically adjust the number and memory size of provisioned functions. Our work differs from Siren [16] in that: (1) Siren relies on a black-box method (*i.e.,* DRL) to provision serverless functions, while λ*DNN* explicitly builds an analytical model to predict DDNN training performance. (2) Siren stores model parameters in the shared storage (*e.g.,* S3), where the gradient aggregation cannot be directly calculated and thus causes non-negligible model update overhead. (3) λ*DNN* explicitly considers the performance degradation caused by the PS resource bottleneck and inadequate small local batch size, which severely impacts the function CPU utilization in serverless platforms.

Cache management in serverless platforms: Several works focus on designing a cache storage system to facilitate the communication among functions. For example, Pocket [46] leverages the storage of different VM instance types to build a cost-effective ephemeral storage system for serverless applications. Locus [47] judiciously combines the slow but cheap storage (*i.e.,* Amazon S3 [22]) and fast but expensive storage (*i.e.,* Redis) to achieve the best performance with lower cost for serverless analytics jobs. Instead of Amazon S3 and ElastiCache[9], a recent work [26] designs an elastic cost-effective caching system named InfiniCache, which leverages the ephemeral serverless function memory resources to cache objects in the cloud. Orthogonal to the prior works above, λ*DNN* can adopt these caching systems to make serverless DDNN training more cost-effective. To particularly handle the function timeout, Kappa [48] adopts the traditional checkpointing mechanism to store the function state in Amazon S3 or Redis, while λDNN develops a *proactive* checkpointing technique to significantly reduce the checkpointing overhead, as validated in Sec. 5.2.

Performance modeling of DDNN Training: A number of works are devoted to predicting DDNN training performance. For example, Yan *et al.* [49] and *Paleo* [50] both build a fine-grained model to predict the training time of DNN models using a set of factors like DNN architecture, choices of parallelism methods and hardware, as well as communications strategies. Without a prior knowledge of the training model and hardware, *Optimus* [23] and CM-DARE [51] build a resource-performance regression model by online fitting the training speed. However, such model fitting brings non-negligible overhead and the model accuracy relies on the quality of fitting data samples. A more recent work named *Cynthia* [2] predicts the DDNN training performance in cloud VMs by considering the CPU and network bandwidth bottleneck and hardware heterogeneity. Different from prior works above, our performance model in λ*DNN* builds a *high-level* DDNN training performance model which solely works in serverless platforms, by leveraging the PS network bandwidth and function CPU utilization.

8. https://aws.amazon.com/step-functions/

9. https://aws.amazon.com/elasticache/

7 Conclusion and Future Work

To achieve predictable performance and save the training budget for serverless DDNN training workloads, this paper presents the design and implementation of λ*DNN*, a cost-efficient function resource provisioning framework in serverless platforms. Leveraging the network bandwidth of PS and function CPU utilization, λ*DNN* builds a lightweight analytical DDNN training performance model, which explicitly considers the performance degradation caused by the resource bottleneck of PS and small local batch size. Based on such a performance model, λ*DNN* is able to provision an adequate number and memory size of functions to train the DNN model within an objective training time, while minimizing the monetary cost of provisioned function resources. Extensive prototype experiments on AWS Lambda and large-scale trace-driven simulations demonstrate that, λ*DNN* can achieve predictable DDNN training performance and save the monetary cost of provisioned functions by up to 66.7%, in comparison to the state-of-the-art function resource provisioning strategies.

We plan to extend λDNN in the following directions: (1) implementing the model parallelism to support extremely large DNN model training in λDNN, and (2) supporting the GPU or TPU architecture once the serverless platforms release the GPU or TPU-based functions.

Acknowledgments

This work was supported in part by the NSFC under grant No.61972158, in part by the Science and Technology Commission of Shanghai Municipality under grant No.20511102802 and No.18DZ2270800, and in part by the Tencent Corporation. Li Chen's work was supported by a grant from BoRSF-RCS under the contract LEQSF(2019-22)-RD-A-21. Zhi Zhou's work was supported in part by the NSFC under grant No.61802449. Fangming Liu's work was supported in part by the NSFC under Grant 61722206 and 61761136014 (and 392046569 of NSFC-DFG) and 61520106005, in part by National Key Research & Development (R&D) Plan under grant 2017YFB1001703, in part by the Fundamental Research Funds for the Central Universities under Grant 2017KFKJXX009 and 3004210116, in part by the National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals.

References

[1] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A Survey on Distributed Machine Learning," *ACM Computing Surveys*, vol. 53, no. 2, pp. 1–33, 2020.

[2] H. Zheng, F. Xu, L. Chen, Z. Zhou, and F. Liu, "Cynthia: Cost-Efficient Cloud Resource Provisioning for Predictable Distributed Deep Neural Network Training," in *Proc. of ICPP*, Aug. 2019, pp. 1–11.

[3] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv preprint arXiv:1902.03383*, 2019.

[4] AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/

[5] Google Cloud Functions. [Online]. Available: https://cloud.google.com/functions/

[6] Azure Functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[7] X. C. Lin, J. E. Gonzalez, and J. M. Hellerstein, "Serverless Boom or Bust? An Analysis of Economic Incentives," in *Proc. of USENIX HotCloud*, Jul. 2020.

[8] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proc. of USENIX ATC*, Jul. 2018, pp. 133–146.

[9] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A Fault-Tolerance Shim for Serverless Computing," in *Proc. of ACM Eurosys*, Apr. 2020, pp. 1–15.

[10] A. Or, H. Zhang, and M. J. Freedman, "Resource Elasticity in Distributed Deep Learning," in *Proc. of MLSys*, Mar. 2020, pp. 1–12.

[11] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proc. of USENIX NSDI*, Feb. 2020, pp. 419–434.

[12] S. Shillaker and P. Pietzuch, "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proc. of USENIX ATC*, Jul. 2020, pp. 419–433.

[13] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting," in *Proc. of ACM ASPLOS*, Mar. 2020, pp. 467–481.

[14] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures," in *Proc. of ACM ICPE*, Apr. 2018, pp. 21–24.

[15] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *Proc. of USENIX ATC*, Jul. 2019, pp. 1049–1062.

[16] H. Wang, D. Niu, and B. Li, "Distributed Machine Learning with a Serverless Architecture," in *Proc. of IEEE Infocom*, Apr. 2019, pp. 1288–1296.

[17] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless Linear Algebra," in *Proc. of ACM SOCC*, Oct. 2020, pp. 281–295.

[18] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing," in *Proc. of ACM SOCC*, Oct. 2020, pp. 1–15.

[19] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: Configuring Serverless Functions using Statistical Learning," in *Proc. of IEEE Infocom*, Apr. 2020, pp. 1–10.

[20] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and Efficient GPU Cluster Scheduling," in *Proc. of USENIX NSDI*, Feb. 2020, pp. 289–304.

[21] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: a Serverless Framework for End-to-end ML Workflows," in *Proc. of ACM SOCC*, Nov. 2019, pp. 13–24.

[22] AWS S3. [Online]. Available: https://aws.amazon.com/s3/

[23] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proc. of ACM EuroSys*, Apr. 2018, pp. 1–14.

[24] MobileNet and MobileNetV2. [Online]. Available: https://keras.io/api/applications/mobilenet/

[25] ResNet and ResNetV2. [Online]. Available: https://keras.io/api/applications/resnet/

[26] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proc. of USENIX FAST*, Feb. 2020, pp. 267–281.

[27] Amazon SageMaker. [Online]. Available: https://aws.amazon.com/sagemaker

[28] D. A. Freedman, *Statistical Models: Theory and Practice*. Cambridge University Press, 2009.

[29] Tensorflow tutorials: CIFAR-10. [Online]. Available: https://github.com/tensorflow/docs/blob/master/site/en/tutorials/images/cnn.ipynb

[30] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[31] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving Deep Learning Models in A Serverless Platform," in *Proc. of IEEE IC2E*, Apr. 2018, pp. 257–262.

[32] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads," in *Proc. of USENIX ATC*, Jul. 2019, pp. 947–960.

[33] Image Super-Resolution using an Efficient Sub-Pixel CNN. [Online]. Available: https://keras.io/examples/vision/super_resolution_sub_pixel/
[34] 1D CNN for text classification. [Online]. Available: https://keras.io/zh/examples/imdb_cnn/
[35] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proc. of ACM SOCC*, Oct. 2020, pp. 30–44.
[36] H. Lee, K. Satyam, and G. C. Fox, "Evaluation of Production Serverless Computing Environments," in *Proc. of IEEE CLOUD*, Jul. 2018, pp. 827–830.
[37] L. Feng, P. Kudva, D. D. Silva, and J. Hu, "Exploring serverless computing for neural network training," in *Proc. of IEEE CLOUD*, Dec. 2018, pp. 334–341.
[38] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proc. of IEEE/ACM Micro*, Oct. 2019, pp. 1063–1075.
[39] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. of USENIX ATC*, Jul. 2020, pp. 205–218.
[40] A. Saha and S. Jindal, "EMARS: Efficient Management and Allocation of Resources in Serverless," in *Proc. of IEEE CLOUD*, Jul. 2018, pp. 827–830.
[41] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized Core-granular Scheduling for Serverless Functions," in *Proc. of ACM SOCC*, Nov. 2019, pp. 158–164.
[42] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A Scalable Low-Latency Serverless Platform," *arXiv preprint arXiv:1911.09849*, 2019.
[43] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari, "A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1442–1455, 2018.
[44] AWS Lambda Power Tuning. [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning
[45] A. Jain, A. F. Baarzi, N. Alfares, G. Kesidis, B. Urgaonkar, and M. Kandemir, "SplitServe: Efficiently Splitting Complex Workloads Across FaaS and IaaS," in *Proc. of ACM SOCC*, Nov. 2019, pp. 487–487.
[46] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proc. of USENIX OSDI*, Oct. 2018, pp. 427–444.
[47] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow Scalable Analytics on Serverless Infrastructure," in *Proc. of USENIX NSDI*, Feb. 2019, pp. 193–206.
[48] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: A Programming Framework for Serverless Computing," in *Proc. of ACM SOCC*, Oct. 2020, pp. 328–343.
[49] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems," in *Proc. of ACM SIGKDD*, Aug. 2015, pp. 1355–1364.
[50] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proc. of ICLR*, Apr. 2017.
[51] S. Li, R. J. Walls, and T. Guo, "Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers," in *Proc. of IEEE ICDCS*, Jul. 2020, pp. 1–11.

Fei Xu received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2014. He received Outstanding Doctoral Dissertation Award in Hubei province, China, and ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award in 2015. He is currently an associate professor with the School of Computer Science and Technology, East China Normal University, Shanghai, China. His research interests include cloud computing and datacenter, virtualization technology, and distributed systems.

Yiling Qin is currently working toward the master's degree in the School of Computer Science and Technology, East China Normal University, Shanghai, China. Her research interests focus on cloud computing and distributed machine learning systems.

Li Chen received the BEngr degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2012 and the MASc degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2014 and the PhD degree in computer science and engineering from the Department of Electrical and Computer Engineering, University of Toronto, in 2018. She is currently an assistant professor with the Department of Computer Science, School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, USA. Her research interests include big data analytics systems, cloud computing, datacenter networking, and resource allocation.

Zhi Zhou received the B.S., M.E., and Ph.D. degrees in 2012, 2014, and 2017, respectively, all from the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan, China. He is currently an associate professor in the School of Computer Science and Engineering at Sun Yat-sen University, Guangzhou, China. In 2016, he was a visiting scholar at University of Göttingen. He was nominated for the 2019 CCF Outstanding Doctoral Dissertation Award, the sole recipient of the 2018 ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award, and a recipient of the Best Paper Award of IEEE UIC 2018. His research interests include edge computing, cloud computing, and distributed systems.

Fangming Liu (S'08, M'11, SM'16) received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, datacenter and green computing, SDN/NFV/5G and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars, and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of Ministry of Education in China, as well as the Second Class Prize of National Natural Science Award in China.