Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

8-2020

A Data-parallel Approach for Efficient Resource Utilization in Distributed Serverless Deep Learning

Kevin Tunder Elom Assogba
ta7930@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

Assogba, Kevin Tunder Elom, "A Data-parallel Approach for Efficient Resource Utilization in Distributed Serverless Deep Learning" (2020). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

A Data-parallel Approach for Efficient Resource Utilization in Distributed Serverless Deep Learning

by

Kevin Tunder Elom Assogba

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. M. Mustafa Rafique

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

August 2020

A Data-parallel Approach for Efficient Resource Utilization in Distributed Serverless Deep Learning

by

Kevin Tunder Elom Assogba

has been examined and approved by the following Committee:

Dr. M. Mustafa Rafique, Chair

Dr. Taejoong Chung, Reader

Dr. Minseok Kwon, Observer

Dedication

To my beloved parents, wife and daughter for support and belief in me.

Acknowledgments

"If we knew what is was we were doing, it would not be called research, would it?"

— *Albert Einstein*

This work concludes two years of passionate exploration of a field initially new to me. Diving into the nuts and bolts of Computer Science, I became part of a community filled with people of great integrity, professionalism and leadership. I am grateful for the inspiration of many and the wisdom of others in their guidance, assistance and services. I would like to acknowledge the support of the Fulbright Program, the International Institute of Education and Rochester Institute of Technology for the opportunity to undertake my M.Sc. degree. I express my sincere gratitude ...

—to my Research Advisor **Dr. M. Mustafa Rafique** for his well-oriented and enriching guidance throughout this research.

—to **Dr. Minseok Kwon**, Observer of the Defense Committee and Instructor of the Computer Networks course, for his undisputed professionalism and rigor.

—to **Dr. Taejoong Chung**, Reader of the Defense Committee, for his promptitude and efficiency throughout this research.

—to **Dr. Hans-Peter Bischof**, Program Director of M.Sc. in C.S., for his rigor and pedagogy. Your Advanced OOP class triggered my interest in Distributed Systems.

—to **Dr. Mohan Kumar**, Chair of the C.S. Department, for the opportunity to grade for his Analysis of Algorithms class.

—to **Mrs. Rebecca O Connor**, Academic Advisor, for her assistance in diverse circumstances, especially in the final semesters.

—to **Mr. Allahsera A. Tapo, Aqueel Syeda, Aseem Mehta** for being friends and brothers, and contributing to my academic-social balance.

Abstract

A Data-parallel Approach for Efficient Resource Utilization in Distributed Serverless Deep Learning

Kevin Tunder Elom Assogba

Supervisor: Dr. M. Mustafa Rafique

Serverless computing is an integral part of the recent success of cloud computing, offering cost and performance efficiency for small and large scale distributed systems. Owing to the increasing interest of developers in integrating distributed computing techniques into deep learning frameworks for better performance, serverless infrastructures have been the choice of many to host their applications. However, this computing architecture bears resource limitations which challenge the successful completion of many deep learning jobs.

In our research, an approach is presented to address timeout and memory resource limitations which are two key issues to deep learning on serverless infrastructures. Focusing on Apache OpenWhisk as severless platform, and TensorFlow as deep learning framework, our solution follows an in-depth assessment of the former and failed attempts at tackling resource constraints through system-level modifications. The proposed approach employs data parallelism and ensures the concurrent execution of separate cloud functions. A weighted averaging of intermediate models is afterwards applied to build an ensemble model ready for evaluation. Through a fine-grained system design, our solution executed and completed deep learning workflows on OpenWhisk with a 0% failure rate. Moreover, the comparison with a traditional deployment on OpenWhisk shows that our approach uses 45% less memory and reduces the execution time by 58%.

Contents

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

Abstract . . . . **v**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Problem Definition 3
- 1.3 Research Objective and Contribution 5
- 1.4 Layout of the Thesis 6

2 Literature Review **7**
- 2.1 Parallel and Distributed Deep Learning 7
- 2.2 Serverless Computing 10
- 2.3 Serverless Machine Learning 11
- 2.4 Chapter Summary 13
  - 2.4.1 Limitations of Current Approaches 13
  - 2.4.2 Common Challenges 14

3 Enabling Technologies **15**
- 3.1 Apache OpenWhisk 15
  - 3.1.1 System Workflow 16
  - 3.1.2 OpenWhisk Invoker 17
- 3.2 TensorFlow Framework 18
  - 3.2.1 Distributed Computing Using TensorFlow Framework 18
  - 3.2.2 Distributed TensorFlow Performance Analysis 22

4 A Scalable Serverless System for Deep Learning **25**
4.1 System-Level Improvement 25
4.2 Distributed Serverless Deep Learning Module 31
4.2.1 System Interaction 31
4.2.2 Custom Action Runtime Image 32
4.2.3 Controller 33
4.2.4 Scheduler 34
4.2.5 Model Training and Averaging 35

5 Computational Experiments **37**
5.1 Experiments Design and Data 37
5.1.1 Test Cases 38
5.1.2 Data Instances 38
5.1.3 Evaluation Metrics 39
5.2 Analysis and Discussion 40
5.2.1 Numerical Analysis 40
5.2.2 Sensitivity Analysis 44
5.2.3 Discussion and Implications 45

6 Conclusions **48**
6.1 Lessons Learned 48
6.2 Future Work 49

Bibliography **50**

7 Appendix **57**
7.1 Experiments Configuration 57
7.1.1 Environment Setup 57
7.1.2 Additional resources 59

List of Tables

1.1 Comparison of some capabilities of virtual machines and serverless platforms. 3

5.1 Summary of experiments data instances. 40
5.2 Comparison of experiments results of the proposed solution and a brute-force approach on OpenWhisk. Results obtained with 4 GB as maximum memory are labeled in bold. 41
5.3 Comparison of failure rate of the proposed solution and a brute-force approach on OpenWhisk. Results obtained with 4 GB as maximum memory are labeled in bold. 42
5.4 Comparison of experiments results of the proposed solution and a deployment on a virtual machine. 43

List of Figures

1.1 Failed action invocation using default memory allocation. 4
1.2 Successful action invocation after increasing the memory limit. 4

2.1 Illustration of the data parallelism strategy. 8
2.2 Illustration of the model parallelism strategy. 9

3.1 OpenWhisk action invocation workflow. 16
3.2 Behind the scenes of OpenWhisk operations [10]. 17
3.3 Flowchart of TensorFlow Lite operations. 21
3.4 Flowchart of TensorFlow Extended. 22
3.5 Variation of computation time with respect to the epoch parameter. 23
3.6 Variation of computation time with respect to the batch size. 23

4.1 Finite state machine illustrating the container pool-proxy contract. 27
4.2 Enhancement of the Run function within OpenWhisk's docker runtime image. 30
4.3 Workflow of the controller of our proposed module. 33
4.4 Distributed deep learning workflow using serverless resources. 36

5.1 Impact of the serverless architecture on the model training accuracy and loss. 45
5.2 Impact of the batch size on the memory usage. 46
5.3 Variation of the memory usage with regards to the number of epoch. 47

Chapter 1

Introduction

1.1 Motivation

The scale and complexity of computations in this era of big data significantly challenge the engineering of highly efficient software and systems. The end-to-end design of a modern distributed system is subject to issues related to the location of the required data, communication overhead, latency, resource sharing, scalability, fault-tolerance, etc. Traditionally, using a single compute node results in running problem-solving algorithms for hours, days or weeks. But nowadays, the ability to perform trillions of tasks within seconds has motivated many researchers to employ advanced computing strategies for large scale problems. Several parallel and distributed architectures have evolved for high performance computing, especially solving problems in deep learning, DNA sequencing, galaxies simulation, etc. Existing strategies involve sharing resources of multiple machines and assigning chunks of instructions and/or data to different processors without compromising minimum latency and space requirements [15]. These distribution strategies are employed in a variety of computing tools like TensorFlow [1], one of the most used deep learning frameworks.

Horizontally scaling by adding more machines reduces the workload of the existing system and decreases the overall computation time. However, the ubiquitous practice of using physical machines for occasional computations is economically inefficient, even for long running jobs. Furthermore, deploying large scale deep learning algorithms on a cluster of computers requires an extensive resource management, engendering additional operations costs to companies for the recruitment of systems administrators. Subsequently, more

companies are moving towards cloud computing, a compute paradigm where designs for complex problems are simplified.

Diverse distributed systems are deployed in the cloud, and benefit from cloud computing features such as elasticity, pay-as-you-grow, diversity and ease of deployment. Services such as Platform as a Service (PaaS) [17], Infrastructure as a Service (IaaS) [50], Software as a Service (SaaS) [54], Function as a Service (FaaS) [65] are offered to cloud natice application developers, and the list has been growing with the provision of more specialized services like Machine Learning as a Service (MLaaS) [41] and Data Analytics as a Service (DAaaS) [55].

Our research focuses on FaaS, also known as *Serverless Computing*. These services offer a strong support to deep learning applications, hence more companies nowadays are deploying their software using serverless resources. A classic example is IoT (Internet of Things) where research are being conducted to improve the performance of edge devices [61, 12]. Aware of the limited processing capacity of smartphones, watches, camera or other small IoT devices at training or serving deep learning models, applications often offload tasks to cloud infrastructures. While the maintenance of these infrastructures causes an economic and operational burden, serverless computing services remove all operations concerns and provide a high level of concurrency for faster executions [12]. The latter represents an effective support for inference jobs which are short-lived and can leverage the concurrency level to reduce the response time [33]. Furthermore, the event-triggered architecture of serverless platforms offers an ideal environment to deploy IoT applications [37].

Despite the unequivocal advantages of serverless computing, Distributed Deep Learning (DDL) still accounts for a mere percentage of the research conducted on the subject. Essentially, operations like MapReduce [22] in data science or hyper-parameters update in deep learning are challenged by a requirement of the architecture restraining direct communication between running cloud functions. Consequently, there is still substantial room for further research on the topic of distributed deep learning on serverless infrastructure, especially on how to train large models and large data sets.

Virtual Machine	Serverless Platform
Ability to rollback during system updates	No system update needed
Networking among multiple nodes	Isolation of cloud actions
High Upfront cost	No upfront cost and Pay as you grow
Complexity in cluster configuration and management	No server management

Table 1.1: Comparison of some capabilities of virtual machines and serverless platforms.

1.2 Problem Definition

Consider a cloud native software company X which offers machine learning services to its customers. Having built an image classification platform and trained model M to recognize customers' inputs and classify them accordingly, X stored an averagely accurate model M on Y Cloud, and contracted OpenWhisk for inference jobs. After a certain period of time, X noticed that the average inference accuracy has declined, and thus decided to adopt a periodic re-training strategy to improve model M's performance.

Considering available options such as contracting virtual servers or employing OpenWhisk to retrain model M with a medium size data instance, X evaluates some of the advantages and disadvantages of each option summarized in Table 1.1, and decides to move forward with OpenWhisk. The iterative process of deep learning jobs leverages the advantages offered by multiple layers in a neural network. However, these memory-intensive operations significantly challenge an efficient deployment of deep learning applications on serverless infrastructure.

As the basic unit of serverless computing, an action is configured with resource limits to ensure concurrency, scalability and security. The configuration of action limits varies among platforms, but differences are little. While there exists many platforms, this research focuses on Apache OpenWhisk [9] as it is an open-source system and is considered as a de-facto standard for conducting research in serverless computing [13]. On OpenWhisk, the default timeout is 60 seconds allowing up to a maximum of 300 seconds; any action

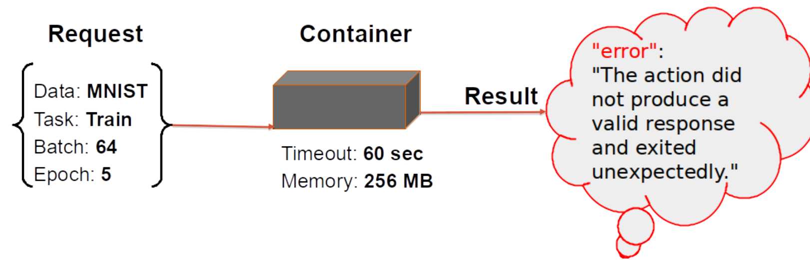


Figure 1.1: Failed action invocation using default memory allocation.

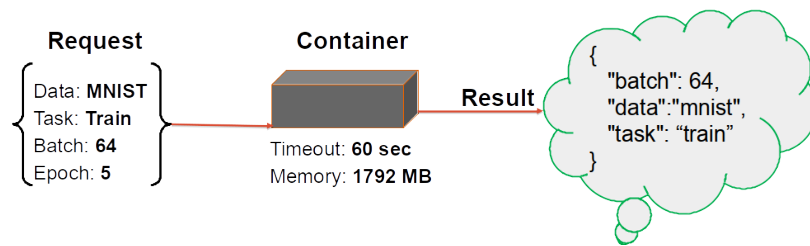


Figure 1.2: Successful action invocation after increasing the memory limit.

exceeding the defined timeout exits with an error. Similarly, a default memory limit per action of 256 MB applies. The maximum memory limit of 512 MB constrains the size of self-contained packages, and challenges the deployment of applications with large dependencies, e.g. of machine learning applications. OpenWhisk has other action limits related to logs, payload size, number of triggers, etc., but our research only addresses the timeout and memory resource limits.

To further illustrate the problem under consideration, we train a deep learning model for image classification on the MNIST [40] dataset using TensorFlow on OpenWhisk platform. The entire algorithm is written as a cloud function and therefore expected to fit in a single action. Results are reported on Figure 1.1 and Figure 1.2 representing respectively cases of error and successful completion. As shown on the figures, although the dataset size (32.06 MiB) is significantly smaller than the default memory limit, the action exits with an error message. However after several trials, setting 1792 MB as memory limit by adding the tag *–memory <value>* during invocation yields a success response. In fact, the allocation of more memory was necessary as each package imported within the function is loaded in the corresponding container, and takes an amount of memory before assigning the remainder

to the variables.

In spite of these constraints on resource utilization, developers have found ways to incorporate additional resources owing to OpenWhisk's open-source nature. Serverless distributed frameworks such as PyWren [35] and ExCamera [25] employ external services to complete fork-join operations in order to ensure consistency, but thereby violate the basic "no server management" policy of serverless platforms [13]. Another common approach is the use of in-memory storage for shared objects in order to minimize latency [14], and many applications use Redis [49] for such purpose.

In summary, unsuccessful completion due to timeouts or low memory availability are common scenarios when running deep learning jobs on serverless infrastructure. The purpose of our research is to address these challenges, and enable the migration of more deep learning applications towards serverless computing by reducing the failure rate observed with training deep learning models using serverless resources.

1.3 Research Objective and Contribution

In this thesis, we consider the need for a trade-off between cost, complexity, accessibility, and user satisfaction, and propose a distributed system for DDL using Apache OpenWhisk. Our main hypothesis is that *the efficient management of resource utilization using distributed computing strategies would ensure the reduction of the failure rate of deep learning's training jobs*. In order to verify that hypothesis, we intend to enhance OpenWhisk's core system with the capabilities to collect resource utilization metrics and allocate additional resource when requested by running actions. This approach is also supported by the implementation of an external module aiming at reducing the workload of individual actions through data parallelism and improving the rate of successful executions of training. Considering its widespread in the community, open-source nature, and support from a large number of contributors, our research employs TensorFlow as deep learning framework, creates models within the scope of TensorFlow Mirrored strategy [57], and deploys Apache OpenWhisk system on Kubernetes. A series of experiments on the proposed solution is

conducted on CloudLab [23] using a collection of test instances and evaluation metrics.

While important contributions were already made on serverless deep learning, to the best of our knowledge, research on training models using serverless resources are very scarce, and very few work exist on the deployment of deep leaning jobs on Apache OpenWhisk. Therefore, our research contributes in the following ways:

1. We focus on the time and memory resource constraints challenging to the utilization of large packages as in deep learning,

2. We propose a fine-grained design of the deep learning workflow, and introduce data parallelism to support the reduction of job failures.

3. Considering that machine learning on a distributed system usually requires users to configure parameter server and workers, this research offers a single entry point through command line interface and require user to simply specify the target dataset and job.

1.4 Layout of the Thesis

The remainder of this manuscript is organized into 5 Chapters.

In Chapter 2, we introduce the background of the research through a detailed review of existing theoretical and experimental advances in parallel and distributed computing and in serverless machine learning. Chapter 3 describes OpenWhisk and TensorFlow which are the technologies enabling deep learning using serverless architectures in the context of our research. Chapter 4 presents our proposed solution approach, and includes a detailed description of some failed attempts to modify OpenWhisk's core system. Chapter 5 exposes the settings of conducted experiments, explains in detail the different data-sets, test cases and evaluation metrics. This chapter also presents the outcome of the experiments, analyzes and discusses the computational results. Concluding remarks and future works are proposed in Chapter 6.

Chapter 2

Literature Review

Extending on existing technologies, this research involves theoretically and experimentally supported fields of research such as Distributed Computing, Machine Learning and Serverless Computing. This section provides a review of previous research on the frameworks, systems architectures, and strategies related to our topic. The following subsections will explore related works and comment on how they offer an effective background for the success of this research but could still benefit from further improvements.

2.1 Parallel and Distributed Deep Learning

Artificial Intelligence (AI) has existed since the 1950s [62], and is based on the assumption that human cognition can be described in detail and be replicated by a machine. Since then, AI has evolved into many sub-fields like Robotics and Machine Learning (ML). The latter constitute the most widely explored AI field of research, and led to the introduction of new topics like deep learning, whereas the former has revolutionized the manufacturing sector.

The rising complexity of deep learning models is tightly coupled with the advances in High Performance Computing (HPC) where strategies involving the sharing of multiple machines' compute resources have emerged to facilitate faster computation. As the community grows, there is a considerable amount of research in parallel and distributed systems for deep learning which leverages data, model, or hybrid parallelism to optimize resource utilization without compromising latency and space requirements [15].

Data parallelism. Data parallelism [32] is the most widely used approach among the

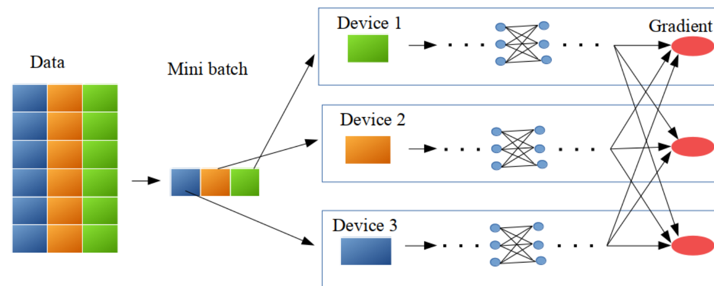


Figure 2.1: Illustration of the data parallelism strategy.

aforementioned parallel computing models for its simplicity and ease to modify. It is a data batch splitting approach which consists in assigning batches to different processors. Figure 2.1 shows data parallelism with TensorFlow where each compute node completes the entire program on a given set of data. Following the computation of the error function, a collective all-reduce is applied to update the gradient on every device and maintain consistency throughout all epochs. Among early DDL frameworks, DistBelief [21] offered significant performance, but as the number of parameters has grown exponentially, TensorFlow's release propelled a faster emergence of DDL. Subsequently, researchers increasingly focused on integrating more features to TensorFlow to yield new distributed architectures. For example, Message Passing Interface (MPI) was used and proved efficient as a communication interface for TensorFlow in a large distributed cluster [64].

Model parallelism. Model parallelism [41] consists in splitting the model and having each processor perform a specific task from the model's graph. This approach eliminates synchronization between devices for hyper-parameters update, but it requires data transfer between operators. An example of model parallelism is shown in Figure 2.2, where device placement is employed to assign a given operation to each device. Therefore, data is transported in its entirety, and operations are performed following their order of implementation in the model graph. In most cases, collective all-reduce communication occurs to update the gradient before launching the following epoch.

Expert designed parallelization strategies are also used to achieve high performance in

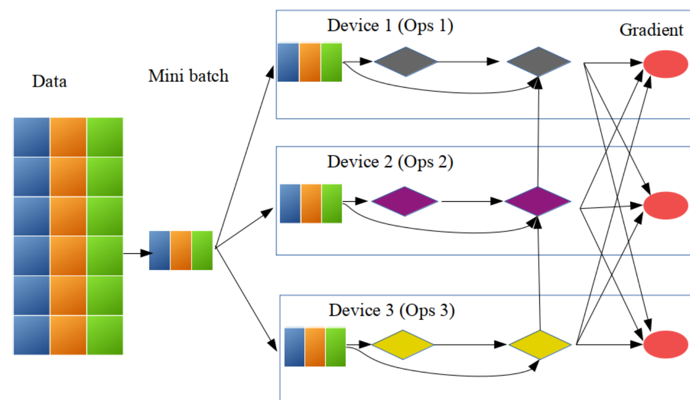


Figure 2.2: Illustration of the model parallelism strategy.

deep learning. Some researchers integrate data, model and other types of parallelism [26], whereas others use reinforcement learning for operator assignment in model. To reduce the level of uncertainty and non-suitability in the choice of parallelization strategies, some experts determine the most suitable in a search space through simulations. FlexFlow [34] was proposed to simultaneously integrate operators, parameters, data, and attributes of the model graph for tasks distribution. The system uses SOAP (Sample, Operator, Attribute, and Parameter) search space to determine the most effective strategy for parallel training. To this end, FlexFlow employs a fast, incremental execution simulator to evaluate different strategies, and a Markov Chain Monte Carlo search algorithm which takes advantage of the incremental simulator to rapidly explore the large search space.

In summary, the advent of HPC and the design of multiple parallelism models have facilitated the evolution of distributed deep learning. The aforementioned strategies proved efficient in different applications, but also bear certain disadvantages. Despite its popularity, data parallelism may need excessive memory and higher latency to synchronize parameters updates when training very large models. Model parallelism suits very large models [52], but the implementation of algorithms for model parallelization is complex due to the amount of involved operations. In view of these possible scenarios, several factors such as the type of the problem, the available memory resource, the concurrency limit,

etc., should be under consideration during the design of any distributed system for deep learning.

2.2 Serverless Computing

The abstraction of server management, the reduction of computation costs, the scalability of compute systems and the availability of resources on request are key features of serverless computing as defined by Amazon Web Services (AWS) [6]. For many years, this computing model has revolutionized ways of deploying software, and offers economic and engineering advantages to users from different level of technical skills. It provides an architecture which facilitates an ease-of-deployment and ease-of-management, and thereby attracts diverse applications irrespective of the programming language [4, 67, 53]. Many use cases of serverless applications exist and an increasing range of serverless platforms are being engineered by researchers and developers [8, 27, 9, 24]. Moreover, aiming at alleviating the deployment overhead of an application, orchestration frameworks have also been released [7, 43, 11], allowing developers to define compositions and launch sequences of actions with a single call. Comparing the overhead of sequential computations and parallel workflows, Barcelona-Pons et al. [13] used a benchmark of 5 to 320 parallel tasks to analyze the efficiency of Amazon Step Functions (ASF) [7], Azure Durable (ADF) [43], and OpenWhisk Composer [11]. Conducted experiments predefined 20 seconds as fixed duration for a task, and concluded Composer as better at performing parallel workloads.

Research related to serverless computing also focused on solving the cold start, communication, and storage issues, to name a few issues observed during the deployment of a software. Common to all existing platforms, the cold start problem refers to the extra-latency caused during the initialization of newly requested resources [42]. Openwhisk pre-warms containers to reduce the firing time, but only few container runtimes are cached because they use additional resources. Consequently, solutions ranging from agile implementations to an additional application layer are introduced by researchers to achieve a better performance. Mohan et al. [44] introduced a strategy to pre-warm and cache network endpoints,

and bind the latter to containers upon requests for creation. This procedure consumes less space and reduce the initialization delay. From another perspective, Bermbach et al. [16] proposed three different additional mechanisms on top of the serverless platform to leverage the knowledge of an action invocation to cold start the containers before sending the request. Following experiments using AWS Lambda and OpenWhisk, authors conclude a 40% reduction of cold start delay.

2.3 Serverless Machine Learning

Within few years of its advent, serverless computing has emerged as an efficient model to perform a large number of parallel tasks. The computation intensity of machine learning and serverless platforms' capability to perform event-triggered executions led some researchers to develop and evaluate serverless architectures for machine learning.

Related research primarily focused on lightweight computations, specifically on edge computing [47, 46] and serving models for inference [42, 28, 20]. For example, Lin and Glikson [42] deployed a cat/dog image classification model on Knative [28] for inference using TensorFlow, while Ishakian et al. [33] employed AWS Lambda to serve large deep learning models with TensorFlow serving. Rausch et al. [47] explored the feasibility of Edge AI workflow on serverless platforms and proposed a serverless model using edge devices as cluster resources and forming an edge cloud platform. Palade et al. [46] defined a hypothesis that the incorporation of serverless computing at the edge of IoT devices for small tasks will improve their processing time. Accordingly, the authors evaluated qualitatively and quantitatively the performance of frameworks such as Kubeless, OpenWhisk, OpenFaaS and Knative, and concluded that Kubeless offers better response time and higher throughput. Although those research concluded the possibility of machine learning on serverless infrastructure, yet they only consider a single phase (inference) in the entire machine learning workflow.

Exploring the structure of training jobs in neural networks and the challenges to serverless computing, Feng et al. [39] argued that serverless runtime is also beneficial to training small models, and minimizing data transfer between subsequent actions increases the performance of the platform. Carreira et al. [19] expanded the design of serverless architectures to end-to-end machine learning to develop Cirrus, a system which integrates a stateless server-side back-end and addresses resource constraints as well as workers' scalability issues. The authors designed Cirrus to meet three critical goals defined in their preliminary work [18]:

- A high level language based API;
- A low latency and scalable data storage; and
- A lightweight and high performing worker runtime.

Some of the existing serverless platforms for ML do not support worker failure [18], increasing the probability of sudden job interruption, which in turn would impact the accuracy of trained models. Consequently, many developers use additional supports such as external server disks, in-memory data storage like REDIS [49] or cloud data storage like Amazon S3. In-memory data storing has proven more efficient in terms of latency compared to the other options [14], and is employed by a large community of practitioners.

In summary, numerous researchers focused on exploring ways to achieve efficiency and performance in machine learning using serverless architectures. Nevertheless, while some authors address a segment of the machine learning workflow, the review of related work also displays an imbalanced attraction towards commercial platforms like AWS Lambda. Among other reasons supporting its selection, authors accredit better resource constraints. This can also be proven by the comparison of AWS Lambda limits [5] and Openwhisk system limits [45] which shows that AWS Lambda offers more resources. For example, while the maximum memory allocation on the former is 3008MB, the latter just allows up to 512MB. In addition, as a commercial platform, AWS Lambda provides a better interface

and easier configuration. Although commercial serverless platforms facilitate the integration of an application with other systems such as databases, there exists many developers with needs for small scale deployments, or just for experimental purposes in academic and research settings. Under such circumstances, open source frameworks are more effective, as they offer a larger breath for system-level modifications in order to adapt the platform to different problems.

2.4 Chapter Summary

2.4.1 Limitations of Current Approaches

The preceding literature review evaluates research work which are related to distributed deep learning and its achievement using serverless resources. As can be seen, earlier systems have benefited from a significant amount of enhancement, which led to the wide application of distributed computing approaches in deep learning, and in recent years on serverless platforms. However, while the majority of research on parallel and distributed deep learning proposed problem-specific or job-specific designs, little attention was given to resource utilization for distributed deep learning on serverless platforms.

Meanwhile, the few research articles on distributed serverless deep learning have focused on training models. In fact, many authors have argued that this type of jobs poses a challenge to the deployment on serverless platforms as it requires more data, more operations, and subsequently more memory. Training also necessitates the underlining architecture to accommodate the statefullness of hyper-parameters.

The consideration of these problems would drastically improve computations' efficiency as well as user experience, and offer multiple research opportunities especially in this era where the majority of computations, both in academia and industry, are moving towards cloud and edge computing.

2.4.2 Common Challenges

Many limitations of existing serverless platforms undermine the effectiveness of an end-to-end fully server-free deep learning job. Among other issues, the memory and compute time constraints predominantly influence the design of solutions to address the problem under consideration. These two challenges can be described as follows:

1. **Memory size limit.** Existing serverless frameworks have hard memory constraints conditioning running functions with a higher data storing requirement.

2. **Compute time constraints.** Deep learning jobs consume more time, but actions are constrained to complete within given time windows, often defined for few thousands of milliseconds.

At the core of our research, we propose and implement approaches of solutions to bypass these obstacles in order to yield a fully-functional software for end-to-end distributed deep learning.

Chapter 3

Enabling Technologies

3.1 Apache OpenWhisk

The revolution of open-source software development has been a strong enabler of cloud computing [29], yet fewer serverless platforms are available as open-source projects. One of the few open-source serverless computing platforms, Apache OpenWhisk was developed as a joint project by IBM and Adobe and latter released under Apache Software Foundation's flag as open-source project.

OpenWhisk supports multiple concurrent cloud functions with key-value outputs, where each function, also referred to as action, is deployed in a docker container. However, an invoked action is restricted from communicating with any other running one. The invocation often follows a threefold procedure involving the emission of a user request, its validation and assignment, and the execution of the requested action. This process can be expanded to a more comprehensive approach including an event source, a trigger, some rules, and the target action. Illustrated in Figure 3.1, the event source can be a web-based request, an external feed from a database, a Git update, or commands executed by the user through OpenWhisk CLI. An event would automatically fire a defined trigger in order to complete a given action or sequence of actions. The developer can also optionally specify rules stating actions to execute, or use a single trigger to call for different actions. It is important to note that actions and triggers are not always linked, thus any user can invoke an action without defining a trigger.

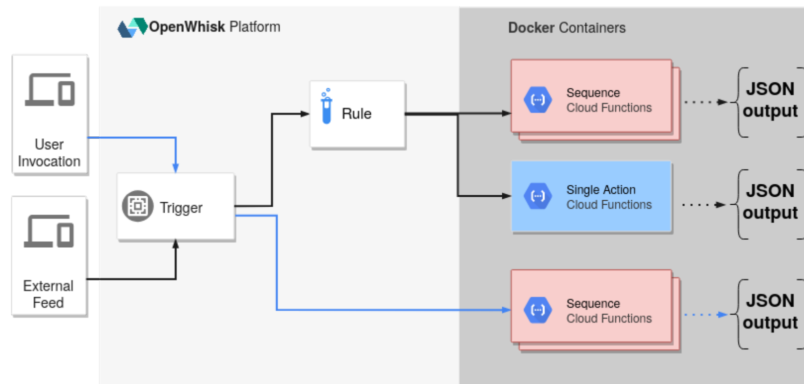


Figure 3.1: OpenWhisk action invocation workflow.

3.1.1 System Workflow

The architecture supporting OpenWhisk's operations integrates several well-know software stacks such as Nginx, Kafka, Docker, CouchDB, Kubernetes, to name a few. Each component plays an important role backing the two main entities of the serverless computing platform which are the controller(s) and invoker(s).

OpenWhisk can be deployed through a standalone platform using Docker containers or with Kubernetes to manage the entire cluster. While the former option is straightforward and simpler, the latter is more resilient for middle or large scale clusters. Figure 3.2 depicts what happens behind the scenes of Openwhisk system, showing the interaction between the enabling packages. Nginx, a high performance open-source web-server, opens the system to the outside exposing HTTP services to clients. This server is primarily used for SSL termination and a reverse proxy to forward the client's request to the controller.

At the core of OpenWhisk, the controller implements a REST API to authenticate and proceed with the system's orchestration. Upon reception of a call, the controller checks the credentials of the user, and verifies the authenticity of the user's request on the basis of data stored in CouchDB. After validation, an activation message is generated and published by the loadBalancer to an invoker through Kafka. In addition to the loadBalancer, OpenWhisk's controller encompasses a leanBalancer which directly communicates with

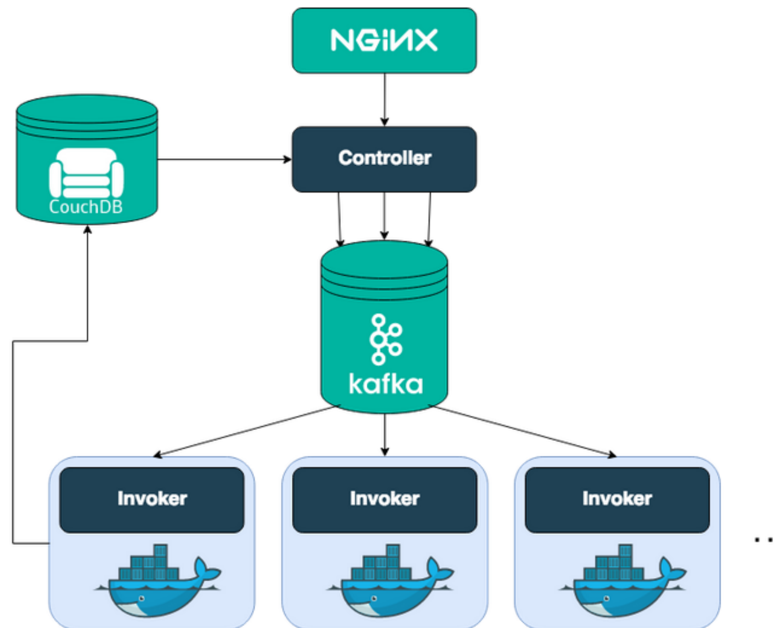


Figure 3.2: Behind the scenes of OpenWhisk operations [10].

the invoker without Kafka as intermediary. The loadBalancer has a holistic view of the entire system, and the workload of the invokers, hence proceeds to the selection of the most appropriate one considering its availability and features.

The invoker, manager of the pool of containers, picks the activation messages from Kafka or through feeds, collects the code corresponding to the action from CouchDB, and proceeds to the invocation. In comparison to the controller, the invoker is the closest entity to the action's container. Consequently, its in-depth evaluation is conducted in the following chapter in order to examine possible improvements for a better memory management. Meanwhile the following section describes the operations of the invoker.

3.1.2 OpenWhisk Invoker

The invoker is the final stage in the OpenWhisk flow of processing actions. Upon detection of an activation message through the procedure described in Algorithm 1, a container with the runtime language corresponding to the action is selected among warmed containers, or created if none is available for use. OpenWhisk supports several programming languages

Algorithm 1: OpenWhisk Invoker workflow.

Data: Command line arguments: *invoker [options] [proposedInvokerId]*
Result: Void *(Launches server and listens for activation messages)*
basic system configuration;
Kamon shutdown;
load properties from the environment;
if *properties and runtime manifest are valid* **then**
    initialize execution runtime manifest;
get command line arguments and assign invoker ID;
initialize Kamon with the assigned ID;
instantiate the invoker provider;
launch Invoker server;

such as NodeJs, Java, Python, Go, Swift, PhP, and has docker images for each runtime hosted in its repository. After the selection or creation of a target container, the action code is extracted from CouchDB, and injected into the container for execution. In the end, results and logs are collected and stored back into CouchDB. The container is afterwards destroyed or kept warm for future use.

The direct interaction of the invoker with the pool of containers makes it an ideal target for understanding how action memory is handled in the system. Through the analysis of source codes, and as Algorithm 2 shows, we noticed that an invoker is instantiated with the configuration of the entire container pool, and has access to the user memory limit.

3.2 TensorFlow Framework

3.2.1 Distributed Computing Using TensorFlow Framework

TensorFlow [1] is a machine learning platform which is widely used in the community by developers and researchers for fast computation and efficient completion of jobs like image classification, pattern recognition, regression, etc. TensorFlow allows developers to perform data, model or hybrid parallel computations with very high accuracy for many problems while maintaining a simple parallelism design. It has extensively been improved

Algorithm 2: OpenWhisk Invoker Provider workflow.

Data: WhiskConfig, InvokerInstanceId, MessageProducer, conttainerPoolConfig, ConcurrencyLimitConfig

Result: Invoker Provider

Execution context and whisk configuration;
instantiate a container factory to provide container proxy actor;
initialize container factory;
clean runtime containers if any existed;
initialize necessary databases;
initialize message consumers;
define ack messages transmision pipeline;
instantiate a container proxy actor which will be parsed to the container pool and used when containers are called;
define an activation feed with a method to process activation messages read from Kafka(refer to Algorithm 3);
create the container pool with the proxy, pool configuration instructions, and the activation feed;

since its initial release by Google Brain in 2015 as an open-source software, and currently counts variants such as TensorFlow Lite, TensorFlow Extended, Mesh TensorFlow, etc.

TensorFlow Lite

TensorFlow Lite (TFLite) [58] is the lightweight version of TensorFlow, designed for mobile and embedded devices. The architecture consists of a converter which transforms a TensorFlow model into TensorFlow Lite format, and an interpreter which reads and works with the converted model. An illustration of the workflow is provided in Figure 3.3. Note that currently only sequential inferences are implemented, offering research opportunities for distributed training on mobile devices.

Algorithm 3: How activation messages are processed.

Data: Activation message
Result: Sends request to container
get transaction ID;
set trace context;
load properties from the environment;
if *action namespace not blacklisted* **then**
  get the action;
  create action executable;
  send Run(executable, message) request to the poll for assignment to an idle container;

TensorFlow Extended

TensorFlow Extended (TFX) [59] is a distributed architecture ensuring a rapid software production where the developer distributes tasks like transforms, model analysis and serving to different components of the distributed computing platform. The framework incorporates Apache Beam, and supports Flink, Spark or Dataflow to ensure data transmission across the entire pipeline.

As shown in Figure 3.4, TFX uses a driver to read from a metadata store, where information about the model is kept, and passes it to an executor. Executions of tasks mentioned above can run in parallel before sending the ready-to-serve product to the publisher. The publisher uploads the data back to the metadata store. Note that the executor does not read or write directly to the storage, thereby it can avoid read-write problems which may emerge out of running tasks in a distributed environment.

Mesh TensorFlow

Mesh TensorFlow (MTF) [51] incorporates both data and model parallelism. It is efficient at addressing problems with a large number of parameters, attributes, operators. This method can stipulate the type of parallelism through the setting of the mesh shape and the computation layout. As well, it can define the size and configuration of the computational

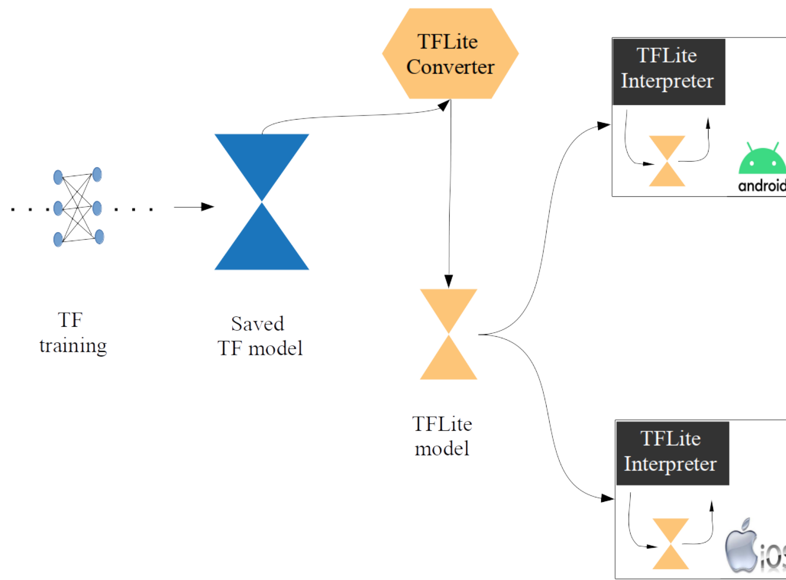


Figure 3.3: Flowchart of TensorFlow Lite operations.

resources as 1D, 2D, or moreD meshes. A Mesh TensorFlow graph consists of parallel operations coupled with all-reduce as collective communication.

Distributed TensorFlow

Distributed TensorFlow consists in distribution strategies such as Mirrored, Parameter server, One-device, Multi worker Mirrored, to name a few, and follows a dataflow computational model. Operations are represented as a direct acyclic graph replicated on each compute node. The process is common to the majority of the strategies, except parameter server which employs a slightly different approach to update hyper-parameters. The cluster specification, in case of the parameter server strategy, requires the definition of at least one device as parameter server. At the end of each epoch, every worker device sends its gradient to the designated parameter servers to update the global value. Similarly, devices refer to the parameter servers to collect the necessary parameter for a given computation. Thus, less space will be used on each worker, however, it may create a communication overhead owing to the number of concurrent requests [15]. The mirrored strategy [57] is a

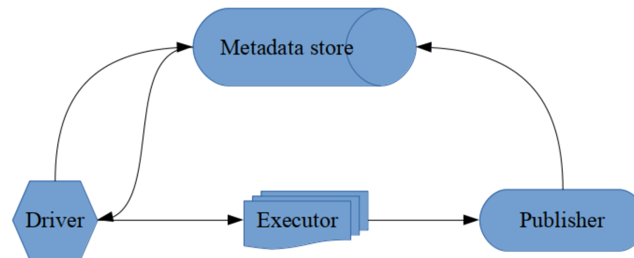


Figure 3.4: Flowchart of TensorFlow Extended.

base model in Distributed TensorFlow where variables are maintained as mirrored objects across devices. The common reduction methodology is an all-reduce approach which mirrors the variable onto all devices forming the compute cluster. All-reduce is analogous to broadcasting a value to all components of a network. In TensorFlow, a value-destinations pair is maintained and used to ensure that the same value is broadcast to all the devices.

3.2.2 Distributed TensorFlow Performance Analysis

We evaluate the performance of Distributed TensorFlow through the computation time taken per iteration, and per batch size. For comparative analyses, we respectively trained an image classification model with a sequential program and within the scope of the MirroredStrategy distribution. One CPU-enabled node from CloudLab hosting Ubuntu 18.04 was instantiated. We imported version 2.0.0b0 of the TensorFlow framework, and defined other parameters as following: data $=\{$MNIST, CIFAR-10, CIFAR-100$\}$; epochs = 5, for visualization and interpretation purposes; batch size $=\{8, 16, 32, 64\}$; buffer size = 10000.

Computation results are collected and presented on Figures 3.5 and 3.6 for comparison. Accordingly, our observations and conclusions can be summarized in the following points:

- An increase in the batch size yielded more computation speed. From a distributed computing standpoint, this observation implies the reduction of communication overhead because larger batches minimize the number of parameter updates. In addition,

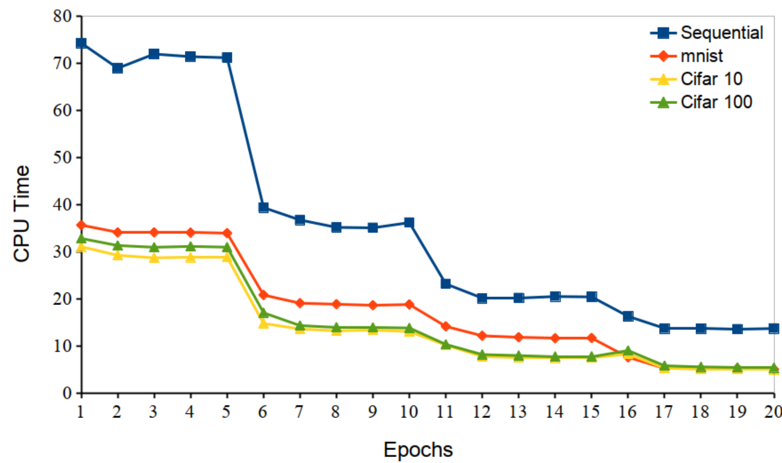


Figure 3.5: Variation of computation time with respect to the epoch parameter.

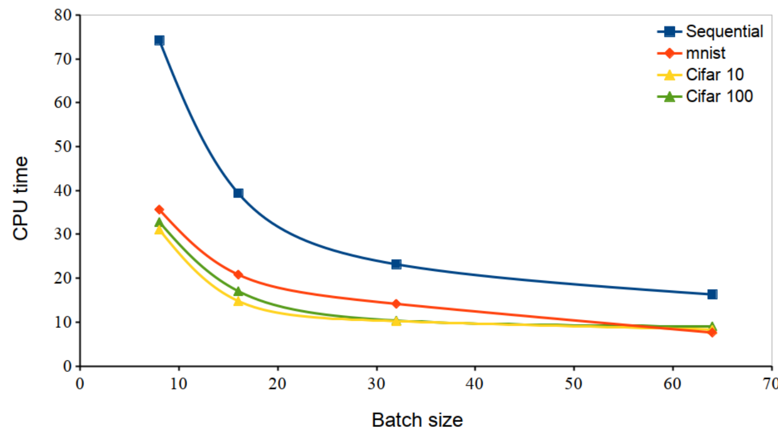


Figure 3.6: Variation of computation time with respect to the batch size.

larger batches improve randomness in the training, which does not only avoid overfitting, but also allows faster convergence. However, the batch size should not be extreme as doing more job at a time will impact the accuracy of the output.

- As the index of the epoch increases, less variation is observed in the computation. However, there is an improvement in the accuracy and reduction of the loss. On the one hand, it illustrates the process of gradient descent, where the update of the gradient at the end of an epoch allows better performance during the next one. On the other hand, it explains the iterative process where the entire dataset, in the form

of mini batch, goes through the neural network at each epoch.

- Distribution of tasks to multiple compute nodes allows faster computation. The co-ordination of each node's resources reduces their workload, and consequently allows to make an efficient use of the CPU cycles for more operations.

Chapter 4

A Scalable Serverless System for Deep Learning

With regards to OpenWhisk timeout and memory resource constraints, our research aims at enhancing the system and providing the most efficient approach to train deep learning models and/or infer testing data. Therefore, we explore two different enhancement strategies, and analyze their feasibility taking into account the design of the current platform.

4.1 System-Level Improvement

The overview of Apache OpenWhisk presented in Section 3.1.2 has introduced the factors which define the selection of a pre-warm or warm container to execute requested functions. We emphasized on the choice of a container which kind (runtime language) matches the invoked action, but the memory requirement of the request is also considered. As a matter of fact, an action can be configured by the user with a desired value for memory allocation, and it is part of the invoker's responsibilities to ensure that the selected container satisfies the request. Henceforth, not only does the kind of a warm container play a role in deciding whether or not to cold start a new container, but the capacity of the latter is also assessed.

A record of the memory allocated to active containers is maintained by the invoker. Using a pool hierarchy to track free, busy, pre-warm starting and prewarmed containers, an OpenWhisk invoker communicates with managed containers through their proxies, and established a contract to define the interaction. The container proxy is a representation of

the running container, and is used to route connections that would have otherwise directly reached the invoker. While communication is achieved through HTTP, the proxy contract defines the content of the request as well as actions to perform on the basis of the response.

The contract between the invoker and the container proxy is implemented using Akka Actor model [2]. The interaction is interpreted by a Finite State Machine (FSM) [3] using a set of relations as follows: *In the occurrence of Event E in State S, perform Action A and move to State T.*

$$State(S)\ x\ Event(E) \rightarrow Action(A),\ State(T)$$

Figure 4.1 describes the contract established with the proxy in order to efficiently manage resources. A container can be in one of eight different states (*Initialized, Starting, Started, Running, Ready, Pausing, Paused and Removing*), and one or more actions can be performed following the occurrence of an event. For example, following transmission of a *Run* request with pre-warmed data, a started container changes its state to *Running*. The current system configuration allows the container proxy to receive *Start*, *Run*, *Remove* and *HealthPingEnabled* requests from the pool, respectively to initialize, launch computations, reclaim the container, and check if the container is healthy. In return, the container proxy can send to the invoker requests such as *NeedWork*, *ContainerPaused*, *ContainerRemoved*, *RescheduleJob*, *PreWarmCompleted*, *InitCompleted* and *RunCompleted*.

A running container is allowed to request for more work from the container pool. In view of the possibility of such action, can we redirect this mechanism to request for more memory? In order to evaluate the feasibility, we formulate the following proposition and verify them on OpenWhisk.

Proposition 1. *Let the container proxy periodically check for memory usage, and send a NeedMemory request to the pool when the remaining memory is critical.*

The implementation of a resource monitor in the container proxy is our first attempt to achieve a dynamic memory management. As shown in Algorithm 4, the container would periodically get the remaining memory, checks if there are less than 10 MB available. Be

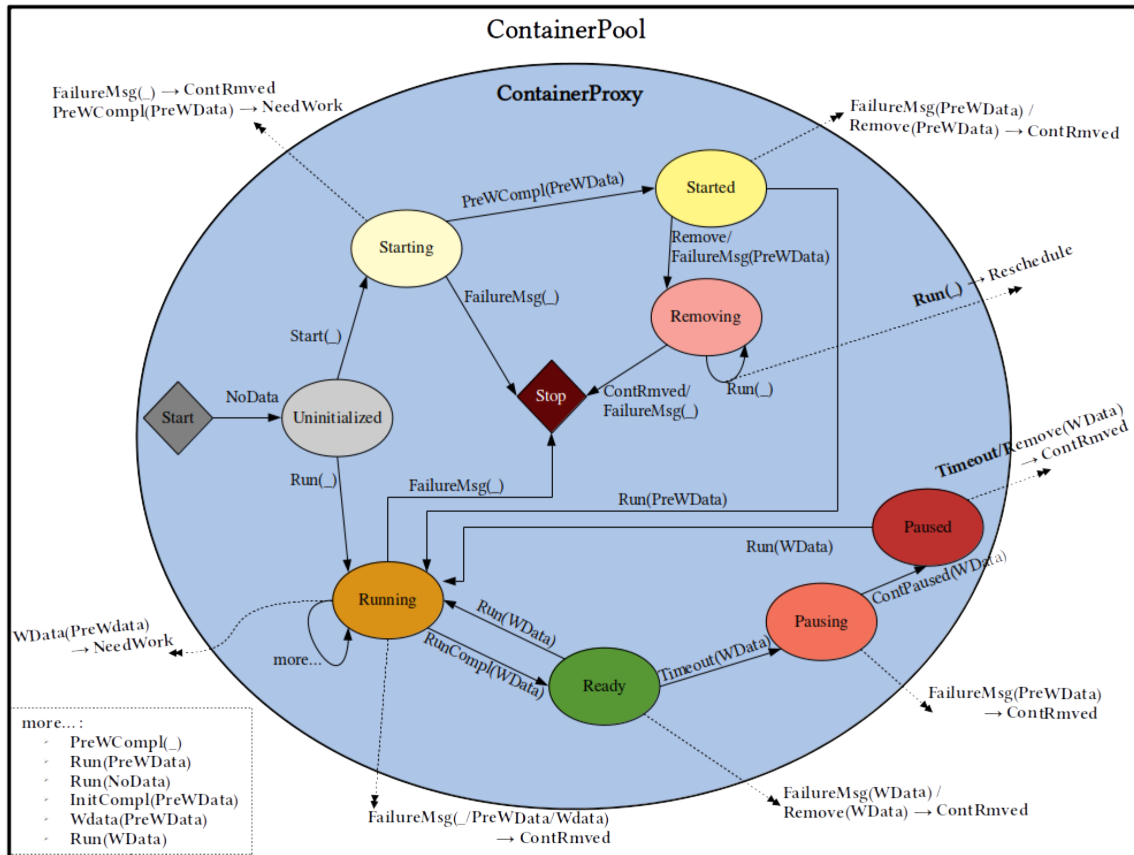


Figure 4.1: Finite state machine illustrating the container pool-proxy contract.

the result true, the container would notify the pool through a NeedMemory request. The container's information is encapsulated inside a warmData variable which is forwarded along with the request. In case of a NeedMemory event, Algorithm 5 shows that the pool checks if it has enough memory to assign an additional 256 MB to the requesting actor, otherwise assign the amount of free memory in the pool. In order to successfully compile these algorithm, we had to also modify the variable holding the memory limit of an action from "val" to "var" as the former defines immutable variables while the latter allows modifications after the variable's definition.

Although this approach maintains the consistency of resource inventory on every actor of the transaction, it failed to collect the accurate memory usage because the container proxy does not operate inside the docker container where the action is run. Henceforth, the

Algorithm 4: Memory resource monitor.

```
while status is running do
    get remaining memory;
    if remains less or equal to 10.MB then
        ContainerPool ← NeedMemory(WarmData);
    end
end
```

Algorithm 5: ContainerPool handling of NeedMemory requests.

```
if Event: NeedMemory then
    if hasPoolSpaceFor(freePool, 256.MB) then
        // If there is not enough memory, increase by 256.MB;
        warmData.increaseMemoryBy(256);
    else
        // Remove a a too cold container to free up space in the containerPool;
        availableSpace = Math.min(256, memoryConsumptionOf(freePool)).MB;
        ContainerPool.remove(freePool, availableSpace).map(removeContainer);
        warmData.increaseMemoryBy(availableSpace.toMB.toInt);
    end
end
```

resource monitor just keeps track of what is internally happening inside the invoker pod, which is not the desired behavior.

Lemma 1. *The container proxy, entity which holds metadata about the running container is hosted by the invoker pod. Therefore, we cannot monitor the active memory usage of a running container from the proxy.*

Proposition 2. *Aware that actions run inside docker containers within a different pod from that of the invoker, let adapt the existing communication strategy between the running docker container and the invoker and incorporate a warning notification in case of critical memory consumption level.*

As mentioned above, the invoker uses HTTP to send requests to the container. Two routes, namely */init* and */run*, are available in the existing system architecture respectively to initialize the container and launch the execution of the action. The expansion mentioned in Proposition 2 involves modifying the behavior of the *run* function executed upon reception

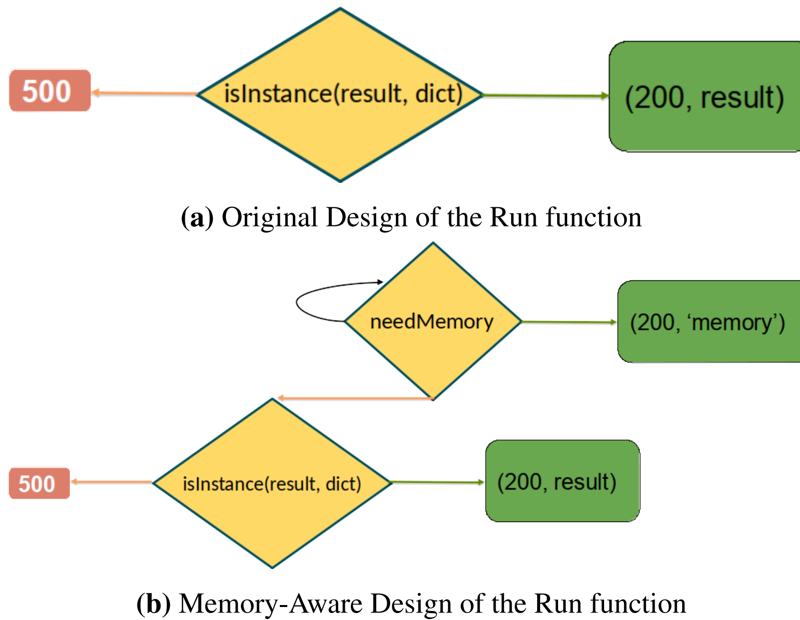
Algorithm 6: Implementation of the run function.

set parameters in the global context;
launch memory monitor; //This step will be added for our metrics collection;
execute code in global context;
get results of the execution;
if *result is a dictionary* **then**
    return result with a 200 status code;
else
    return a 502 status code with an error message;
end

of a request through the */run* route. The resource monitor is launched inside the container before the execution of the code, as shown in Algorithm 6. Once the monitor detects the usage in the critical region, our implementation summarized in Figure 4.2 requires the container to return a 200 HTTP status code with the word "memory" as a warning, and continue listening until the execution of the code is completed. The container proxy then forwards a NeedMemory to the pool which update the entire deployment accordingly. Our choice of the response status is defined by the current system in which the invoker considers any response with a 500 status code as a failure, and removes the corresponding container from the pool. Willing to maintain the action running until completion, an HTTP response with a 200 status code appears ideal.

In part, this system enhancement works because the memory usage of the running container is accurately monitored. However, setting a new memory limit was not successful as the underlying deployment environment is Kubernetes, and the resource limits of a running pod cannot be modified. The destruction of the running pod, cold starting a new one and resuming the action from beginning does not present an efficient strategy as it infers a longer execution time and worse results.

In summary, an OpenWhisk deployment on Kubernetes creates separate pods for the



(a) Original Design of the Run function

(b) Memory-Aware Design of the Run function

Figure 4.2: Enhancement of the Run function within OpenWhisk's docker runtime image.

invoker and the docker container hosting the execution of the invoked action. The communication is established and maintained through HTTP services, and we could monitor the utilization of either pod successfully. However, the modification of the memory resource limits of the action's pod failed as Kubernetes does not support the update of a running pod's resources. The execution of the command *kubectl patch pod wskwsktf-invoker-00-1-prewarm-nodejs10 -n openwhisk -p '{"spec":{"containers":[{"name":"user-action","resources":{"limits":{"memory":"512Mi"}}}]}}'* aiming at externally modifying the memory limit of a testing pod "wskwsktf-invoker-00-1-prewarm-nodejs10" failed with an error which declares that *The Pod "wskwsktf-invoker-00-1-prewarm-nodejs10" is invalid: spec: Forbidden: pod updates may not change fields other than 'spec.containers[-*].image', 'spec.initContainers[*].image', 'spec.activeDeadlineSeconds' or 'spec.tolerations' (only additions to existing tolerations)*. Although we can destroy the running container an assign more memory before redeploying, yet this approach would undermine the efficiency of our solution because there would be a significant and inefficient memory resource utilization. Considering the risk of a worse performance, our research has expanded

to explore the feasibility of an external support module and the implication of its implementations in the following section.

4.2 Distributed Serverless Deep Learning Module

A deep learning workflow is an iterative process which improves the efficiency of the trained model on the basis of hyper-parameters update. Employing serverless computing to perform such tasks necessitates the ability to synchronize actions and accordingly update parameters. However, not only is this feature unavailable on OpenWhisk, but communication between actions is also restricted, thereby requiring for an outside support. Moreover, the timeout and memory resource limitations enforced by serverless platforms expose deep learning jobs to a higher risk of unexpected termination, compromised results and worse accuracy. Considering the challenges of a dynamic memory allocation at the system level, our research proposes a distributed module to provide scheduling and task assignment support using a fine-grained system design. We introduce DISDEL, a distributed serverless deep learning module, which employs data-parallelism to ensure a fine-granularity of deployed actions, and computes containers memory resources based on an SAP (Sample - Attributes - Parameters) feature set. These features are factored to determine the appropriate memory allocation of a container in order to avoid over-provisioning or under-provisioning as the former would lead to a poor resource utilization and the latter to the interruption of running containers.

4.2.1 System Interaction

The proposed architecture follows a workflow applicable to many problems be it image classification, object detection, text processing, etc. In view of storage constraints of cloud actions and aware that communication between actions is not supported on existing platforms, our strategy combines both client and serverless resources. As shown in Figure 4.4, the entry point into the system is a command line interface where users submit their requests by providing the target dataset and the job type, and optionally specify the desired

batch size and the number of epochs. In case of omission of optional parameters, default values of 64 data entries per batch and five epochs are applied.

A user request is received by a controller, which is the main component of our additional module, and the only part which is launched on the host system. As the name specifies, the controller processes the request and proceeds accordingly. An example is used in Figure 4.4 to illustrate a training request from a user who targets the mnist dataset. First, an initialization phase occurs and consists in reading the parameters, computing the number of actions to create, and creating a package to wrap the entire composition. Another use of the package is to define all the parameters as a dictionary which can be inherited and accessed by invoked actions. Second, actions are created. For the example under consideration, only the training and collectWeights actions are created, but if the requested job were an inference, an additional action would have been created for the evaluation phase. Third, training actions are invoked concurrently with a blocking call maintaining the main thread until all worker threads join. Each action stores the trained model in a Redis in-memory data store for later use. The fork-join mechanism is employed to ensure that more data points are used to build a more accurate ensemble model. Following the join operation, the collectWeights action collects all trained models from the data store for a weighted averaging of each layer and generates an ensemble model. This model is stored back into Redis, and can be used at time of inference.

4.2.2 Custom Action Runtime Image

Considering the different entities enabling deep learning on OpenWhisk, the default stack consists in the execution of docker containers inside Kubernetes pods, with an underlying docker image built from the base OpenWhisk docker runtime image. Under several circumstances, images specific to the runtime of the requested action are often used, or recreated as custom images with additional packages to accommodate applications which import additional modules. Nevertheless, it is important to point out that although OpenWhisk is compatible with custom runtime images, the system requires the implementation

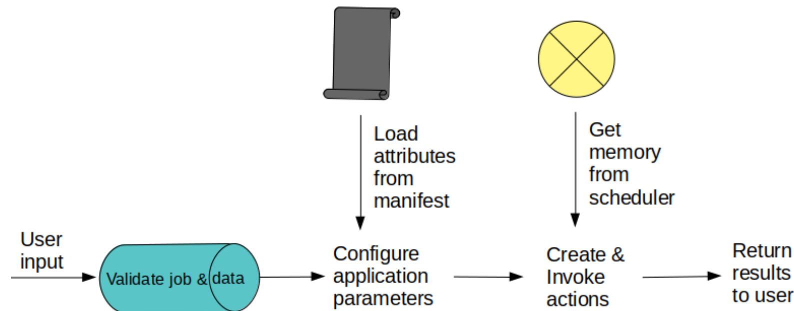


Figure 4.3: Workflow of the controller of our proposed module.

of the action interface, an interface where routes for invocation requests are defined.

The default python action runtime image does not contain TensorFlow framework, thus, it is imperative to parse a custom docker image with the *–docker* flag at creation. However, in our setting where many actions are concurrently deployed, we reduced the overhead of parsing the image each time by building a custom image which contains all the required modules for the python runtime along with TensorFlow, TensorFlow-datasets and Redis. For simplicity and ease of reproduction, we adapt OpenWhisk's image for python actions[1], and define the resulting custom image as default runtime for python actions in the invoker pod's manifest.

4.2.3 Controller

At the core of DISDEL, the controller is the main processing unit of our proposed module. Its primary functions, as shown in Figure 4.3, are the reception and authentication of the user's request, the generation of application configuration parameters, the transmission of a request to the scheduler to determine the memory allocation, the initiation of the fork-join operations, and the transmission of the result back to the user. The validation of the job is a simple comparison to ensure the entry is either train or infer, whereas that of the dataset ensures the target data is available in the TensorFlow Dataset Catalog [56].

[1] Available using *"docker pull openwhisk/python3action"*

Data is a critical ingredient in every deep learning model training or inference. The availability and quality of collected data does not only impact the execution's duration, but it also influences the accuracy of the generated model. Our goal being to enable the training of medium and large datasets on serverless platforms without the failure of any invoked container, we implemented our module around a data parallelism approach. Based on the number of actions, the controller splits the target dataset into smaller chunks which would fit in a single action. The fitness of the obtained data size is evaluated with the assistance of a scheduler which determines the most suitable memory allocation for the data. More on the scheduler is provided in the next section. Upon reception of the memory to allocate, the controller creates and invokes serverless actions which are executed on OpenWhisk. Finally the result of the execution is collected and returned to the user.

4.2.4 Scheduler

Training a deep learning model is a memory-intensive job. From the assembly of the model to its compilation and fitting, a significant amount of the host's resources are consumed, and the shortage in the amount of memory needed for an operation leads to the sudden interruption of the entire training. In this section, we introduce our scheduler, one of the main components of DISDEL, which has the primary responsibility of partitioning memory based on a set of features.

The memory computation considers three different features which are the Sample (number of examples/records in the dataset), Attribute (input size, number of output classes, number of channels in a given sample), and Parameter (number of trainable parameters at each layer). The defined model is of three layers (one convolution layer and two dense layers) with a max pooling phase after the first layer. Therefore, the total number of parameters is computed as the aggregation of each layer. At the convolution layer, the number of trainable parameters is computed using $p = (m \times k^2 \times n) + n$ [36] where m designates the number of input channels, k is the filter size and n the number of output channels. Subsequently, we estimate the memory usage of fitting the model as $fit = e \times p \times s$ where e

represents the number of epoch, *p* refers to number of parameters and *s* denotes the number of steps during training.

Considering the number of parameters and the memory estimation for the *fit* function, we include the size of the generated runtime image *pkg*, the size of the dataset *dta*, the size of the compiled model *mod* and a safety limit *saf* in order to compute the total memory estimation of an action as $memory = fit + pkg + dta + mod + saf$. The purpose of the safety limit is to ensure that there is enough space to hold generated variables, the directories containing logs and checkpoints, and a *.h5* file storing the generated model.

4.2.5 Model Training and Averaging

The maximum user memory on OpenWhisk is defined by default as 2048 MB, which is the maximum memory allocated to a pool of container as well. Aware of the memory usage of the custom runtime image, the compiled model, logs and checkpoints, training a large dataset in a single action represents a significant risk of execution failure due to the lack of memory. However, the restriction on inter-action communication prevents the concurrent use of multiple actions to perform all-reduce updates of parameters.

We propose a fork-join strategy in which training is performed concurrently on portions of the dataset inside different containers. Upon completion of all training jobs, we invoke a single action to collect all models stored during training actions from a designated data store, proceed to their weighted averaging (equal weights are used in our implementation), and return the ensemble model to the data store for future use during inference. With regards to the traffic between actions and the storage, the selection of a suitable intermediate storage proves critical as it may incur additional delay. For the sake of faster insertion and retrieval, we employ Redis due to its efficiency as proved by many researchers and its popularity in the open-source community. An experience with 1 KB payload also supports the performance of Redis as the average latency of put and get requests to an Amazon S3 database largely exceeds storage on Redis [14].

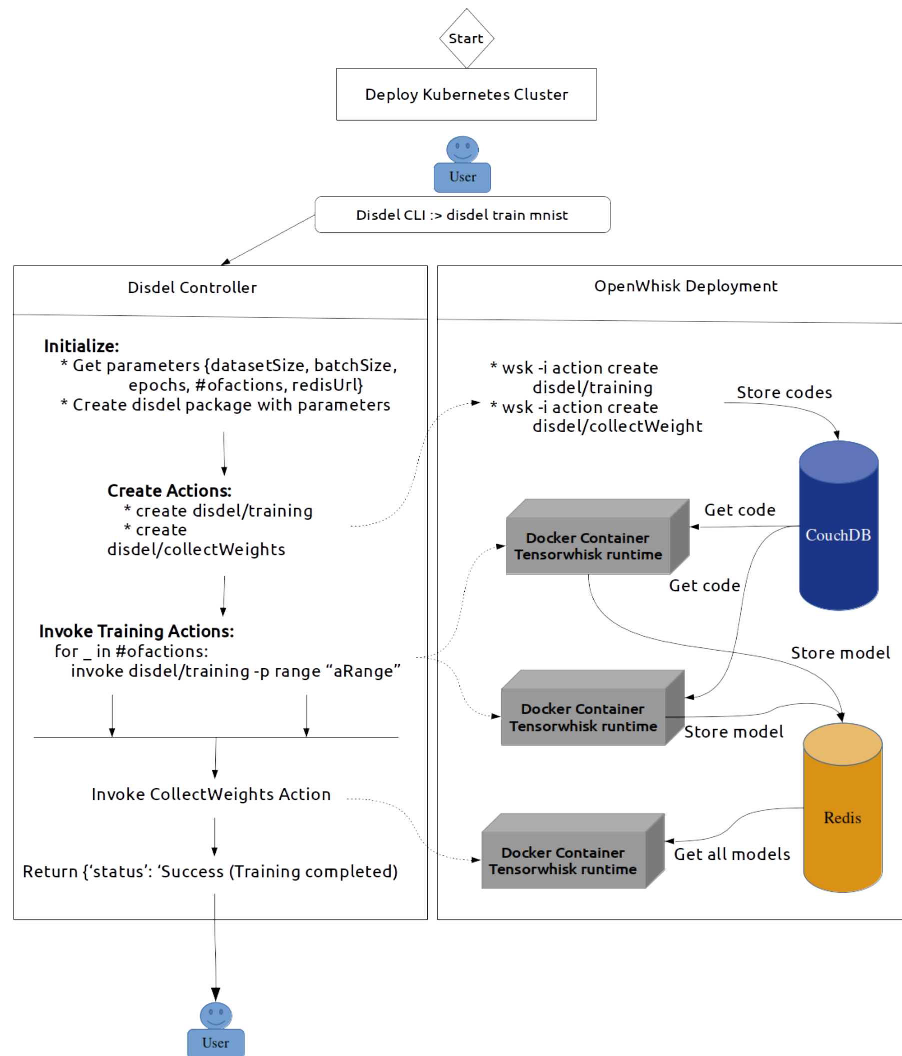


Figure 4.4: Distributed deep learning workflow using serverless resources.

Our strategy does not only yield an efficient performance, but is also scalable as a controller takes care of the scheduling of the number of actions, implements data parallelism, manages the fork-join process, and returns the result. Ultimately, the user would not have to manage the burden of individually determining the amount of data to assign to an action or the number of action to create. The proposed system just requires the name of the dataset and the job to complete. The rest is handled internally and results are returned to the user.

Chapter 5

Computational Experiments

This section presents computational experiments designed to evaluate the performance of the proposed module. In section 5.1 we introduce different components of our experiments starting from the system configuration, and explain the considered data instances along with comparison metrics used to assess the implemented test cases. Section 5.2 reports the results and discusses their implications.

5.1 Experiments Design and Data

The proposed module is validated on a CloudLab testbed where OpenWhisk is deployed in a Kubernetes cluster. Prior to computation, we installed software packages such as Docker [63], Kind [60], Helm [30], Kubernetes [31], OpenWhisk CLI (wsk) [9] that are required for the cluster deployment. Kind creates the cluster for running Kubernetes locally; Helm installs the deployment; Kubernetes orchestrates the OpenWhisk system; Docker containers host the execution of users' requested actions. The script used to setup the experiment is provided in Appendix 7.1 and the code is available in the GitHub repository of RIT's High Performance Distributed Systems Lab[1].

DISDEL is developed in Python to ensure that it is lightweight and user-friendly, and additional shell scripts are written to orchestrate the creation and invocation of our actions. A collection of five data instances from the TensorFlow datasets catalog [56], six evaluation metrics and three different test cases were used to facilitate the validation of our solution.

[1] https://github.com/hpdsl/disdel

5.1.1 Test Cases

Three different scenarios are evaluated in our experiments in order to assess diverse run-time environments and validate the efficiency of our proposed solution. The test cases are described as following:

- **DISDEL:** This implementation employs our proposed module for distributed serverless deep learning as described in Section 4.2.
- **Brute Force:** We directly run the algorithms on OpenWhisk and increase either the memory size by 256 MB or the timeout by 10 seconds every time the container fails based on the returned error type. These values are maintained as they avoid restarting the containers many times which ends up increasing the total memory usage and duration.
- **Virtual Machine:** We execute the deep learning algorithms on the host virtual machine to imitate deep learning on a server and for further comparisons.

5.1.2 Data Instances

Five data instances classified into two groups are used. Table 5.1 describes each instance and presents their respective size, number of labels, number of examples and the job requested in our computational experiments. The first set refers to datasets of size measured in KiB and the second set groups datasets measured in MiB.

- **Binary_Alpha_Digits[2]:** This dataset contains 20×16 representations of numbers from 0 to 9 and capital letters from A through Z.
- **CIFAR10 [48]:** From the Canadian Institute For Advanced Research (CIFAR), this dataset is a collection of images that are commonly used to train machine learning and computer vision algorithms. Ten different classes representing airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks are included.

[2] The homepage is https://cs.nyu.edu/~roweis/data/

- **CIFAR100 [38]:** This dataset is similar to CIFAR10, but contains 100 classes with 600 images each.
- **MNIST [40]:** The Modified National Institute of Standards and Technology (MNIST) database is a large database of handwritten digits that is commonly used for training various image processing systems.
- **Fashion_MNIST [66]:** This dataset consists of gray scale 28×28 images labeled from ten different classes.

5.1.3 Evaluation Metrics

We use the following six metrics to measure the performance of our test cases:

1. Number of created containers
2. Total memory usage
3. Total execution duration
4. Total container initialization time (cold start)
5. Number of failed containers
6. Failure rate

Computational results are obtained on a 3-node cluster composed of Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz machines from the CloudLab [23] Wisconsin data center. Each node has 40 CPUs with 32-bit and 64-bit op-modes, offers 196 GB of RAM, and hosts an Ubuntu 18.04 operating system. For the sake of diversity among the results, we ran experiments varying the batch size and number of epochs such that: batch={32, 64, 128}; epochs={5, 10}.

Set	Name	Size	Classes	Number of examples	Jobs requested
1 (KiB)	Binary_Alpha_Digits	519.83	36	1404	train
2 (MiB)	CIFAR100	292.74	100	60000	train
	CIFAR10	294.58	10	60000	train & test
	Fashion_MNIST	65.78	10	70000	train & test
	MNIST	32.06	10	70000	train & test

Table 5.1: Summary of experiments data instances.

5.2 Analysis and Discussion

5.2.1 Numerical Analysis

We provide the results of our computations with a batch size of 64 over five epochs shown in Table 5.2,Table 5.3 and Table 5.4. Among others sets of parameters, we report results with the aforementioned values as they provide the best sample for analysis, especially with "Out Of Memory" errors which are obtained as outcomes of some experiments. For all the tables reporting the computational results, Column 1 states the dataset used during the experiment under consideration. In Table 5.2 and Table 5.4, Columns 2 and 3 respectively report the memory usage during the execution of the deep learning job in each of our test cases; Columns 4 and 5 contain the duration of the execution, while Table 5.2 specifically has Columns 6 and 7 present the time taken by the initialization of a container for the requested action. We have not measured the initialization time for the direct deployment on a virtual machine as this case does not use serverless resources. In Table 5.3, Columns 2 and 3 report the number of containers used to execute jobs requested in cases of deployments using OpenWhisk; Columns 4 and 5 present the number of restarted containers due to out-of-memory or timeout failures; Columns 6 and 7 compute the failure rates in the compared cases.

We first evaluate the performance of DISDEL with respect to a brute-force approach as

Instance	Memory (MB)		Duration (s)		Initialization (s)	
	DISDEL	Brute Force	DISDEL	Brute Force	DISDEL	Brute Force
Binary_Alpha_Digits	1268	1536	42	60	6	6
CIFAR100	8060	**16896**	279	**510**	9	**20**
CIFAR10	7952	**16896**	262	**630**	9	**20**
Fashion_MNIST	4201	7168	178	400	7	12
MNIST	3904	7168	89	210	7	12

Table 5.2: Comparison of experiments results of the proposed solution and a brute-force approach on OpenWhisk. Results obtained with 4 GB as maximum memory are labeled in bold.

summarized in Table 5.2. The over-performance of DISDEL is shown to be consistent over all the data instances, and expresses the efficiency and ellitism of our module at executing training and inference jobs on serverless infrastructure. The considerable difference in the memory usage and the execution duration proves the importance of data parallelism in the efficient resource allocation and deployment of an application to train deep learning models. On the one hand, pre-allocating enough memory ensure lower resource utilization and faster computation. As a consequence under the same setting of 2048 MB as maximum memory limit, we achieved up to 45% and 58% reduction respectively in the memory usage and duration. On the other hand, our approach eliminates the need to restart a container every time its fails, and as reported in Table 5.3 reduces the rate of failed executions to 0%.

Further analyses of the results in Table 5.2 show that the execution of deep learning jobs does not only depend on the size of the dataset, but it is also influenced by the size of the model. Compared to the brute-force solution, the proposed data-parallel approach has successfully trained more models, namely instances such as Binary_Alpha_Digits, Fashion_MNIST, MNIST while maintaining the default maximum memory allocation intact. Indeed, following the estimation and allocation of memory, DISDEL launched the concurrent execution of requested actions, but containers went out of memory during training.

Instance	Used Containers		Restarted Containers		Failure Rate	
	DISDEL	Brute Force	DISDEL	Brute Force	DISDEL	Brute Force
Binary_Alpha_Digits	3	3	0	2	0%	67%
CIFAR100	5	**11**	0	**10**	0%	**91%**
CIFAR10	5	**11**	0	**10**	0%	**91%**
Fashion_MNIST	4	7	0	6	0%	86%
MNIST	4	7	0	6	0%	86%

Table 5.3: Comparison of failure rate of the proposed solution and a brute-force approach on OpenWhisk. Results obtained with 4 GB as maximum memory are labeled in bold.

These failures are due to requirements for more memory as an assignment of 2048 MB was not enough, and allocating more would yield an error: *Unable to create action 'nameOfAction': The request content was malformed: requirement failed: memory 'allocatedMemory' MB exceeds allowed threshold of 2147483648 B*. Consequently, we raised the maximum consumable memory of an action to 4096 MB in order to successfully train models on CIFAR10 and CIFAR100 datasets. Note that fashion_MNIST, an instance with the largest size of 65.78 MB among successful cases, has an input shape of (28, 28, 1). Meanwhile, CIFAR10, an instance with the largest input shape (32, 32, 3) among successful cases, has a size of 308.89 MB. Henceforth, we can conclude that the size of compiled models stood a significant amount of memory, and that data-parallelism did not have enough impact as the entire model is replicated inside each running action.

Table 5.4 reports the results out of the execution of requested jobs on a virtual machine. We observe a percentage decrease compared to our solution, suggesting that a bare-metal deployment consumes less memory and time. However, reported results for the virtual machine do not account for the launch time of the machine as well as the memory usage of installed packages and modules. Meanwhile, each container deployed by DISDEL includes a runtime docker image of 225 MB, logs and checkpoint files which size is specific to

Instance	Memory (MB)		Duration (s)	
	Proposed	Virtual Machine	Proposed	Virtual Machine
Binary_Alpha_Digits	1268	118	42	4
CIFAR100	8060	255	279	28
CIFAR10	7952	527	262	29
Fashion_MNIST	4201	449	178	31
MNIST	3904	408	89	29

Table 5.4: Comparison of experiments results of the proposed solution and a deployment on a virtual machine.

the data and the model, and requires a minimum memory limit of 128 MB. Aware of the considerable time taken to fire-up a virtual machine, and in view of the size of TensorFlow installation package (wheel file of size 320 MB), we could argue that our implementation is more efficient. As a supporting argument, we proceeded to only serve the previously trained model for inference using the Fashion_MNIST and MNIST test data. Without including the size of the runtime docker image and the downloaded dataset in the memory usage of DISDEL, results show a higher performance of our system. For Fashion_MNIST, we yielded a memory usage of 327 MB compared to 345 MB for the virtual machine, whereas the inference with MNIST consumed 267 MB against 326 MB for the virtual machine. Consequently, our approach is also efficient from the system performance perspective for inference jobs.

In summary, the aforementioned results support the benefits of data parallelism at reducing the workload of running actions, and allow them to execute without failure. Subsequently, our solution raises the capacity of serverless infrastructures at training and/or inferring deep learning models. Nevertheless, the model size also influences the memory allocation of serverless actions, and should be taken into consideration for a more comprehensive and accurate estimation.

5.2.2 Sensitivity Analysis

To further investigate the merits of distributed serverless deep learning, we ran a set of experiments aiming at measuring the sensitivity of the memory usage to the batch size and the number of epochs, as well as the influence of a deployment with OpenWhisk on the accuracy and the loss throughout the training phase.

Accuracy and loss are two key features which measure the efficiency of a model. While many algorithms have proved accurate on physical or virtual machines, employing serverless resources to train models with similar algorithms could raise some questions as to what effect does the infrastructure has on the quality of the model. Therefore, to answer that question, an impact analysis is conducted on the MNIST and Fashion_MNIST datasets. Results on Figure 5.1 compare the accuracy on the sub-figures above and the loss on the lower level. On the legend, we denote Openwhisk's curb by "wsk" and that of the virtual machine by "vm". According to the measurements, the serverless infrastructure is as effective as the virtual machine to run deep learning jobs. In certain cases, a slight over-performance of the serverless approach is also noticed through a higher accuracy and lower loss. Consequently, we can conclude that DISDEL does not only improve cost and resource utilization, but also maintains a high accuracy for trained models.

Furthermore, we evaluated the impact of the batch size and the number of epochs on the memory usage, and report computational results on Figure 5.2 and Figure 5.3. For the sake of more variations, we have selected the Binary_Alpha_Digits as target data instance in these experiments. With its reduced size, this dataset facilitates training the model over more than 50 epochs. The trend of generated graphs illustrate a gradual increase in the memory usage with respect to the increment of either the batch size or the number of epochs. Specifically, as shown in Figure 5.2, training jobs consumes more memory as the batch size increases. Similarly in Figure 5.3, increasing the number of epoch shows a rise in the usage. The peak and drop trend of the diagram shows variations in the memory usage for the same jobs at different times, and eventually memory leaks which is a common issue in the training phase of machine learning models.

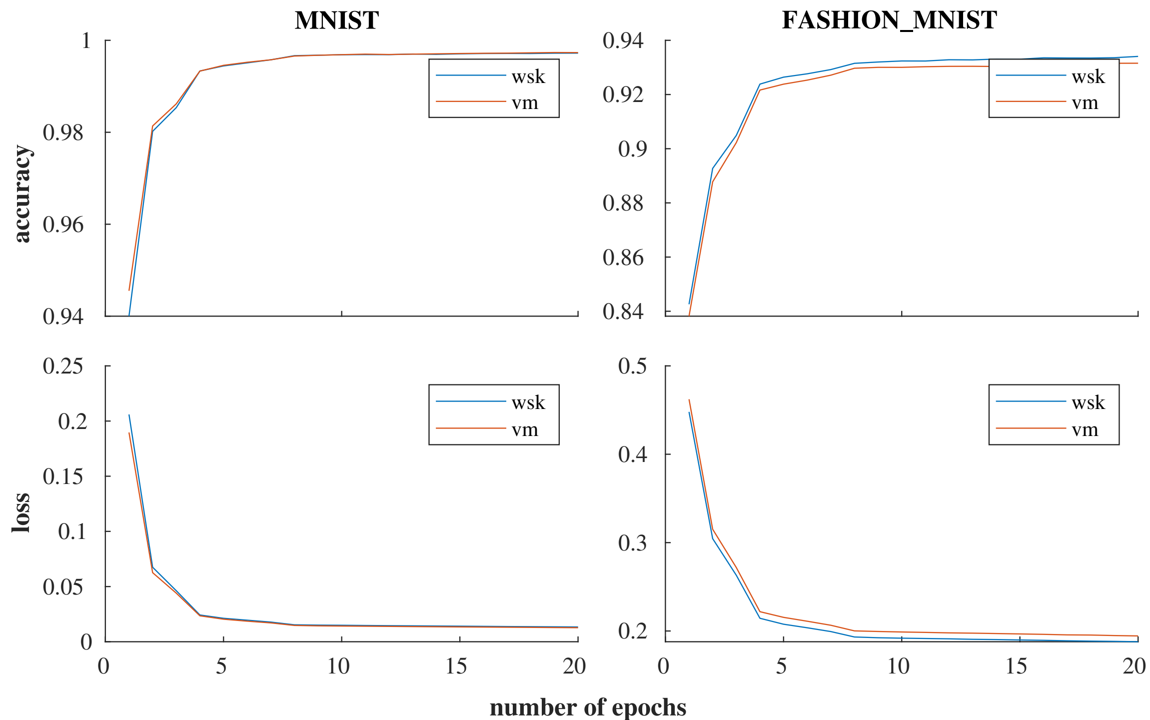


Figure 5.1: Impact of the serverless architecture on the model training accuracy and loss.

5.2.3 Discussion and Implications

In this section, we discuss the results from the computational experiments, and present their implications from both design and economic standpoints. One the one hand, the examination of the diverse diagrams throughout this chapter yields some observations described in the following paragraphs.

First, the design of a supportive system for resource allocation is beneficial to reduce the rate of failing training jobs, and attain a more efficient resource utilization during the entire deep learning workflow. The implementation of a brute-force mechanism may ensure the successful training of some models, but as the scale of the data and the model enlarges, the entire application is exposed to more failures and resource utilization. Standing on this remark, we can infer that our proposed data-parallel strategy offers more scalability to the deployment of deep learning jobs on serverless infrastructures.

Second, the necessity of a fine-grained design to address OpenWhisk's timeout limit

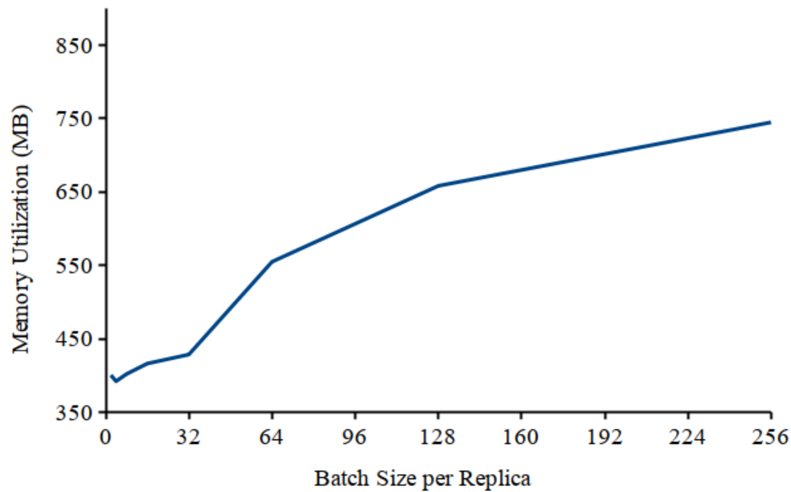


Figure 5.2: Impact of the batch size on the memory usage.

compelled us to deploy multiple actions. Subsequently, the approximation of the most suitable memory allocation resulted in a little but existing amount of unused memory resources. Similarly, the timeout of the proposed approach combines the timeout of each action, and the transmission delay of the parameters to and from the Redis data store. Consequently, there is still a scope of improvement to DISDEL as part of the optimization of the entire system.

Third, we observed that fitting the model to the data is most memory intensive part of training jobs. The increase in the batch size or the number of epoch induces more memory usage, but these factors do not impact the accuracy of training deep learning models on OpenWhisk. In other words, the infrastructure and architecture of serverless computing do not reduce the accuracy of the model provided the memory and execution duration are within permissible ranges.

Combining the analysis of OpenWhisk in Section 4.1 and the results of our computational experiments, we can formulate the following conclusions:

1. The proposed module, DISDEL, successfully reduce the failure rate of training deep learning models on serverless infrastructures down to 0%. Therefore, the efficient management of resource utilization using distributed computing strategies ensures

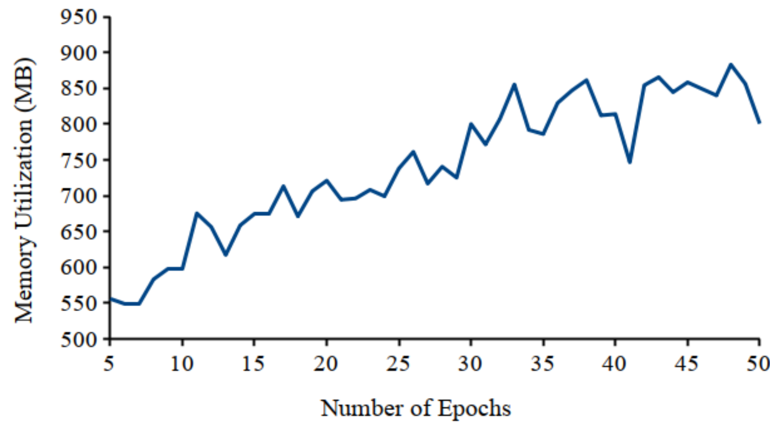


Figure 5.3: Variation of the memory usage with regards to the number of epoch.

the reduction of the failure rate of deep learning's training jobs.

2. Our solution opens new perspectives for distributed serverless deep learning by enabling the execution of training and inference jobs on medium size datasets. Although data parallelism addresses with success issues related to the size of the dataset, our results illustrate a significant challenge with the model size for certain instances. Under circumstances where the trained model is expected to be large, our approach would require additional enhancement to avoid containers running out of memory.

3. There is a significant challenge operating a system level modification to OpenWhisk's core infrastructure in order to dynamically update the memory allocation of running containers. However, deep learning is attainable by adding a module which integrates parallel and distributed computing strategies to alleviate the heavy workload of each cloud actions.

Chapter 6

Conclusions

6.1 Lessons Learned

An application level enhancement allows distributed deep learning developers to deploy their software using serverless architecture with a very high success rate. Formerly representing an ordeal for the deployment of applications, the handling of memory and time-out limits led many users to adopt commercial platforms known for their flexibility, user-friendliness, and the availability of accessories such as database to store large datasets. Our research has focused on an open-source serverless infrastructure, OpenWhisk, and examined possible improvements to the system which would accommodate a dynamic scaling of memory limits and computation timeouts while solving deep learning problems.

Subsequent to the evaluation of OpenWhisk, we observed two challenges to the implementation of a resource partitioner within core components like the Invoker or the Controller. On the one hand, changes to OpenWhisk invoker does not impact the container executing the action because the two entities just interact through HTTP services, and do not run inside the same Kubernetes pod. On the other hand, a deployment of OpenWhisk in a kubernetes Cluster uses docker containers to execute actions. However, executing containers inside Kubernetes pods challenges the update of the container's resources among which the memory limit is of interest in the problem under consideration.

Consequently, we have introduced a fine-grained system design which employs data parallelism to assign jobs to concurrent actions, and averages intermediate outputs in order to generate an ensemble model. The proposed module, DISDEL, uses Redis in-memory

data store to maintain the intermediate state of the training job. From the results of our computational experiments, we can conclude that although distributed serverless deep learning using TensorFlow framework is achievable without any enhancement, yet the implementation is subject to a high failure rate. On the contrast, DISDEL improves the performance by reducing the memory usage by 40% while saving up to 58% of the execution time, but also ensures a failure rate of 0% for requested deep learning jobs.

6.2 Future Work

Expanding on the results, remarks, and failures of this research, there are two future directions which can be categorized as enabling more support and optimizing the current system.

In view of our incapacity at changing the memory limit of a running action because it executes inside a pod, the examination of a standalone OpenWhisk deployment (only using docker containers) could provide better insights on how to perform system level modifications.

Moreover, the optimization of the performance of the current system will be our focus, and solutions like the reduction of the number of cold started containers will be developed. Observing that all the containers used by DISDEL started cold, integrating additional strategies to optimize containers' initialization phase will systematically reduce the total duration of serverless deep learning jobs.

Besides, the improvement of the data input pipeline to accommodate datasets in traditional formats will provide the system with more flexibility. Further, we will improve the performance of DISDEL by combining data and model parallelism to implement a hybrid parallel system in order to train medium to large size models. As well, we intend to diversify compatible deep learning modules to attract developers outside the TensorFlow community.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Akka Documentation. Actors. https://doc.akka.io/docs/akka/2.5/index-actors.html. Accessed: 2020-06-15.

[3] Akka Documentation. Classic fsm. https://doc.akka.io/docs/akka/current/fsm.html. Accessed: 2020-06-15.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[5] Amazon Web Services. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. Accessed: 2020-04-10.

[6] Amazon Web Services. Serverless Computing. https://aws.amazon.com/serverless/. Accessed: 2020-03-20.

[7] Amazon Web Services. What Is AWS Step Functions? https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. Accessed: 2020-01-16.

[8] Amazon Web Services. Serverless Architectures with AWS Lambda: Overview and Best Practices. https://d1.awsstatic.com/whitepapers/

serverless-architectures-with-aws-lambda.pdf, 2017. Accessed: 2020-03-20.

[9] Apache OpenWhisk. https://openwhisk.apache.org/documentation.html. Accessed: 2020-01-16.

[10] The internal flow of processing. https://github.com/apache/openwhisk/blob/master/docs/about.md#the-internal-flow-of-processing. Accessed: 2020-06-15.

[11] Apache OpenWhisk Composer. https://github.com/apache/openwhisk-composer. Accessed: 2020-01-16.

[12] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178, 2017.

[13] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 25–30, 2019.

[14] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, 2019.

[15] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 52(4):1–43, 2019.

[16] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. Using application knowledge to reduce cold starts in faas services. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[17] M. Boniface, B. Nasser, J. Papay, S. C. Phillips, A. Servin, X. Yang, Z. Zlatev, S. V. Gogouvitis, G. Katsaros, K. Konstanteli, G. Kousiouris, A. Menychtas, and D. Kyriazis. Platform-as-a-service architecture for real-time quality of service management in clouds. In *2010 Fifth International Conference on Internet and Web Applications and Services*, pages 155–160, 2010.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at Neural Information Processing Systems*, 2018.

[19] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, 2019.

[20] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-Mei W. Hwu. Trims: Transparent and isolated model sharing for low latency deep learning-inference in function as a service environments. *CoRR*, abs/1811.09732, 2018.

[21] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[24] A Ellis et al. Openfaas: Serverless functions made simple, 2019.

[25] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, 2017.

[26] Amir Gholami, Ariful Azad, Kurt Keutzer, and Aydin Buluç. Integrated model and data parallelism in training neural networks. *CoRR*, abs/1712.04432, 2017.

[27] Google Cloud. Cloud Functions. `https://cloud.google.com/functions`. Accessed: 2020-03-20.

[28] Google Cloud. Knative. https://cloud.google.com/knative. Accessed: 2020-04-10.

[29] Eugene Gorelik. *Cloud computing models*. PhD thesis, Massachusetts Institute of Technology, 2013.

[30] Helm. Using helm. https://helm.sh/docs/intro/using_helm/. Accessed: 2020-03-20.

[31] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.

[32] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[33] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. *CoRR*, abs/1710.08460, 2017.

[34] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[35] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, 2017.

[36] Kavitha Chetana Didugu. Ultimate Guide to Input shape and Model Complexity in Neural Networks. https://towardsdatascience.com/ultimate-guide-to-input-shape-and-model-complexity-in-neural-networks-ae665c728f4b. Accessed: 2020-07-20.

[37] Alireza Keshavarzian, Saeed Sharifian, and Sanaz Seyedin. Modified deep residual network architecture deployed on serverless framework of iot platform based on human activity recognition application. *Future Generation Computer Systems*, 101:14 – 28, 2019.

[38] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, 2009.

[39] Feng Lang, Kudva Prabhakar, Da Silva Dilma, and Hu Jiang. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341, 2018.

[40] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[41] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Primitives for dynamic big model parallelism, 2014.

[42] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *CoRR*, abs/1903.12221, 2019.

[43] Microsoft Azure. What are Durable Functions? `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp`. Accessed: 2020-01-16.

[44] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[45] OpenWhisk system details. `https://github.com/apache/openwhisk/blob/master/docs/reference.md`. Accessed: 2020-04-10.

[46] A. Palade, A. Kazmi, and S. Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642-939X, pages 206–211, 2019.

[47] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge AI. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[48] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do CIFAR-10 classifiers generalize to cifar-10? *CoRR*, abs/1806.00451, 2018.

[49] RedisLabs. Redis. `https://redis.io/`. Accessed: 2020-01-16.

[50] S. Shahzadi, M. Iqbal, Z. U. Qayyum, and T. Dagiuklas. Infrastructure as a service (iaas): A comparative performance analysis of open-source cloud platforms. In *2017 IEEE 22nd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–6, 2017.

[51] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young,

Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018.

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019.

[53] Josef Spillner. Transformation of python applications into function-as-a-service deployments. *CoRR*, abs/1705.08169, 2017.

[54] Wei Sun, Kuo Zhang, Shyh-Kwei Chen, Xin Zhang, and Haiqi Liang. Software as a service: An integration perspective. In *Service-Oriented Computing – ICSOC 2007*, pages 558–569, 2007.

[55] Zhaohao Sun, Huasheng Zou, and Kenneth Strang. Big data analytics as a service for business intelligence. In *Open and Big Data Management and Innovation*, pages 200–211, 2015.

[56] TensorFlow. Datasets. https://www.tensorflow.org/datasets/catalog/overview#all_datasets. Accessed: 2020-06-15.

[57] TensorFlow. Distributed training with Keras. https://www.tensorflow.org/tutorials/distribute/keras. Accessed: 2020-01-16.

[58] TensorFlow. TensorFlow Lite guide. https://www.tensorflow.org/lite/guide. Accessed: 2020-03-20.

[59] TensorFlow. The TFX User Guide. https://www.tensorflow.org/tfx/guide. Accessed: 2020-03-20.

[60] The Kubernetes Authors. Quick start. https://kind.sigs.k8s.io/docs/user/quick-start/. Accessed: 2020-03-20.

[61] Sergio Trilles, Alberto González-Pérez, and Joaquín Huerta. An iot platform based on microservices and serverless paradigms for smart farming purposes. *Sensors*, 20(8):2418, 2020.

[62] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[63] J. Turnbull. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014.

[64] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed tensorflow with MPI. *CoRR*, abs/1603.02339, 2016.

[65] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[66] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[67] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, pages 923–935, 2016.

Chapter 7

Appendix

7.1 Experiments Configuration

7.1.1 Environment Setup

To setup up the environment, follow the following steps.

Installation of Docker

```
echo "CONFIG INFO:: Installation of Docker........."
sudo apt-get install -y apt-transport-https \
            ca-certificates curl gnupg-agent \
            software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/
                        ubuntu $(lsb_release -cs) stable"
sudo apt-get install -y docker-ce docker-ce-cli containerd.io

# Manage docker as non-user root
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

Installation of Golang

```
curl -O https://storage.googleapis.com/golang/go1.13.5.linux-amd64.tar.gz
tar -xvf go1.13.5.linux-amd64.tar.gz
sudo chown -R root:root ./go
sudo mv go /usr/local
echo -e "export GOPATH=\$HOME/go" >> ~/.profile
echo -e "export PATH=\$PATH:/usr/local/go/bin:\$GOPATH/bin" >> ~/.profile
source ~/.profile
```

Installation of Kubernetes and Helm and Kind

```
echo "CONFIG INFO:: Installation of Kubernetes........."
sudo apt-get update && sudo apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
sudo apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" |
sudo tee -a /etc/apt/sources.list.d/kubernetes.list
sudo apt-get install -y kubectl

echo "CONFIG INFO:: Installation of helm.........."
curl -L https://git.io/get_helm.sh | bash

echo "CONFIG INFO:: Installation of kind.........."
GO111MODULE="on" go get sigs.k8s.io/kind@v0.6.1
```

Installation of wsk CLI

```
echo "CONFIG INFO:: Configuring wsk api............"
cd $GOPATH
go get github.com/apache/openwhisk-cli
cd $GOPATH/src/github.com/apache/openwhisk-cli
go get -u github.com/jteeuwen/go-bindata/...
sudo apt install go-bindata
go-bindata -pkg wski18n -o wski18n/i18n_resources.go wski18n/resources
go get -u github.com/kardianos/govendor
sudo apt install govendor
govendor sync
rm -rf vendor/github.com/spf13
go build -o wsk

# Create a folder maned bin and copy binary into it
mkdir bin
mv wsk bin

# Setting the path to the binary with"
echo -e "export PATH=$PATH:~/go/src/github.com/apache/openwhisk-cli/bin" >>
~/.bashrc
source ~/.bashrc
```

Continue to setting the OpenWhisk API host and authentication

```
# Values are as in the cluster deployment yaml file
wsk property set --apihost <whisk.ingress.apiHostName>:<whisk.ingress.apiHostPort>
wsk property set --auth <whisk-auth-key>
```

7.1.2 Additional resources

This section groups the links to the homepage of the different components enabling our research.

- Disdel (Our module):
  https://github.com/hpdsl/disdel
- TensorWhisk Docker Image:
  https://hub.docker.com/repository/docker/kevinassogba/tensorwhisk
- Apache OpenWhisk:
  https://github.com/apache/openwhisk-deploy-kube
- OpenWhisk Python Runtime:
  https://github.com/apache/openwhisk-runtime-python
- OpenWhisk Docker Runtime:
  https://github.com/apache/openwhisk-runtime-docker
- TensorFlow:
  https://github.com/tensorflow/tensorflow
- Redis:
  https://github.com/redis/redis