

On Realizing Efficient Deep Learning Using Serverless Computing

Kevin Assogba[†], Moiz Arif[†], M. Mustafa Rafique[†], Dimitrios S. Nikolopoulos^λ

[†]Rochester Institute of Technology, ^λVirginia Tech

[†]{kta7930, ma3890, mrafique}@cs.rit.edu, ^λdsn@vt.edu

Abstract—Serverless computing is gaining rapid popularity as it enables quick application deployment and seamless application scaling without managing complex computing resources. Recently, it has been explored for running data-intensive, e.g., deep learning (DL), workloads for improving application performance and reducing execution cost. However, serverless computing imposes resource-level constraints, specifically fixed memory allocation and short task timeouts, that lead to job failures. In this paper, we address these constraints and develop an effective runtime framework, *DiSDeL*, that improves the performance of DL jobs by leveraging data splitting techniques, and ensuring that an appropriate amount of memory is allocated to containers for storing application data and a suitable timeout is selected for each job based on its complexity in serverless deployments. We implement our approach using Apache OpenWhisk and TensorFlow platforms and evaluate it using representative DL workloads to show that it eliminates DL job failures and reduces action memory consumption and total training time by up to 44% and 46%, respectively as compared to a default serverless computing framework. Our evaluation also shows that *DiSDeL* achieves a performance improvement of up to 29% as compared to bare-metal TensorFlow environment in a multi-tenant setting.

Index Terms—Data-intensive Computing, Serverless Computing, Deep Learning, Data Parallelism, OpenWhisk, TensorFlow

I. INTRODUCTION

Function as a service (FaaS), also known as *Serverless Computing*, has emerged as a popular computing paradigm for modern applications. It enables rapid application deployment and provides a high level of concurrency for faster executions [1]. The basic unit of computation in serverless computing is a *function* or an *action*, which is executed in cloud datacenters on behalf of the given workload in response to an event. An action is defined as a function developed in a supported language or an executable that is executed to perform a specific task. The serverless execution is divided into two main steps, i.e., creation and invocation. The creation step provides a reference to the action on the serverless platform, and the invocation step consists of provisioning and initializing a container for executing the defined function.

The high level of concurrency enabled by serverless computing has motivated its use in many areas of high-performance computing (HPC), such as, deep learning (DL) [2]–[4] and federated computing [5]. Although existing serverless platforms do not support fork-join operations, existing software tools for orchestrating parallel workloads enable more serverless HPC applications. Similarly, existing resource management efforts to improve data locality and achieve high scalability [6] have yielded better performance for HPC workloads [7], [8] over serverless resources. High-

performance big data processing [9], specifically for designing effective pipelines for data stream processing, also benefits from serverless computing with fine-tuned resource allocation and timeout management to eliminate action failures.

Training a DL model is an iterative process and the completion time of a training job depends heavily on the available hardware resources. Serverless platforms apply resource restrictions on the function execution to ensure that the underlying resources are efficiently utilized. The amount of memory allocated for a function execution is capped by the serverless computing platforms, along with the number of parallel actions that can be executed by limiting the container pool memory available to the system. Similarly, a timeout applies to the function execution to ensure that actions do not run indefinitely. These restrictions limit the performance of serverless computing for highly parallel compute and data-intensive workloads where most jobs run for several hours and require a large amount of memory to store the input, intermediate, and output application data. An application can choose to restart a failed DL job by specifying different memory and timeout limits, however, identifying correct limits for the given job is challenging, and retrying with inappropriate limits leads to a further loss of computation and a higher cost for the user. The number of concurrent functions invoked on a serverless platform depends on the total allocated memory, whereas, there is no such limit in bare-metal systems.

In this paper, we address the challenges of memory and timeout allocation for data-intensive DL workloads. We develop a *Distributed Serverless Deep Learning (DiSDeL)* framework that improves the performance of DL jobs and efficiently manages the resource allocation and execution of training jobs on serverless platforms. Moreover, it eliminates job failures that occur due to static resource allocations in serverless platforms. Our memory allocation strategy incorporates the characteristics of the given DL model and training dataset to estimate and assign memory resources to the action container. *DiSDeL* splits the training data to control the execution load of each action and handles the timeout configuration for each DL job using historical execution metadata, i.e., training data size, DL model, batch size, number of epochs, and memory and timeout limits of each action. To demonstrate the effectiveness of *DiSDeL*, we integrate it with the open-source Apache OpenWhisk [10], which is widely used for conducting research on serverless computing. To the best of our knowledge, this is the first effort to simultaneously address memory and timeout constraints for data-intensive DL workloads in a serverless environment.

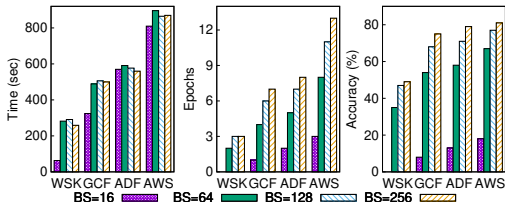


Fig. 1: Impact of varying batch size and number of epochs on training time and accuracy using MobileNet over CIFAR100.



Fig. 2: Impact of retries on the execution time using MobileNet on CIFAR100; Epochs=15.

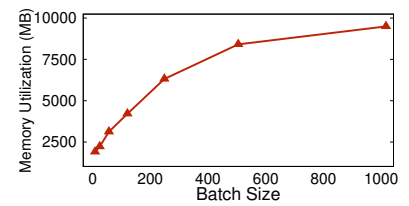


Fig. 3: Impact of the batch size on memory using MobileNet on CIFAR100; Epochs=5.

Specifically, we make the following contributions.

- We analyze the use of serverless computing for data-intensive DL training jobs and explore challenges related to fixed resource allocation in serverless environments.
- We propose *DiSDeL* that improves the performance of DL workloads on serverless platforms by leveraging batch splitting and identifying appropriate memory and timeout limits based on the requirements of the submitted jobs.
- We implement and integrate *DiSDeL* with the Apache OpenWhisk platform and conduct a thorough evaluation to study its impact on the TensorFlow platform [11]. Our evaluation shows that *DiSDeL* reduces the training time of DL jobs by up to 46% and 40% as compared to the *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively, and improves the memory utilization of an action execution by 44% on average as compared to the *Default Serverless TensorFlow* while maintaining a high training accuracy of up to 97%. Moreover, *DiSDeL* reduces the training time by 55% and 29% as compared to *Default Serverless TensorFlow* and *Bare-metal TensorFlow* respectively in a shared multi-tenant setting.

II. MOTIVATION

Cloud-based Infrastructure as a Service (IaaS) and FaaS environments [12], [13] are routinely used for running DL workloads. IaaS provides virtual machine abstractions but requires complex resource provisioning, configuration and workload management, whereas FaaS abstracts stateless functions executions [2] and simplifies resource provision and deployment. Serverless computing reduces the provisioning and management overhead and provides an easy-to-use, scalable, flexible, and cost-effective alternative to the traditional server-centric compute model. It provides short-lived execution environments, where the cost and resource consumption of DL jobs is controlled by effectively handling the resource limits on function execution. It enables simpler application development and scalable deployment while offering opportunities to reduce the carbon footprint of DL jobs by scheduling lightweight functions on low-powered servers and compute and data-intensive functions on more powerful servers. Distributed DL on serverless platforms offers better performance for training DL models as compared to IaaS platforms under the same cost constraint [2]. Furthermore, serverless computing is suitable for periodic model training, e.g., continuous learning for incremental learning systems such as recommendation and anomaly detection systems where a prediction model is periodically updated after acquiring new data.

DL models consist of several layers and training them over large datasets requires large memory. The increasing complexity of DL workloads leads to increasing demand for higher level of parallelism. While modern GPUs are used to achieve parallelism, their acquisition requires a significant capital investment. Unlike GPUs, CPU-enabled serverless platforms support the concurrent execution of thousands of functions at an affordable cost. However, traditional serverless computing assumes short-lived tasks and enforces limits on resource utilization. For example, Apache OpenWhisk, AWS Lambda, Azure Function, and Google Cloud Function have a default timeout of 300 sec., 900 sec., 600 sec., and 540 sec., respectively. We conducted a series of experiments to investigate the impact of the fixed limits of various serverless platforms on the total execution time, the number of epochs, and training accuracy, and report the results in Figure 1. We use the default timeout values of various serverless platforms in OpenWhisk to observe these impacts. With the default limits, successful training is completed for OpenWhisk in 282 sec. for the DL model with a batch size of 64, over two epochs, to achieve an accuracy of 35%. The first epoch with a batch size of 16 exceeds the default timeout and results in job failure. AWS timeout only runs the training for eight epochs to achieve an accuracy of up to 67% in 896 sec. Subsequent training epochs fail as the training time exceeds the default AWS timeout allocated for actions execution. The correlations between the number of epochs and model accuracy show that the model accuracy increases as the model training spans over more epochs at the cost of increased training time. Therefore, successful execution and training of high-quality models cannot be guaranteed unless the desired memory and timeout limits are specified for each action.

It is challenging to accurately estimate the training time, required memory, and the number of epochs to achieve the desired accuracy before submitting DL jobs. Allocating lower resource limits leads to job failures which must be restarted using a retry strategy with increased memory and timeout limits. Each retry attempt consumes additional time and resources until suitable limits are used. We implemented a simple retry strategy that increments the timeout by 240 sec. for failed actions to train MobileNet [14] over the CIFAR100 [15] dataset using a batch size of 64, 128, 256, and 512 over fifteen epochs. Figure 2 shows the results. The total execution time includes failed retry attempts time as failed actions exceed the allocated time limits. The retry strategy uses 7 and 5 retry attempts for a batch size of 64 and 128 respectively, and uses 4 retry attempts for a batch size of 256 and 512

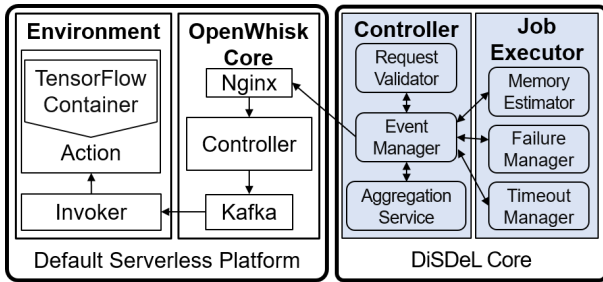


Fig. 4: High-level architecture of *DiSDeL*.

for successfully completing the job. We observe an additional latency of 5,400 sec. and 2,640 sec. for batch sizes of 64 and 128 respectively, and a latency of 1,620 sec. for batch sizes of 256 and 512. Therefore, retrying with different resource limits results in longer training times and a loss of compute cycles.

We evaluate the impact of serverless computing on the performance of a DL job by modifying the batch size. Specifically, we train the MobileNet model for five epochs on the CIFAR10 [15] dataset to evaluate the memory consumption and show in Figure 3 that there is a gradual increase in memory utilization for larger batch sizes because more memory is allocated to the job to accommodate larger input data. This behavior advocates the use of data parallelism for DL jobs to ensure that the dataset is divided into smaller batches for efficient memory management. Furthermore, training using large batch sizes leads to memory contentions, which further promotes the use of serverless computing where training can be parallelized in small batches such that each batch fits in the resources allocated for executing an action.

The default resource allocation on serverless platforms leads to failures due to insufficient resources required for successful model training. In this paper, we develop *DiSDeL* to address the challenges of memory and timeout allocations in serverless platforms. *DiSDeL* partitions the input dataset into batches after incorporating the number of concurrent functions that can be executed on the underlying serverless platform. Furthermore, *DiSDeL* seamlessly analyzes workload characteristics to predict the memory requirement per action and incorporates this in launching concurrent functions to eliminate job failures. *DiSDeL* benefits both users and cloud service providers as suitable memory and timeout allocation reduces the training cost, optimizes job scheduling, and increases resource utilization.

III. SYSTEM DESIGN

In this section, we describe the design objectives of *DiSDeL*, and explain its core components and implementation.

A. Design Objectives

DL models are trained over multiple epochs until the desired accuracy is achieved using gradient descent and hyper-parameters updates. It requires structured coordination of different units and a well-developed synchronization mechanism to ensure consistency throughout the training phase. Serverless computing supports the event-triggered invocation of training jobs and the composition of sequences of actions, but it restrains communication between actions and thereby restricts

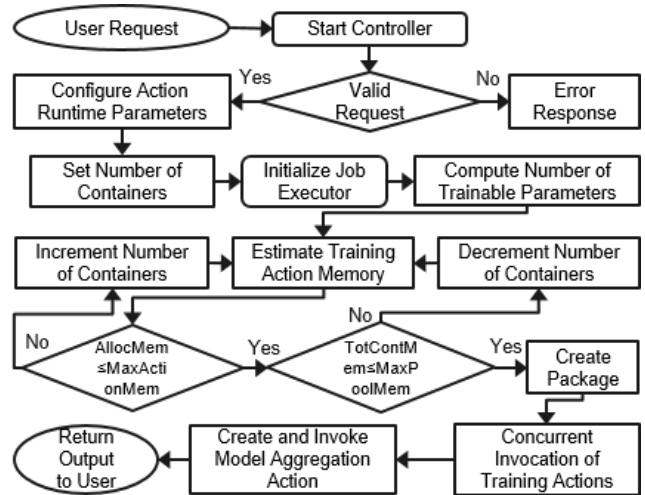


Fig. 5: Execution flow of our framework for distributed DL using serverless.

broadcasting new parameters to all processing units. This restriction results in an additional delay in transferring states of the completed actions to external storage. The resource limitations enforced by the serverless platforms make DL jobs prone to unexpected termination, missing updates, and suboptimal model accuracy. In this paper, we propose *DiSDeL* that addresses these limitations with the following objectives:

- Develop an approach to determine the appropriate memory and timeout limits for executing serverless functions.
- Eliminate function execution failures in FaaS to improve the utilization of datacenter resources.
- Improve the performance of DL workloads by reducing their function execution time on serverless platforms.

B. Optimized Serverless Platform for Deep Learning

A high-level architecture of *DiSDeL* is shown in Figure 4. The system interacts with two main external components, i.e., the core of the serverless system and the container runtime system. *DiSDeL* leverages the open-source Apache OpenWhisk as a serverless platform because of its popularity and adoption in the research community. We designed *DiSDeL* as self-contained modules and we leave exposing their functionality e.g., *configure*, *schedule*, and *process_requests*, as APIs for future work. *DiSDeL* includes two core components, i.e., a controller and a job executor, which drive the execution flow as presented in Figure 5. The user submits a request containing the DL model name, training dataset, batch size, and the number of epochs. The controller validates the request, fetches the dataset attributes, creates a package to wrap the entire composition, and starts the training process.

We leverage data parallelism to yield fine granularity of deployed actions, reduce the memory requirement per container, and address the resource constraints of serverless platforms. Data parallelism is widely used in model training for achieving a high level of parallelism, improving resource utilization, and reducing the overall training time. Moreover, it requires less memory per device which is particularly suitable for serverless computing paradigm where a function is launched to execute a fine-grained task. Figure 6 illustrates the composition of

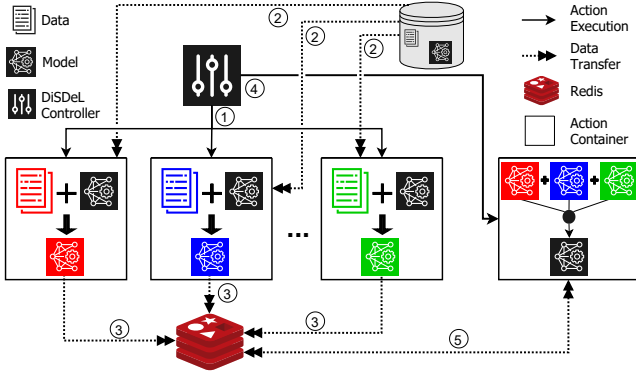


Fig. 6: Composition of containerized job execution based on data parallelism. executed actions and is explained as follows. ① The controller concurrently launches containers to host serverless functions to train a given DL model and assigns a subset of the training dataset to each training action based on the total number of scheduled containers. ② Each serverless function reads the assigned data chunk from the storage and trains the model using that data. ③ Each function stores its intermediate weights in an in-memory data store, e.g., Redis [16] after completing the training. ④ The controller launches a container to run the aggregation function. ⑤ The aggregation function reads intermediate model weights of all functions from the in-memory store and aggregates them to generate the final model. The final model is stored back into the in-memory data store.

1) *Controller*: The controller is the main component of our framework, and contains three main sub-components, i.e., request validator, event manager, and aggregation service. It coordinates all operations, e.g., validating user requests, configuring application parameters, and collecting execution results. Based on the number of required actions, the controller splits the target dataset into smaller chunks to fit in separate actions, and requests memory estimation for each action from the job executor. The input dataset is divided into equal parts for balanced workload distribution. The number of deployed containers $c_d > 1$ is initially set to its minimum possible value 2, which is later dynamically adjusted based on the estimated action memory. Whenever the estimated memory $m_e > M$, where M represents the configured action memory limit, the number of containers is increased by $\lceil (m_e - M)/M \rceil$, and is resubmitted for memory estimation. Once a valid allocation scheme is determined, the controller orchestrates fork-join operations and returns the execution results to the user.

a) *Request Validator*: The request validator handles all validation tasks that include checking the validity and completeness of the submitted request. Once the job request is submitted, the validator verifies the request parameters, such as DL model name, training dataset name, batch size, and the number of epochs. Once the request is validated it is then forwarded to the Event Manager.

b) *Event Manager*: The Event Manager coordinates all events that take place within the controller and the job executor as illustrated in Algorithm 1. It interacts with other components to request, assign and collect responses of different tasks,

Algorithm 1: Execution workflow of Event Manager.

Input: *model, data, batch size (bch), #. of epoch (epc).*
Output: Status of the execution (Succeeded OR Failed OR Error).

```

1 begin
2   parse DL job request request
3   if is_valid(request) then
4     Retrieve model, data, bch, and epc from request
5   else
6     Return Error
7   Get job execution history from file execution.log
8   if job not in execution.log then
9     Estimate memory  $m_a$  and timeout  $e_{time}$ 
10    Determine aggregation scheme agg
11    Get #. of training  $n_{tr}$  and #. of aggregation  $n_{agg}$  actions
12  else
13    Load  $m_a, e_{time},$  and agg from execution.log
14  Invoke  $n_{tr}$  training actions concurrently
15  for all cluster  $n_{cst} \in n_{agg}$  do
16    for all action  $n_{idx} \in n_{cst}$  do
17      join  $n_{idx}$ 
18    Invoke aggregation action of cluster  $n_{cst}$ 
19  Return status Succeeded OR Failed

```

e.g., estimated memory, assigned timeout, and the number of containers to launch. It also submits jobs to OpenWhisk and sends the response back to the user.

c) *Aggregation Service*: The aggregation service determines the appropriate number of aggregation actions to launch in order to meet the memory requirements of training actions. The procedure starts with level-1 aggregation where the aggregation service determines if one level-1 container is enough to handle the workload from all level-0 containers. If the memory requirement to aggregate all level-0 containers exceeds the allocation of level-1 containers then the service launches two or more level-1 containers and adds one level-2 container for all level-1 containers. This process continues until the entire aggregation process is completed. The launched training containers are divided into logical clusters based on the total number of deployed containers and their respective memory estimates. Each logical cluster is assigned to an aggregation action that is launched as soon as containers in that cluster complete the training process. Each aggregation action updates its copy of the model before the controller launches the highest level aggregation container to complete the aggregation service and stores the final trained model in the data storage. This strategy addresses the challenges caused by large DL models and provides fault tolerance to DL jobs.

2) *Job Executor*: DL jobs are memory intensive and would fail if insufficient memory is available to run the job. To avoid failures due to memory and timeout allocation, we have developed a Job Executor that estimates the memory requirement of each action by considering model and dataset attributes, determines appropriate timeout, and analyzes failure scenarios. The Job Executor contains a Memory Estimator, Timeout Manager, and Failure Manager to perform these tasks.

a) *Memory Estimator*: The estimation process depends on the configuration of the serverless platform, i.e., the maximum action memory, which limits the memory allocation to a container, and container pool memory, which limits the number of concurrent actions. Therefore, in addition to parameters e.g., the batch size and the dimensions of the input data, our

TABLE I: Computation of # of parameters and activations.

Layer Type	# of Parameters	# of Activations
Standard convolution	$c \times s^2 \times f + f$	$m \times n \times c$
Depthwise convolution	$c \times s^2 + c \times f + f$	$m \times n \times c$
Fully connected	$in \times out + out$	out

memory estimator correlates the action memory and the container pool memory limits to ensure that the estimated memory does not exceed the system memory limit. The batch size, dimensions of the input data, and model structures are used to determine the total number of activations and parameters generated during DL jobs. Each training iteration consists of a forward and a backward propagation phase. During forward propagation, each hidden layer processes the input data with the selected activation function. During backward propagation, stochastic gradient descent is applied to update all trainable parameters in the DL model.

Memory required by a DL job depends upon the given DL model, dataset, runtime environment, and execution logs stored in the container. If fit denotes the memory required to fit the DL model, pkg designates the size of the generated runtime image, dta refers to the memory space consumed by the entire input dataset on the runtime container, mod represents the size of the compiled model, and saf is a safety limit to ensure that enough space is available to hold variables generated by the main training program and the directories containing logs and check-pointed data, then the memory consumption of a training action $a \in \{1, 2, \dots, A\}$ can be estimated using Eq. 1.

$$m_a = fit + pkg + dta + mod + saf \quad (1)$$

The values of pkg and dta correspond to the data in the filesystem for storing binary image and training datasets. The value of mod is calculated as the size of the data structure to store the total number of model parameters. In this work, we use a fixed value of 256 MB for saf , and leave its optimal calculation for future work.

The training dataset consists of records of width n and height m . Each convolution layer takes an input feature map of size $c_{in} \times n_{in} \times m_{in}$ through f filters of size s and outputs a feature map of size $c_{out} \times n_{out} \times m_{out}$. The fully connected layers take inputs from a number in of neurons and provide output using a number out of neurons. This process is repeated for every mini-batch fed to the neural network during each epoch. Input datasets are fed into the network in b mini-batches, and the model is trained over e epochs. We compute the number of trainable parameters and the number of activations for each layer type using the equations in Table I [17]. The number of activation also depends upon the convolution stride and padding. We use a convolution stride and padding of one pixel to produce input with the same height and width dimensions. The total number of trainable parameters t_{par} and activations t_{act} are obtained by aggregating parameters from individual layers. We estimate the memory utilization of fit using Eq. 2 aggregating the memory utilization of parameters m_{par} , activations m_{act} , and miscellaneous variables m_{misc} . Weights, activations, and other variables are represented in 32-bit precision by default

in TensorFlow. Therefore, we multiply our estimation by four to yield the memory in bytes.

$$fit = 4 \times (m_{par} + m_{act} + m_{misc}) \quad (2)$$

Parameters in a neural network mainly consist of weights and biases and the resulting memory consumption corresponds to the space used to store these parameters along with the gradient and momentum variables. The latter consumes the same memory as weights and biases. Hence, the total memory utilization of parameters can be evaluated as shown in Eq. 3 as a product of the total number of parameters (t_{par}) and $z = 4$.

$$m_{par} = z \times t_{par} \quad (3)$$

The memory consumption of activations is computed by correlating the total number of activations (t_{act}) with the number of propagation (x) in the training process. Training keeps track of activations throughout both forward and backward propagation consuming twice as much memory space. Accordingly, t_{act} is multiplied by $x = 2$ to compute the memory utilization of activations for a single image. The total memory used by activations can be calculated using Eq. 4.

$$m_{act} = x \times b \times t_{act} \quad (4)$$

Convolution networks reserve memory space for caching the training data batches, referred to as miscellaneous memory reservation, which can be calculated using Eq. 5 by incorporating the dimensions of input images, i.e., the width m , height n , number of channels c , and the number of epochs e .

$$m_{misc} = m \times n \times c \times e \quad (5)$$

b) Timeout Manager: The Timeout Manager collects the execution time of submitted jobs to build execution history. When a DL job is submitted, the Timeout Manager analyzes the job and utilizes the historical job execution information to assign an appropriate timeout limit. This ensures that the job is executed once without failure due to insufficient timeout. Nevertheless, in the absence of historical execution data, the Timeout Manager considers the expected computation cost e_{cost} provided by the user. e_{cost} is used along with the estimated memory to determine an expected maximum execution time $e_{time} = (e_{cost}/c_d)/(\mu \times m_e)$. e_{time} is used during the creation of training and aggregation actions. This does not ensure successful execution because the timeout is constrained by the user's expected computation cost. If the user does not provide a computation cost, the Timeout Manager applies the maximum action timeout of the serverless framework. DL frameworks, e.g., TensorFlow, provide a mechanism to track the execution time of each epoch, which can be recorded by profiling one or two initial epochs and used to estimate the timeout for a particular job. However, the profiled execution time varies depending on many factors, such as, the load on each server [18], and leads to inaccurate estimations. Moreover, large training jobs require a significant amount of time for profiling. Our approach avoids this overhead by estimating timeout before an action's execution.

c) *Failure Manager*: The Failure Manager collects errors during execution to detect anomalies where the predicted memory and timeout allocations result in a job failure. It analyzes job execution logs along with the physical resource utilization to identify the cause of a failure, i.e., container out of memory (OOM) and out of time (OOT) errors. Based on the actual error, the values for memory and timeout allocations are adjusted and saved for future executions of the same model.

3) *Execution of DL Jobs*: After memory estimation, the Event Manager invokes training actions for the DL job on the given dataset. To ensure timely completion of DL job and achieving specified accuracy, we employ a bi-objective optimization strategy to minimize the loss function cost. We consider that each action costs μ for the amount of memory consumed (in GBs) for the duration (in seconds) for its execution. In our approach of deploying c_d containers, the total memory consumed by a job is calculated as the sum of the memory consumed by each action, and the execution duration as the difference between the finish time of the latest container and the start time of the earliest container. Both memory consumption and execution time are factored by the unit cost μ to yield the total cost of the submitted jobs.

Each action runs independently and the performance of ongoing jobs is evaluated at the end of each epoch to determine if the required loss is achieved. We implement a custom callback function to define an early stopping criterion for each action. At the end of each epoch, training actions check if the loss value l_a has reached a certain threshold ϵ . The designed optimization problem minimizes the total cost f_{cost} and the mean loss obtained by averaging the loss l_a of each action. Next, we apply the ϵ -constraint method [19] incorporating the average loss objective function as a constraint. The optimization problem can be formulated as:

$$\min f_{cost} = \mu \times (T_{max} \times \sum_{a=1}^A m_a) \quad (6)$$

subject to

$$e > 1; c_d > 1; m_e \leq M; \text{ and } \frac{1}{A} \times \sum_{a=1}^A l_a \geq \epsilon;$$

The Event Manager evaluates the objective function to yield the cost of completed execution and updates the historical execution logs for the submitted job. The historical information includes the DL model, training dataset, batch size, number of epochs, estimated memory, timeout, consumed memory, training time, and cost budget, and can be used to train the same model in the future.

C. Implementation

We implemented *DiSDeL* using approximately 600 lines of python code and integrate it with the open-source Apache OpenWhisk platform. We demonstrate the effectiveness of *DiSDeL* using an open-source serverless platform, however, *DiSDeL* can be integrated with proprietary serverless platforms, e.g., AWS Lambda, Azure Functions, and Google Cloud Function. *DiSDeL* exposes a command-line interface that accepts user requests, handles scheduling, resource allocation, and invocation of serverless functions. It abstracts

away server management and resource allocation from the user to improve usability and experience. The intermediate model weights are stored in Redis, an in-memory data store that we use for fast insertion and retrieval of data. We also adapted OpenWhisk deployment to set the maximum concurrency level from 1 to 10, maximum action memory limit from 512 MB to 70 GB, and container memory pool from 2 GB to 140 GB. These values are used to evaluate the performance of serverless platforms under various conditions. The memory limits for the container pool and the actions are determined by the model and dataset. At a minimum, a single action memory limit should be enough to host the entire model and dataset. To enable multiple actions to run simultaneously for faster computation we set the container pool limit to the total system memory.

IV. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of *DiSDeL* using representative DL workloads. We use Apache OpenWhisk as a serverless platform in our evaluation.

A. Evaluation Setup

1) *Evaluation Testbed*: Our testbed consists of a cluster of 8 bare-metal servers from the Chameleon testbed [20]. Each server has two Intel Xeon Gold 6126/6240R/6242 processors, contains 192 GB of main memory, and runs Ubuntu 18.04 LTS server operating system. We deploy OpenWhisk on a Kubernetes cluster along with Docker, OpenWhisk CLI (wsk), and CouchDB [21]. We also deploy Redis in a Docker container to store the model weights.

2) *DL Models*: In our evaluation, we use popular DL models including InceptionV3 [22], ResNet50/152 [23] and VGG-16 [24]. InceptionV3 is the third edition of Google’s Inception Convolutional Neural Network used in computer vision for object classification. ResNet is an artificial neural network (ANN) that introduces identity shortcut connections using skip connections or shortcuts to jump over layers. Several ResNet variations exist based on the number of layers and training weights. VGG-16 is a CNN architecture containing 16 layers with about 138 million parameters. We compile each model with categorical cross-entropy loss function [25] and Adam optimizer [26]. These models are widely used to evaluate optimizations made to the TensorFlow platform and are representative benchmarks for larger DL models.

3) *DL Training Datasets*: We use three popular datasets from the TensorFlow catalog. These datasets are:

- **MNIST [27]**: This dataset, of size 33.55 MB, contains handwritten digits used for image classification jobs.
- **CIFAR10 [15]**: This dataset, of size 308.28 MB, contains images from ten categories, which are commonly used to train machine learning and computer vision models.
- **DMLAB [28]**: This dataset, of size 3221.22 MB, is a set of 360×480 color images used to evaluate the distance between an agent and objects in a 3D environment.

B. Execution Environments

To the best of our knowledge, Cirrus [29] is the only closely related open-source effort to *DiSDeL* that uses serverless computing for DL jobs. Cirrus only supports traditional models,

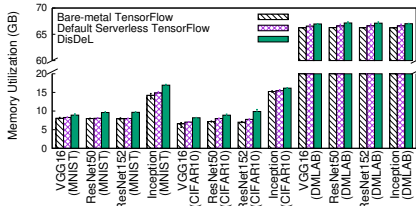


Fig. 7: Memory footprint for the three execution environments; Batch size=64, Epochs=50.

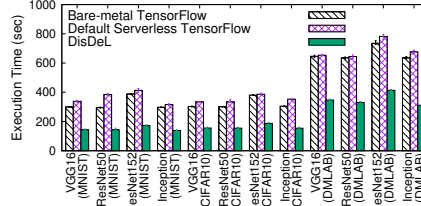


Fig. 8: Total execution time for the three execution environments; Batch size=64, Epochs=50.

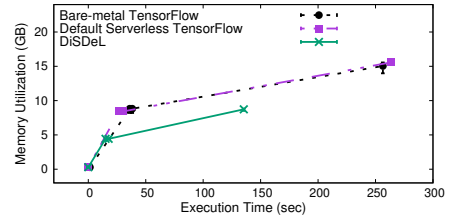


Fig. 9: Training makespan of InceptionV3 on MNIST for the three execution environments.

e.g., Sparse Logistic Regression [30], Latent Dirichlet Allocation [31], Softmax [32], and Collaborative Filtering [33], whereas, *DiSDeL* focuses on DNNs. Therefore, we use the following three execution environments to analyze the performance of *DiSDeL*:

- **Default Serverless TensorFlow:** This is the default TensorFlow running over the OpenWhisk platform. This scenario directly executes the user’s request on OpenWhisk without any optimization or middleware to control the deployment of action containers. The end-to-end workflow is managed by Apache OpenWhisk. In our evaluation, we assume that appropriate memory and timeout values, which are selected using multiple tries, are used to successfully execute DL jobs in a single attempt using one container. We consider this environment as a baseline serverless environment because this environment avoids run-time failures due to inadequate memory and timeout allocation. We deploy Apache OpenWhisk with maximum container timeout and memory allocation of 120 minutes and 70 GB, respectively.
- **DiSDeL:** This implementation of OpenWhisk includes our proposed modules. It dynamically selects suitable memory and timeout allocation for each action.
- **Bare-metal TensorFlow:** In this environment, we run DL jobs on a dedicated bare-metal cluster where no limit is imposed on the amount of memory, and DL jobs are allowed to run till completion. This is the ideal scenario where the entire server is available to run the given DL job. Distributed Training on a single server uses mirrored strategy [34] on multiple local CPUs concurrently. Distributed training on a cluster uses a multi-worker mirrored strategy [35] that utilizes multiple distributed CPUs.

We quantitatively validate the performance and effectiveness of *DiSDeL* by analyzing the memory footprint and total execution time of each DL job. We run each experiment five times and report the averages along with the observed error margins for each environment. On average, we observe a negligible, i.e., 2.8%, variance between different runs.

C. Performance Results

1) Impact on Memory Footprint and Execution Time:

We train the studied DL models using popular datasets and report the memory footprint (Figure 7) and the total training execution time (Figure 8). We show the memory consumption of a single action for serverless environments.

Figure 7 shows the memory consumption of all execution environments for the studied models and datasets. *DiSDeL*

successfully executes the submitted jobs staying within 11.5% and 8.9% of the total memory consumption of *Bare-metal TensorFlow* and *Default Serverless TensorFlow* approaches, respectively. Recall that *Default Serverless TensorFlow* is the ideal serverless deployment where the entire training runs in a single container with enough resources to complete the job. *DiSDeL* consumes more memory due to the replication of DL models in each container. Nevertheless, the batch splitting approach of *DiSDeL* allows each action to consume less than half of the total memory of *Default Serverless TensorFlow* and *Bare-metal TensorFlow*. For example, using *DiSDeL*, the ResNet50 model is successfully trained with CIFAR10 using two independent containers and consumes 44% and 39% less memory as compared to the *Default Serverless TensorFlow* and *Bare-metal TensorFlow* approaches respectively.

Figure 8 shows the total execution time of DL jobs on our execution environments. *DiSDeL* completes the DL job in significantly less time than *Bare-metal TensorFlow* and *Default Serverless TensorFlow* because of the efficiency of concurrent DL job executions on a serverless platform. The memory estimation for each action in *DiSDeL* is fine-tuned based on the model and dataset type, enabling simultaneous execution of multiple actions. For example, *Bare-metal TensorFlow* and *Default Serverless TensorFlow* trained InceptionV3 model over 60,000 data records of the MNIST dataset, but *DiSDeL* launched two concurrent containers for training on 30,000 data records each to reduce the overall training time. With a reduced data size for processing, containers in *DiSDeL* complete data pre-processing within 16 seconds while *Bare-metal TensorFlow* and *Default Serverless TensorFlow* take 30 and 20 seconds respectively to complete the same task. Overall, both training actions complete their processing in 143.7 and 144.3 seconds. Despite the additional overhead caused by the aggregation containers, *DiSDeL* yields an average training time reduction of 46% and 40% over *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively.

2) **Impact on the Training Makespan:** We evaluate the makespan of a DL training job on the three execution environments to examine their memory usage as training progresses. Figure 9 shows, as a data point, the amount of memory used for three different phases in the training process, i.e., at job submission, after data pre-processing, and after model fitting. The values for *DiSDeL* show an average of all concurrent training actions. We observe a memory consumption of approximately 280 MB across all environments at job submission due to the memory consumption of different software modules,

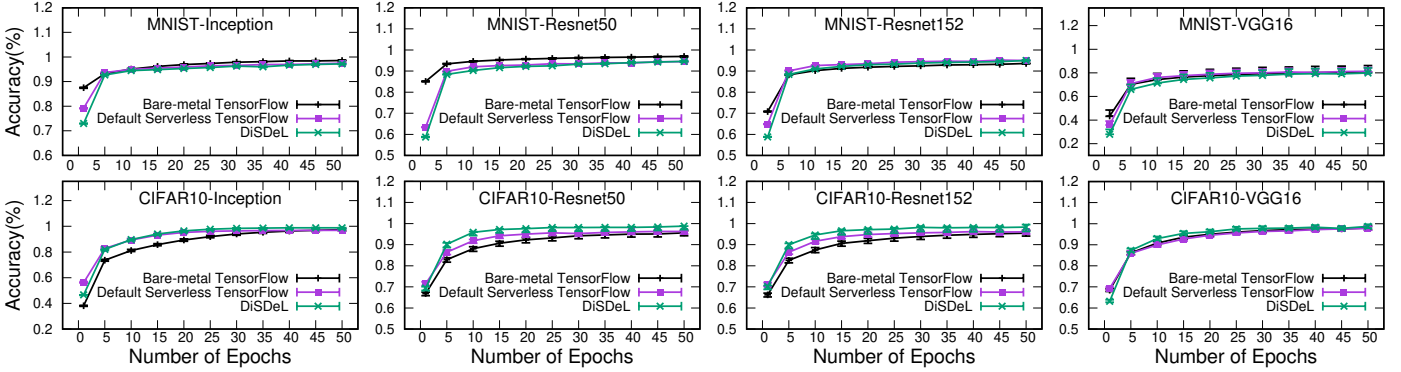


Fig. 10: Impact of the serverless computing on the training accuracy; Batch size=64, Epochs=50.

i.e., TensorFlow, TensorFlow Datasets, and Redis. While the data pre-processing phase completes within the same time and consumes the same memory for *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, *DiSDeL* requires less time and memory due to high parallelism. For all environments, we observe a significant increase in memory consumption during the training phase for storing weights, biases, and other hyper-parameters required by the training process. *DiSDeL* requires less memory per action leading to a reduced overall execution time by using concurrent actions.

3) *Impact on Training Accuracy*: We examine the impact of our execution environments on training accuracy. Figure 10 shows the training accuracy of InceptionV3, ResNet50/152, and VGG16 models on MNIST and CIFAR10 datasets over 50 epochs for each environment. We observe that serverless platforms are as effective as the *Bare-metal TensorFlow* environment. For example, while *Default Serverless TensorFlow* and *Bare-metal TensorFlow* respectively achieve 97.8% and 98.0% accuracy when training the VGG16 model with CIFAR10 dataset, *DiSDeL* yields an accuracy of 98.7% that is suitable for many production environments. We observe that for some models, e.g., Inception and ResNet using CIFAR10 dataset, *Default Serverless TensorFlow* and *DiSDeL* reach a higher accuracy than *Bare-metal TensorFlow* in early epochs, but the accuracy converges for all environments as training continues with more epochs.

4) *Impact of Memory Utilization on Execution Time*: The amount of memory available on the system directly impacts the job execution time. The action memory is restricted at two levels: first at the action itself and second at the container pool which is hosting the action’s execution. OpenWhisk assigns a default memory limit of 2 GB to container pools, which cannot be adjusted dynamically for each job. However, DL jobs require much more memory than the default container pool limit. Figure 11 shows the result of running a DL training job with DMLAB using four training actions in *DiSDeL*. The estimated memory usage of an action is 32 GB, and the container pool has a maximum memory limit of 70 GB. As soon as the DL job is submitted, the action container is launched, which brings the container pool’s memory consumption to 49% of its memory limit. At this point, another container is launched, bringing the container pool consumption to 99%, and any further actions

are queued until one of the previous actions is completed. This shows that defining a smaller container pool limits queues actions and increases the overall execution time. Increasing the container pool limit to 140 GB enables more actions to run concurrently and reduces the total execution time by 52.3%. *DiSDeL* efficiently allocates the appropriate number of containers and ensures that the underlying hardware resources are not exhausted while concurrently running multiple actions.

5) *Impact of Batch Jobs on Execution Time*: In production environments, serverless resources are shared between multiple users, and jobs are executed in batches. We evaluate the behavior of the three execution environments when a batch of eight DL jobs is submitted and show its impact on memory and total execution time. We set the action memory limit to 70 GB and the container pool memory limit to 190 GB to analyze the performance of *DiSDeL* when the available memory is the same as of *Bare-metal TensorFlow*. In Figure 12, the availability of ample memory resources allows *DiSDeL* to deploy a swarm of serverless actions to execute DL jobs. We observe a higher variation in the memory footprint using *DiSDeL* due to data parallelism that enables concurrent invocation of short-lived functions. Overall, *DiSDeL* achieves 55% and 29% faster execution of batch jobs than *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively. This shows higher scaling capabilities and confirms the effectiveness of *DiSDeL* in a multi-tenant environment as compared to *Bare-metal TensorFlow* and *Default Serverless TensorFlow*.

6) *Impact of Scaling Cluster Size on Batch Job Performance*: We conduct experiments to observe the impact of the number of cluster nodes on the three execution environments for executing a batch of eight DL jobs. We vary the cluster size from 1 to 8 nodes with 4 node increments and report the total execution time. Figure 13 shows the result. We observe that the execution time in all three environments decreases as more servers become available to run DL jobs. The execution time for *DiSDeL* is higher than *Bare-metal TensorFlow* deployment with Multi-Worker Mirrored distributed strategy as *Bare-metal TensorFlow* has access to all resources, whereas, *DiSDeL* is limited by the container pool memory. However, as the number of nodes in the cluster increase, the performance gap decreases between *Bare-metal TensorFlow* and *DiSDeL* which shows the efficiency of *DiSDeL* in leveraging the additional computing

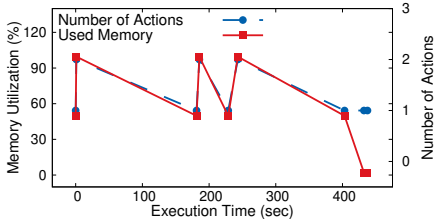


Fig. 11: Impact of system load on training time of ResNet152 on DMLAB dataset.

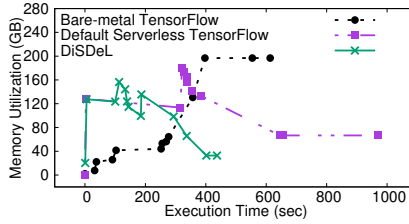


Fig. 12: Memory footprint of a batch of eight job in all three execution environments.

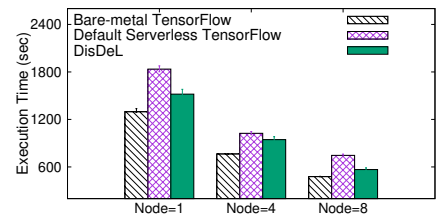


Fig. 13: Impact of the cluster size on job execution time in a shared multi-tenant environment.

resources to schedule the action containers on the appropriate nodes. Overall, *DiSDeL* shows better scalability and reduces the execution time by $0.4\times$ when the total number of nodes is doubled from 4 to 8 as compared to *Default Serverless TensorFlow* which reduces the execution time by $0.27\times$.

In summary, these experiments show a clear benefit of dynamically adapting resource allocations, specifically, memory allocation and timeout limits to improve the performance of DL workloads in serverless environments. *Default Serverless TensorFlow* runs actions in parallel to utilize the entire container memory pool allocation without considering the impact on the underlying system resources and leads to over-utilized and unresponsive servers. *DiSDeL* launches a limited number of concurrent containers, leverages batch splitting to assign less work to the deployed actions with less memory per action to meet the action memory limit, and reduces the memory consumption of the container pool.

V. RELATED WORK

Serverless computing provides an efficient, reliable, flexible, and scalable infrastructure to a variety of HPC applications. FaaS enables applications development by using granular functions, offering benefits similar to modern workflow management systems [36]. This is aligned with recent efforts to design distributed systems for DL by leveraging data parallelism, model parallelism, or hybrid strategies [7] to optimize resource utilization and enable resource sharing in HPC clusters.

Open-source FaaS: The widespread adoption of open-source software has become a driving force for cloud computing [37], and many of these systems benefit from serverless computing which provides simplified deployment and management for a variety of applications. Open-source FaaS platforms, e.g., Apache OpenWhisk, Kubeless [38], Fn Project [39], SAND [40], and *funcX* [5] offer flexible options for private deployments. However, these platforms do not support dynamic memory allocation or timeout adaption.

FaaS for DL Applications: Existing efforts to use serverless computing for DL applications mainly target lightweight computations, specifically on edge devices [9] and inference engines [41]. Lin and Glikson [42] deploy a *cat/dog* image classification model on Knative [41] for inference using TensorFlow, while Ishakian et al. [43] use AWS Lambda to serve large DL models using TensorFlow. Rausch et al. [44] explore using an Edge AI workflow on serverless platforms and propose a serverless model using edge devices as cluster resources for edge-cloud platforms. Palade et al. [9] explore

the hypothesis that incorporating serverless computing into IoT devices for small tasks reduces processing time.

There is a growing interest to deploy serverless functions for tensor-parallel operations and for end-to-end model training to achieve higher parallelism [13], [45]. Feng et al. [46] argue that serverless is ideal for training small models, and minimizing data transfer between subsequent actions improves the performance of the platform. Cirrus [29] expands the design of serverless architecture to support ML systems. It integrates a stateless server-side back-end and addresses challenges of resource constraints and workers’ scalability. It addresses memory resource limits by streaming batches of training data from storage, however, jobs are exposed to failures when training large models. The memory estimation strategy in *DiSDeL* addresses this by proactive allocations of memory before starting model training. Furthermore, Cirrus cannot run TensorFlow workloads in serverless environments due to resource constraints, whereas, *DiSDeL* fully supports DL training using TensorFlow over serverless platforms.

SIREN [2] proposes a distributed ML framework over AWS Lambda. It deploys cloud actions at each epoch to process training jobs and the scheduler selects the number of actions and handles memory allocation. This approach is similar to *DiSDeL*, however, we assigned actions per mini-batch instead of epoch to eliminate system overload, and minimize additional data access latency. Moreover, controlling the number of actions eliminates CPU over/under utilization.

Existing efforts have explored different aspects to ensure the execution of DL workloads with serverless resources. These efforts propose problem-specific or job-specific designs and do not focus on improving the resource utilization of serverless platforms. In this paper, we address the fundamental challenges of fixed resource allocations to improve the performance of DL training jobs on serverless platforms.

VI. CONCLUSION

Serverless computing cannot efficiently execute distributed deep learning (DL) workloads due to fixed memory allocation and static timeout limit of each function execution. In this paper, we propose *Distributed Serverless Deep Learning (DiSDeL)*, which is an efficient runtime framework for running long-running DL jobs on serverless platforms. *DiSDeL* ensures that the appropriate memory and timeout resources are allocated for distributed DL applications. *DiSDeL* improves performance by leveraging data parallelism for assigning jobs to concurrent actions, uses an in-memory data store to maintain the intermediate states of the training jobs, and aggregates

intermediate outputs to generate the final parameters of the trained model. We integrate *DiSDeL* with the open-source Apache OpenWhisk platform and evaluate it using representative benchmarks. We observe that *DiSDeL* fully eliminates the observed high failures in the *Default Serverless TensorFlow* environment while reducing the memory consumption of each container by 44% and training time by up to 46% while maintaining a high training accuracy of up to 97%. Moreover, *DiSDeL* reduces the training time by up to 40% as compared to *Bare-metal TensorFlow*. In a shared multi-tenant setting, *DiSDeL* reduces the training time by 55% and 29% on average as compared to *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively. In our future work, we will extend *DiSDeL* to reduce the cold start latency of containers, and extend it to include model parallelism for diverse DL applications. Moreover, we will explore improving the utilization of serverless resources by improving the scheduling of DL jobs in multi-tenant environments.

ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation under Awards No. 2106634 & 2106635 and by Cisco Award No. VTF-446663. Results presented in this paper are obtained using the Chameleon and CloudLab testbeds supported by the National Science Foundation.

REFERENCES

- [1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *Research Advances in Cloud Computing*, 2017.
- [2] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *Proc. IEEE INFOCOM*, 2019.
- [3] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proc. ACM Middleware*, 2019.
- [4] M. S. Kurz, "Distributed double machine learning with a serverless architecture," in *Proc. ACM/SPEC ICPE*, 2021.
- [5] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funx: A federated function serving fabric for science," in *Proc. ACM HPC*, 2020.
- [6] 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. ACM SoCC*, 2020.
- [7] A. Gholami, A. Azad, K. Keutzer, and A. Buluç, "Integrated model and data parallelism in training neural networks," *arXiv preprint arXiv:1712.04432*, 2017.
- [8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE CVPR*, 2009.
- [9] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *Proc. IEEE SERVICES*, 2019.
- [10] *Apache OpenWhisk*, 10 2021. [Online]. Available: <https://openwhisk.apache.org/>
- [11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. USENIX OSDI*, 2016.
- [12] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *Proc. IEEE ICMLA*, 2015.
- [13] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," *Proc. ACM SIGMOD*, 2021.
- [14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
- [15] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Toronto, Tech. Rep., 2009.
- [16] *RedisLabs - Redis*, 10 2021. [Online]. Available: <https://redis.io/>
- [17] S. Kumawat and S. Raman, "Depthwise-stft based separable convolutional neural networks," *CoRR*, vol. abs/2001.09912, 2020.
- [18] M. Arif, M. M. Rafique, S.-H. Lim, and Z. Malik, "Infrastructure-aware tensorflow for heterogeneous datacenters," in *Proc. IEEE MASCOTS*, 2020.
- [19] G. Mavrotas, "Effective implementation of the ϵ -constraint method in multi-objective mathematical programming problems," *Applied Mathematics and Computation*, vol. 213, no. 2, pp. 455–465, 2009.
- [20] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proc. USENIX ATC*, 2020.
- [21] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide: time to relax*. O'Reilly Media, Inc., 2010.
- [22] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE CVPR*, 2016.
- [23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016.
- [24] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
- [25] S. Mannor, D. Peleg, and R. Rubinfeld, "The cross entropy method for classification," in *In Proc. ICML*, 2005.
- [26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2015.
- [27] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2010.
- [28] "dmlab." [Online]. Available: <https://www.tensorflow.org/datasets/catalog/dmlab>
- [29] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proc. ACM SoCC*, 2019.
- [30] J. Liu, J. Chen, and J. Ye, "Large-scale sparse logistic regression," in *Proc. ACM SIGKDD*. Association for Computing Machinery, 2009.
- [31] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, 2003.
- [32] W. Liu, Y. Wen, Z. Yu, and M. Yang, "Large-margin softmax loss for convolutional neural networks," in *Proc. ICML*, 2016.
- [33] J. L. Herlocker, J. A. Konstan, and J. Riedl, "Explaining collaborative filtering recommendations," in *Proc. ACM CSCW*, 2000.
- [34] B. Pang, E. Nijkamp, and Y. N. Wu, "Deep learning with tensorflow: A review," *J. Educ. Behav. Stat.*, 2020.
- [35] "Distributed training with tensorflow." [Online]. Available: https://www.tensorflow.org/guide/distributed_training
- [36] E. van Eyk, A. Iosup, S. Seif, and M. Thommes, "The spec cloud group's research vision on faas and serverless architectures," in *Proc. ACM WoSC*, 2017.
- [37] E. Gorelik, "Cloud computing models," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.
- [38] "Kubeless," <https://kubeless.io/>, accessed: 2021-10-15.
- [39] "Fn Project - The Container Native Serverless Framework," <https://fnproject.io/>, accessed: 2021-10-15.
- [40] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *Proc. USENIX ATC*, 2018.
- [41] *Google Cloud - Knative*, 10 2021. [Online]. Available: <https://cloud.google.com/knative>
- [42] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," *arXiv preprint arXiv:1903.12221*, 2019.
- [43] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Proc. IEEE IC2E*, 2018.
- [44] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge AI," in *Proc. USENIX HotEdge*, 2019.
- [45] J. Thorpe, Y. Qiao, J. Eyoifson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *Proc. USENIX OSDI*, 2021.
- [46] F. Lang, K. Prabhakar, D. S. Dharma, and H. Jiang, "Exploring serverless computing for neural network training," in *Proc. IEEE CLOUD*, 2018.