FlowCon: Elastic Flow Configuration for Containerized Deep Learning Applications

Wenjia Zheng*
Fordham University
Bronx, NY, USA
wzheng33@fordham.edu

Michael Tynes*
Fordham University
Bronx, NY, USA
mtynes@fordham.edu

Henry Gorelick
Fordham University
Bronx, NY, USA
hgorelick@fordham.edu

Ying Mao
Fordham University
Bronx, NY, USA
ymao41@fordham.edu

Long Cheng
University College Dublin
Dublin, Republic of Ireland
long.cheng@ucd.ie

Yantian Hou
Boise State University
Boise, ID, USA
yantianhou@boisestate.edu

ABSTRACT

An increasing number of companies are using data analytics to improve their products, services, and business processes. However, learning knowledge effectively from massive data sets always involves nontrivial computational resources. Most businesses thus choose to migrate their hardware needs to a remote cluster computing service (e.g., AWS) or to an in-house cluster facility which is often run at its resource capacity. In such scenarios, where jobs compete for available resources utilizing resources effectively to achieve high-performance data analytics becomes desirable. Although cluster resource management is a fruitful research area having made many advances (e.g., YARN, Kubernetes), few projects have investigated how further optimizations can be made specifically for training multiple machine learning (ML) / deep learning (DL) models. In this work, we introduce `FlowCon`, a system which is able to monitor loss functions of ML/DL jobs at runtime, and thus to make decisions on resource configuration elastically. We present a detailed design and implementation of `FlowCon`, and conduct intensive experiments over various DL models. Our experimental results show that `FlowCon` can strongly improve DL job completion time and resource utilization efficiency, compared to existing approaches. Specifically, `FlowCon` can reduce the completion time by up to 42.06% for a specific job without sacrificing the overall makespan, in the presence of various DL job workloads.

CCS CONCEPTS

• **Computing methodologies**; • **Applied computing**; • **Computer systems organization**;

KEYWORDS

cloud computing, deep learning, containerized application, resource management, high performance analytics

ACM Reference Format:
Wenjia Zheng, Michael Tynes, Henry Gorelick, Ying Mao, Long Cheng, and Yantian Hou. 2019. FlowCon: Elastic Flow Configuration for Containerized Deep Learning Applications. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337868

1 INTRODUCTION

With the growth of big data from various domains such as the Web, Internet of Things (IoTs) and Edge [20], deep learning (DL) is becoming increasingly attractive to businesses and researchers alike who wish to extract meaningful information from massive data sets. Popular network architectures such as Convolutional Neural Networks (CNNs) [1] and Recurrent Neural Networks (RNNs) [12] have been widely used in various applications due to their ability to uncover hidden correlations, causal relationships, and other more complicated patterns in massive datasetes. For example, Google AdSense [3] utilizes clients' browsing, searching and email history to provide customized and targeted recommendations.

Deep learning methods, and machine learning methods more generally, aim to fit a function $\hat{f}(\mathbf{x}, \boldsymbol{\theta})$ to a dataset. The function can be used as a model to make predictions based on the data or discover patterns in the data. To fit the model, model error is usually defined by a loss function $J(\boldsymbol{\theta})$ and the target is to optimize the model parameters to minimize the loss. Deep learning architectures often have extremely large parameter sets, sometimes running into the hundreds of millions, and thus require extremely large volumes of data to train. Due to the sheer scale of such data, which in the big data age is often measured in terabytes or even petabytes, today's big data practitioners generally collect, store, process, analyze, and model their data in large datacenters or on the cloud. Training a deep learning model on such a large volume of data is not only time-consuming but also extremely resource-intensive. Particularly when multiple models are being optimized on a shared computing infrastructure, there is a large potential for significant contention between with respect to CPU and memory.

This work aims to improve the resource contention problem for large deep learning tasks in a cloud environment, and consequently

*Both authors contributed equally to this research.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337868

to accelerate their training through efficient system-level resource management. Specifically, we focus on *containerized* learning applications and *containerized cloud architectures*, where training tasks are completed within a *container*: a lightweight, service-level virtualized environment in which processes can be run. Containers such as those provided by Docker [2] and Kubernetes [6] provide a sandboxed, portable environment for the execution of arbitrary code on arbitrary machinery. More importantly, the resource usage of a container can be easily manipulated, which makes the technique quite popular in cloud computing. To illustrate the simplicity of this, a programmer can use the command `docker run <image>` to start a Docker container of type `<image>` on a physical machine, and then update its resource usage limits with a similar command.

To date, popular open-source deep learning frameworks and libraries, such as Pytorch [10], TensorFlow [13], and Keras [5], provide their images for deep learning services. While, various approaches have been proposed to optimize the efficiency of resource sharing across containers on cloud infrastructures in general, few of these approaches are designed specified for deep learning. Specifically, they fail to take into account the nature of DL training jobs, which have an objective to reduce the value of the loss function iteratively until convergence to a minimum. This information is extremely valuable for resource allocation because the rate of convergence is not linear with the amount computing resource. Convergence rates generally decrease with time, which implies that the resource efficiency in terms of training gains per unit resource will decrease with time for a given deep learning job. The current cloud platforms and methods do not consider this information, and thus they are liable to waste resources by allocating them to jobs where the gains in loss with respect to time are small or perhaps not appreciable (more details see Section 2).

In this paper, we propose a novel container workflow management scheme, `FlowCon`, which aims to accelerate the overall system performance of multiple learning tasks running on a containerized cloud cluster through real-time resource allocation. As opposed to a static or fixed configuration, `FlowCon` monitors the progress of learning jobs and dynamically configures the resource limits for each of them based on a novel metric we named *growth efficiency*. The main contributions of this paper are summarized as follows:

- We introduce the concept of growth efficiency, a measure of the magnitude of the change in the loss function with per unit of compute resource, and propose `FlowCon` along with a suite of algorithms to monitor the growth efficiency of deep learning jobs.
- `FlowCon` is designed to elastically allocate and/or withdraw the resources to/from each learning job at run time, allowing jobs to converge more quickly without significant scheduling overhead.
- Evaluation of a `FlowCon` implementation in the popular container platform, Docker, with intensive cloud-based experiments using various DL frameworks demonstrate the effectiveness of `FlowCon`. Specifically, compared to the default system resource allocation scheme, `FlowCon` can reduce completion time of individual jobs by up to 42.06% without sacrificing the overall makespan.

The remainder of this paper is organized as follows. In Section 2, we introduce the background with a motivating example of this paper. We present the system architecture of `FlowCon` in Section 3 and the relevant algorithms in Section 4. We carry out extensive evaluation of `FlowCon` in Section 5. We report the related work in Section 6 and conclude this paper in Section 7.

2 BACKGROUND AND MOTIVATION

In this Section, we briefly introduce containerized applications and motivate our work with an example.

2.1 Containerized Applications

Containerization is a platform-independent virtualization method that enables applications to provide services from a sandboxed runtime environment without launching the overhead of a virtual machine. We will use Docker as a representative containerization platform. With Docker, a new container can be initialized on a physical node by sending commands to the local docker daemon.

In a deployment environment, various required dependencies of a deep learning job can be packaged into an image so that the command `docker run -d <DL_job>` can be used to launch a container running the deep learning job `DL_job` in the background. Once a container is initialized, Docker provides the user with a rich set of options to control and interact with the container. For example, we can use `docker exec <command/program_name>` to run a command or program in a run-time container and use `docker commit <container_id>` to create a new image from a container's changes.

In terms of containerized deep learning applications, developers, researchers, and data scientists can use existing libraries and frameworks to develop learning models. There are several major players in this domain, such as TensorFlow [13], Pytorch [10] and Keras [5]. They provide the platforms to that facilitate the ease of implementation of contemporary ML/DL models like LSTM-CNN [31], LSTM-RNN [32], and DCGAN [30]. The community of developers behind these libraries and frameworks have devoted considerable effort to build their own docker container images for fast distribution and easy management of ML/DL jobs.

2.2 Motivation of `FlowCon`

In order to deploy applications into a production environment, it is difficult to achieve resilience and scalability using only a single compute node. Generally, a cluster (cloud) is used to provide the infrastructure for running a large set of containers at scale. Many toolkits have been designed for container orchestration in cluster environments, such as the Docker Swarm and Kubernetes.

In current containerized cloud systems, running containers compete for resources freely and the system maintains fairness among all of them. Alternatively, users can set an upper limit to each of the containers when initializing them. However, these mechanisms are not optimal for deep learning tasks. There are two main reasons. (1) *Most models don't need to be perfect in a distant future, they just have to be good in the near future.* Suppose we have a set of deep learning tasks running on containers within a cluster. In some settings such as real-time data analytics, a model would be frequently requested by applications (e.g., prediction) even *before* convergence is reached.

In this case, bringing the model to an acceptable (rather than perfect) level of accuracy is the most important. (2) *More commonly, some learning tasks converge faster than others, and their models can reach an acceptable state with fewer iterations (i.e. less time).* If we want to bring all models to a *usable* state while minimizing wait time, simply maintaining the fairness of all tasks will results in a resource waste. This is because jobs that are already in an acceptable state will continue to utilize as much resource as those with much optimization left to do, even though the nearly-converged jobs only make small gains in optimizing their loss function per unit compute resource.

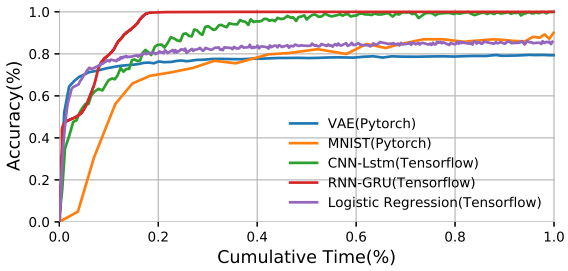

Figure 1: Training progress of five models

As a motivating example, Figure 1 shows the training processes of five different models. There, each model runs inside a container on the same physical node. It can be seen that the training job with RNN-GRU model on Tensorflow reaches 90.0% accuracy at 14.5% of the cumulative time. When it completes training, the accuracy increases to 93.2%. This observation indicates the first 14.5% of the total time, the model achieves 96.8% of its maximum. Later, it takes 85.5% of the total time for another 3.2% of the accuracy. Suppose there are many other learning tasks running in parallel on the same computing node and seeking computational resources. It is reasonable to shift parts of the computational resource occupied by RNN-GRU training to the other learning tasks. In this work, we propose FlowCon to accomplish this goal.

3 THE FLOWCON SYSTEM

In this section, we present the design of FlowCon in detail, including its architecture, system modules, and the optimization problem it solves.

3.1 FlowCon Architecture

In a typical cluster of containers, e.g., Docker Swarm and CNCF Kubernetes, there are multiple managers and workers in the system. Managers accept specifications from the user and are responsible for reconciling the desired state with the actual cluster state, and workers are responsible for running jobs. Figure 2 presents the system architecture of FlowCon. The main components of FlowCon are in the box that represents a worker.

Although mangers have the global view of the workers in the system, FlowCon runs on the worker side to prevent overwhelming the manager, who is responsible for collecting the status information from all workers and assigning jobs to them. In our FlowCon system, mangers only interact with the container pools on the workers, which store the information of all running containers. With this design, the overhead of running the FlowCon is distributed over the whole cluster.

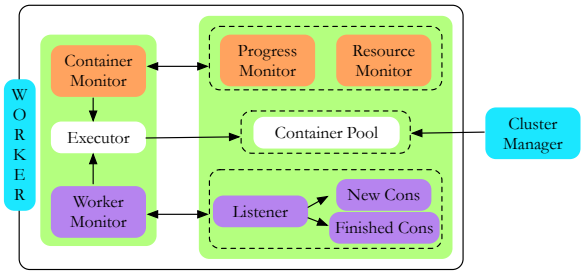

Figure 2: FlowCon System Architecture

3.2 FlowCon Modules

As demonstrated in Figure 2, FlowCon consists of three modules, a *Container Monitor*, a *Worker Monitor* and an *Executor*. Each module runs independently and exchanges information about jobs inside the containers as well as the worker status. Their functionality is detailed below.

3.2.1 Container Monitor. The FlowCon focuses on the containers that provide various machine learning services. A container monitor in FlowCon keeps track of the ML/DL jobs inside each container and collects the progress each of the job in terms of different evaluation functions that are defined by the jobs themselves. Besides that, it collects the resource usage of each container that is running in the pool. At each container level, it records the consumption of four resources: CPU, memory, block I/O, and network I/O.

3.2.2 Worker Monitor. A worker monitor measures the container pool on the worker. There are two listeners, one called New Cons and the other one called Finished Cons. Unlike the container monitor, who focuses on the jobs inside a particular container, these listeners target the status of container pools. The New Cons listener tracks the incoming containers and assigns the appropriate resources to them. The Finished Cons listener monitors the containers with finished jobs and releases their resources to the system.

3.2.3 Executor. The Executor is a key module that collects and analyzes the evaluation functions and resource usage data on the worker. Based on the initial interval, it calculates the required parameters by using the data from Container Monitor and execute the algorithm (described in Section 4) to update the resource configuration for each container. Upon receiving a report from one of the listeners, the Executor will interrupt the current interval and start running the algorithm to update the resource assignment based on the new state of the container pool.

3.3 System Optimization Problem

FlowCon aims to improve resource efficiency, which is generally assumed to be satisfactory in current cloud systems. However, when considering a system with various deep learning applications, the term “in use” fails to accurately reflect efficiency, as illustrated in Figure 1.

Based on the characteristics of deep learning applications, we introduce a new definition of efficiency based on an application's evaluation function. Given a system with a set of running containers, $\{c_{id}\}$, each container uses its own evaluation function to assess its type of machine learning model (e.g., loss reduction and inception score) $E_{c_{id}}(t)$. For each model, based on its $E(t)$, we define the progress score for the container c_{id} to be Eq. 1, where $t_i - t_{i-1}$ is the measurement interval. The value of $P_{c_{id}}(t_i)$ is the per-second progress within the interval.

$$P_{c_{id}}(t_i) = \frac{|E_{c_{id}}(t_i) - E_{c_{id}}(t_{i-1})|}{t_i - t_{i-1}} \tag{1}$$

Here, $P_{c_{id}}(t_i)$ reflects the progress over a given time interval, but it does not account for the resources used towards that progress. Therefore, we propose the *growth efficiency* for each container c_{id} with an active deep learning job. Eq. 2 presents the growth efficiency with respect to different types of resources (e.g., CPU, memory, network I/O and block I/O), denoted by r_i, and $R_{c_{id},r_i}(t_i)$ is a function that returns the average resource usage of c_{id} within the interval $t_i - t_{i-1}$ among each r_i.

$$G_{c_{id},r_i}(t_i) = \frac{P_{c_{id}}(t_i)}{R_{c_{id},r_i}(t_i)} \tag{2}$$

`FlowCon` aims to maximize the sum of growth efficiency for the whole system in each interval, where each learning model has its own evaluation function and can be calculated in real-time. Assume that there are n containers, each runs one job in the system, and $R_{i_{max}}$ denotes the overall resource capacity for r_i. Then, our performance optimization problem can be formalized as $\mathcal{P}$ below, where G_{c_{id},r_i} can be computed from measurements of $E_{c_{id}}(t_i)$ and $R_{c_{id},r_i}(t_i)$.

$$\mathcal{P}: \ \text{Max} \sum_{i}^{n} G_{c_{id},r_i} \tag{3}$$

$$\textbf{s.t.} \ \sum_{i}^{n} r_i \leqslant R_{imax}$$

4 SOLUTION OF FLOWCON

In this section, we present the design and elastic container configuration algorithms in `FlowCon`, which can adjust resource assignment for containers at run time.

4.1 Resource Configuration for Containers

In a traditional cluster of virtual machines (VMs), each VM is assigned with a fixed amount of resources (e.g., CPU cores and memory), which are fixed when the guest operating system is installed. While dedicated VM's for each job enables better isolation, a cluster of VMs fails to efficiently utilize resources for deep learning jobs given their characteristics we presented in Section 2.

In a cluster of containers, system administrators have the option to create, configure, and reconfigure containers in real time. If the containers are started without a specific resource limit, they will compete for resources at runtime just like processes in an operating system. However, the resource plan can be updated at any time after the initialization. For example, the command `docker update <options> container_id` can reset the resource limit as desired. The sample options include `--cpus` for the number of cores, `--cpu-rt-runtime` for CPU real-time runtime in microseconds, `--memory` for memory usage in MB, `--blkio-weight` for a relative weight of block I/O and etc. Finally, values of the limit that set by the `docker update` commands are soft limits, which means that the even if the container cannot maximize its own resource, the unused option will be utilized by others.

Algorithm 1 Dynamic Resource Mgt. for Container c_{id} on Worker W_i

1: Initialization: $c_{id} \in \{c_1, c_2, \ldots, c_n\}$, W_i, Time t, Watching List WL, Completing List CL, New List NL, α and $itval$.
2: **for** $c_{id} \in W_i$ **do**
3: Calculate $G_{W_i,c_{id}}(t)$
4: **if** $G_{W_i,c_{id}}(t) < \alpha$ & $c_{id} \in NL$ **then**
5: NL.remove(c_{id})
6: WL.insert(c_{id})
7: **else if** $G_{W_i,c_{id}}(t) < \alpha$ & $c_{id} \in WL$ **then**
8: WL.remove(c_{id})
9: CL.insert(c_{id})
10: **else if** $G_{W_i,c_{id}}(t) \geq \alpha$ **then**
11: NL.insert(c_{id})
12: WL.remove(c_{id})
13: CL.remove(c_{id})
14: **if** $\forall c_{id} \in W_i, c_{id} \in CL$ **then**
15: **for** $c_{id} \in W_i, r_i \in R$ **do**
16: $L_{c_{id},r_i} = 1$
17: $itval = itval \times 2$
18: **else**
19: **for** $c_{id} \in W_i, r_i \in R$ **do**
20: **if** $c_{id} \in CL$ **then**
21: $L_{c_{id},i} = \frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$
22: $L_{c_{id},i} = Max\{L_{c_{id},i}, \frac{1}{\beta \times |c_{id}|}\}$
23: **else if** $c_{id} \in WL$ **then**
24: $L_{c_{id},r_i} = L_{c_{id},i}$
25: **else**
26: $L_{c_{id},i} = \frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$

4.2 Resource Assignment in FlowCon

In a cluster of containers, the manager accepts the commands from users and selects a worker to host the containers, and containers would compete for resources such as memory and CPU when they are running in the same worker. By default, each container is assigned the same priority resulting in uniform resource distribution among all containers in the worker. This sharing mechanism has acceptable performance. However, as we have discussed, it fails to consider the characteristics of deep learning applications. In comparison, `FlowCon` utilizes a growth-efficiency based method as presented in Algorithm 1, to update resource assignments of each active container in a dynamical way.

As shown in Line 1 of Algorithm 1, each W_i first receives the following parameters from its manager: the time t, the threshold α and the algorithm interval $itval$. Moreover, it initializes three lists as below to categorize each container:

- New List (NL): Young and quickly growing

- Watching List (WL): Near convergence
- Completing List (CL): Converging and growing slowly

Based on the threshold and the growth efficiency that calculated by the container monitor, the algorithm places each active container into the proper list (Lines 2 - 13). If all containers are in the CL, then each container's resource limit is set to 1 allowing them to compete freely for resources (Lines 14 - 16). While FlowCon is permitting free competition, it is no longer necessary for the system to run the algorithm at the initial interval. Instead, FlowCon utilizes an exponential back-off scheme to double the $itval$ in order to reduce the overhead of running the algorithm (Line 17). Once the growth efficiency is less than the preset threshold, FlowCon applies the following rules:

- Each container in the CL has its resource limit set based on its growth during the time interval $\frac{G_{W_i,c_{id}}}{\sum_{c_{id}} G_{W_i,c_{id}}}$ (Lines 18 - 21)
  - If growth is exceedingly small, which is common after convergence, the resource limit is set to a lower bound to prevent abnormal behavior caused by limited resources (Line 22).
- The resource limits of containers in the WL remain unchanged (Line 24).
- Allocate more resources to containers in the NL (Line 26).

4.3 Listeners in FlowCon

The container monitor provides information that allows Algorithm 1 to dynamically allocate resources based on the growth-efficiency in each container and to reduce the scheduling overhead with an exponential back-off scheme. However, there is latency between the time that a worker's state changes (e.g., a new container is initiated) and the point that it can reallocate resources. To improve this issue, FlowCon deploys lightweight background-listeners to track the container states in real-time.

With the same set of parameters, Algorithm 2 presents the workflow of listeners on W_i. First, it initializes the CL, WL, NL and $itval$, and it uses i to record the number of iteration of the listener (Line 1). When the i^{th} iteration is running, it uses the function $T(i)$ to fetch the total number of container on the W_i (Line 2). In all runs after the first run, the listener calculates the difference c, between the most recent two iterations (Lines 3 - 4). If $c > 0$, it means that there are c new containers now active in the system, so the listener will stop and the algorithm finds out the c_{id} of the new containers and add them to the NL (Lines 5 - 7). In the meantime, it resets the $itval$ to the original value in order to break the exponential back-off scheme, and then starts to run Algorithm 1 to update the resource allocation as well as increases the iteration number i. (Lines 8 - 9). The case when $c < 0$ indicates that some containers have completed their jobs. The algorithm will then find the relevant containers by their c_{id}, remove them from their associated category (NL, CL or WL) and release their resources (Lines 10 - 15). Finally, we reset the $itval$, start running Algorithm 1 and increment the iteration number i.

Algorithm 2 Listener on Worker W_i

```
1:  Parameter Initialization: CL, WL, NL, itval, i = 0
2:  T(i) = total number of container at iteration i
3:  if i ≠ 0 then
4:      c = T(i) − T(i − 1)
5:  if c > 0 then
6:      for c_id ∈ W_i & c_id ∉ CL & ∉ WL & ∉ NL do
7:          NL.insert(c_id)
8:      itval = initial_value
9:      Run Algorithm 1 and i + +
10: else if c < 0 then
11:     for c_id ∈ CL | ∈ WL | ∈ NL and c_id ∉ W_i do
12:         NL.remove(c_id)
13:         WL.remove(c_id)
14:         CL.remove(c_id)
15:         Release_resource c_id
16:     itval = initial_value
17:     Run Algorithm 1 and i + +
```

5 EVALUATION

In this section, we evaluate the performance of FlowCon through a set of cloud-executed experiments.

5.1 Experimental Framework

FlowCon uses Docker Community Edition (CE) 18.09 and is implemented as a middleware between worker and manager. It receives tasks from the manager, and then directs the given tasks to the worker for execution.

The testbed is built on the NSF Cloudlab [9], which is hosted by the Downtown Data Center - University of Utah. Specifically, the test-bed uses the R320 physical node, which contains a Xeon E5-2450 processor and 16GB Memory. To ensure a comprehensive evaluation, we test FlowCon with various deep learning models using both the Pytorch and Tensorflow platforms. Table 1 lists the models used in the experiments.

Table 1: Tested Deep Learning Models

Model [11][14]	Eval. Function	Plat.
Variational Autoencoders (VAE) [15]	Reconstruction Loss	P/T
Modified-NIST (MNIST) [8]	Cross Entropy	P/T
Long Short-Term Memory (CFC) [7]	Softmax	T
Long Short-Term Memory (CRF) [39]	Squared Loss	P
Bidirectional-RNN [16]	Softmax	T
Gated Recurrent Unit (GRU) [4]	Quadratic Loss	T

5.2 Experiment Setup and Evaluation Metrics

There are two key parameters in FlowCon: (1) α, the threshold for classifying jobs into NL, WL and CL; and (2) $itval$, the interval for running the Algorithm 1. We evaluate the performance of FlowCon with different parameter configurations and compare it with the original Docker system (denoted as **NA** in this section) within the following three scenarios:

- Fixed scheduling: the time to launch a job is controlled by the administrator.
- Random scheduling: the launch times are randomized to simulate random submissions of jobs by users in a real cluster.

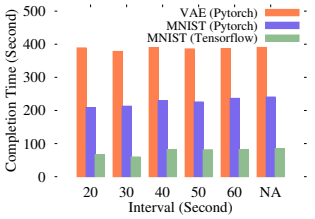

Figure 3: α = 5% and different values of intervals

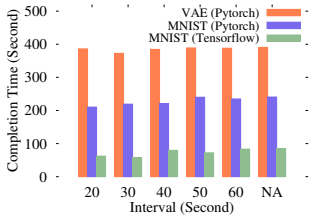

Figure 4: α = 10% and different values of intervals

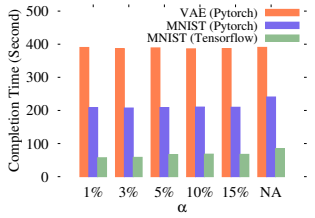

Figure 5: $itval$ = 20 and different values of α

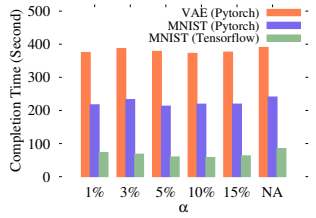

Figure 6: $itval$ = 30 and different values of α

- Scalability: we evaluate FlowCon with an increased number of learning jobs.

The following three metrics are considered in our experiments.

- Overall makespan: the total length of the schedule for all the jobs in the system.
- Individual job completion time: the completion time of each individual jobs in the system.
- CPU usage: all of our tested deep learning models are computation intensive jobs, we focus on analyzing the CPU usage for better understanding FlowCon.

As described in Section 3, all the components of FlowCon and the relevant algorithms reside in worker node. Therefore, in our experiments, we focus on the performance of each individual worker in the system. It should be highlighted here again that the objective of FlowCon is to reduce the individual job's completion time and, at the meanwhile, avoid sacrificing the makespan.

5.3 Fixed Scheduling Results

We fix the schedule of three jobs that VAE on Pytorch starts at 0s, MNIST on Pytorch begins at 40s, and MNIST on Tensorflow launches at 80s. We test our system with different input parameters to understand its performance in various settings.

Makespan: Figure 3 and Figure 4 present the results with different $itval$ in the case of α is 5% and 10% respectively. It can be observed that different $itval$ values have a small effect on the makespan (dominated by VAE), and FlowCon improves makespan by 1% to 5% compared to NA. This is particularly evident when α = 5%, as the makespans are 386.1s, 372.4s, 384.8s, 389.0s, 388.1s and 394.0s respectively. The reason lies in the fact that FlowCon moves some jobs to have their resource limits constrained, thus slowing them down, the jobs which were allocated more resources finished more quickly, thus reducing the overlap between jobs.

Table 2: Completion Time Reduction of MNIST (Tensorflow)

α, $itval$ (Fig. 4)	Reduction	α, $itval$ (Fig. 5)	Reduction
10%, 20	26.2%	1%, 20	32.1%
10%, 30	32.4%	3%, 20	31.0%
10%, 40	14.3%	5%, 20	21.4%
10%, 50	15.3%	10%, 20	19.0%
10%, 60	3.1%	15%, 20	19.8%

In Figure 3, when comparing α = 5%, $itval$ = 20 with NA, the overlap of the two jobs, VAE (Pytorch) and MNIST (Tensorflow), is 213.2s and 240.5s; while the overlap of three jobs is 59.4s and 84.7s. FlowCon decreases the overlap of three jobs by 29.87%. Due to the decrease of the overlap, the completion time of VAE (Pytorch) (the same value as overall makespan in this experiment) will not increase even if we reallocate its resources to another job at runtime. Figure 4 features a key similarity to Figure 3. Using a fixed $itval$ and a varied α, as depicted in Figure 5 and Figure 6, produces similar results as well. Specifically, in Figure 5 and Figure 6, when $itval$ = 20 and $itval$ = 30, with different values of α, FlowCon obviously improves the makespan by 1% to 4% across all settings.

Individual: When the completion time of a specific job is investigated, we find a significant improvement in FlowCon. For example, in Figure 3, when α = 5% and $itval$ = 30, FlowCon reduces the completion time of MNIST (Tensorflow) by 31.9%, from 84.7s to 57.7s. To show more details about how individual learning jobs can benefit from FlowCon, we extract the results of MNIST (Tensorflow) from Figure 4 and Figure 5, and present the reduction of its completion time in Table 2 by comparing FlowCon with NA. We can see that FlowCon performs better than NA in all the parameter settings. When α = 10% with $itval$ = 60, the performance improvement is the smallest one, only 3.1%. This is because that the value of $itval$ is large, and the algorithms need more time to adjust the resource plan for jobs with a large interval. When we fix the value of $itval$ to 20 and vary the value of α, it can be seen that the time reduction generally decreases with the increase of α. The explanation for this result is that jobs stay longer in NL for α = 1%, causing the algorithm to make updates more frequently. For the case α = 15%, jobs stay longer in CL, in which the limits are set to 1, and running tasks will compete for resources freely.

CPU usage: Figure 7 and Figure 8 illustrate the detailed CPU usage of FlowCon (α = 5% with $itval$ = 20) and NA in the presence of the three jobs, respectively. The results in Figure 8 verify that the system equally distributes CPU resources among active jobs when without any configuration (NA). For example, from 40s to 80s and 180s to 280s, the CPU usages of VAE (Pytorch) and MNIST (Pytorch) are approximately equivalent. In comparison, Figure 7 shows both that FlowCon can dynamically set the upper resource limit for each job (actually the resource usage also reflects each job's growth-efficiency). Specifically, when MNIST (Pytorch) is launched at time 40s, FlowCon takes two actions: (1) sets VAE's (Pytorch) resource limit to 0.25 since it is growing slowly, and (2) sets MNIST's (Pytorch) resource limit to 1, allowing for the maximum resource. In this case, VAE (Pytorch) will receive 25% while MNIST (Pytorch) will use 75% of the total resources.

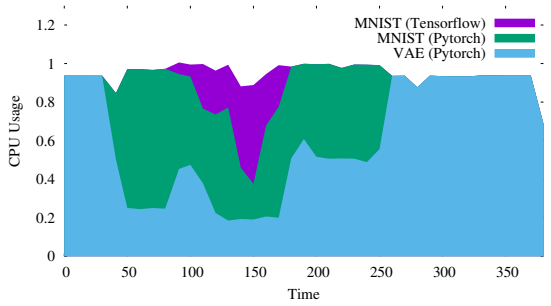

Figure 7: CPU usage of `FlowCon` (α = 5%, $itval$ = 20, 3 jobs)

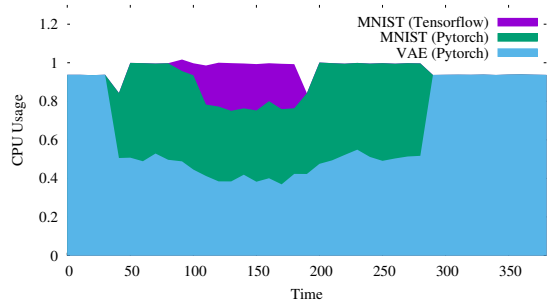

Figure 8: CPU usage of *NA* (3 jobs)

5.4 Random Scheduling Results

For the random scheduling case, we have used five different deep learning models, LSTM-CFC, VAE, VAET, MNIST and GRU, in our experiments. We randomly select a starting time point from 0s - 200s to submit a training job, and the responsible jobs are marked as 1, 2, 3, 4 and 5 respectively in the following results.

Makespan: Figure 9 shows the results of system makespan. Similar to the fixed scheduling case, the results here demonstrate, once again, that `FlowCon` improves the overall makespan, by 1% - 5%. Given the same resource availabilities, `FlowCon` achieves the reduction of makespan by reducing the overlap between jobs.

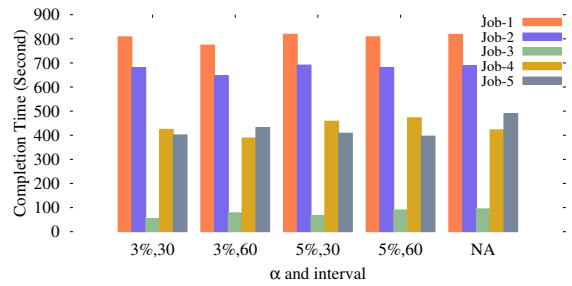

Figure 9: Five different jobs with random submission

Individual: Considering the complete time for each individual job in Figure 9, we can observe that `FlowCon` reduces the completion time for 4 jobs, 5 jobs, 4 jobs, and 4 jobs out of 5 learning jobs for the case with α = 3%, $itval$ = 30; α = 3%, $itval$ = 60; α = 5%, $itval$ = 30; and α = 5%, $itval$ = 60, respectively. The biggest loss happens at the fourth job (denoted as Job-4 and others are similar) with α = 5% and $itval$ = 60. There, Job-4 completes in 472.4s, which is 11.80% slower than *NA*, the completion time of which is 422.5s. The reason is that: although resource allocation to Job-4 greatly decreases when Job-5 begins, the interval of $itval$ = 60 prevents `FlowCon` immediately reallocating resources from Job-4 to Job-5. However, we can see that `FlowCon` obviously reduces the completion time of Job-5, by 19.00%, from 489.4s to 396.4s. The largest performance improvement occurs with the setting α = 3% and $itval$ = 30. In this case, Job-3 completes 42.06% faster. Additionally, the makespan of Job-4 increases 2.1s (424.6s vs. 422.5s), and Job-5 completes 17.92% faster (401.6s vs. 489.4s). Therefore, a smaller value of $itval$ will allow `FlowCon` to reassign the resources more quickly.

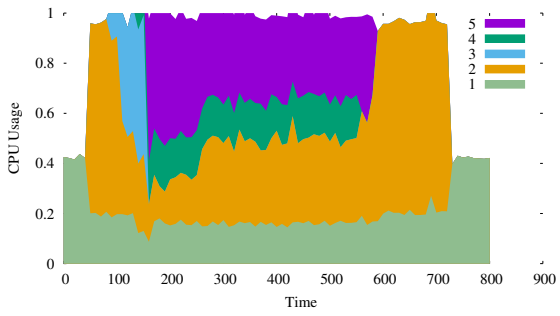

Figure 10: CPU usage of `FlowCon` (α = 3%, $itval$ = 30, 5 jobs)

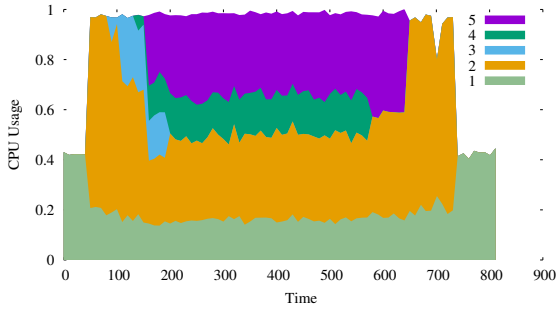

Figure 11: CPU Usage of *NA* (5 jobs)

CPU usage: The random schedule with a larger number of jobs produces more challenges for resource assignments. Figure 10 and Figure 11 present the CPU usage of `FlowCon` with α = 3% and $itval$ = 30 and *NA* in a system of 5 randomly submitted jobs. Unlike Figure 8, the resource usage illustrated in Figure 11 is not equally distributed. For example, from 50s to 80s and from 650s to 730s, the first and the second job are active and use 19% and 79% of the resources, respectively. The reason is that: although the first job (LSTM-CFC) is running alone, it does not maximize the CPU usage (e.g., from 0s to 50s as in Figure 11). Generally, when a container cannot maximize its own resource limit, a portion of the resources may be wasted depending on other jobs in the

system. FlowCon addresses this issue with two techniques: (1) sets a soft resource limit to containers. If a container cannot reach its upper limit, then the resources can be used by others; and (2) although the limits of active containers are correlated and based on the growth-efficiency, the sum of all limits can be greater than 1, since containers in CL employ a lower bound: $\frac{1}{\beta \times |c_{id}|}$ (Line 22 in Algorithm 1).

Remark: Based on the experimental results and analysis in Section 5.3 and 5.4, we can conclude that FlowCon can improve both the system makespan and the completion time of individual learning jobs with different parameter settings, compared to the original Docker system. In the meantime, the values of α and $itval$ can affect the degree of improvement. Since $itval$ indicates the frequency at which Algorithm 1 runs, it is proportional to the overhead including (1) the algorithm resource usage and (2) the delay for reducing the resources of active jobs. Consequentially, as the frequency of running Algorithm 1 decreases, the room for elastic configuration for the running containers will be reduced. Furthermore, the value of α directs how FlowCon categorizes containers in each iteration of the algorithm. Therefore, the best α setting depends on the number of active containers in the system, the machine (deep) learning model in each container and the corresponding datasets.

5.5 Scalability of FlowCon

In this subsection, we conduct experiments to evaluate the performance of FlowCon at a larger scale.

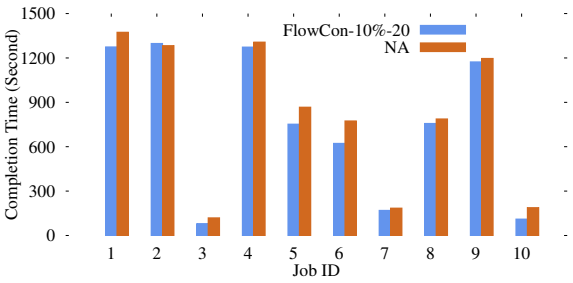

Figure 12: Ten jobs with random submission

5.5.1 10-Job Experiments. Figure 12 shows the completion time comparisons of FlowCon with α = 10%, $itval$ = 20 and NA with 10 jobs that are randomly submitted from 0s - 200s. Under such settings, the makespans are 1350.7s and 1384.9s for FlowCon and NA, respectively. This follows the same trend demonstrated in previous experiments where FlowCon achieves a small improvement in makespan. Considering jobs individually, FlowCon reduces the completion time of 9 out of 10 jobs. Job-2's completion time increases by 15.4s (1.1%) since the interval between Job-2 and Job-3 is very small (13s), and this quickly leads to resource reduction. However, for the other 9 jobs in the experiment, FlowCon produces makespan reductions from 1.8% to 41.2%, with the largest improvement occurring in Job-10: from 188.3s to 110.8s.

Since the FlowCon updates the container's configuration based on the associated value of growth efficiency, we take a deeper look at

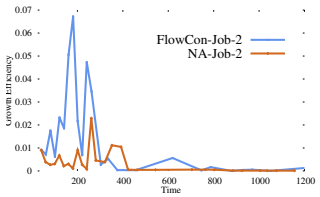

Figure 13: Growth Efficiency Comparison of Job-2

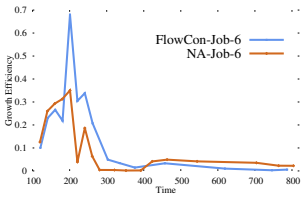

Figure 14: Growth Efficiency Comparison of Job-6

the data from the 10-job experiment. We pick up two representative jobs, Job-2 and Job-6, where Job-6 wins and Job-2 losses a bit in terms of completion time. Figure 13 and Figure 14 illustrate the growth efficiency over the time of Job-2 and Job-6 respectively, in both FlowCon and NA. As we can see from Figure 13, FlowCon gains a lot in growth efficiency at the very beginning. The reason lies in the fact that in FlowCon, Job-2 does not need to compete for resources freely due to an upper limit of resources that applies to every job. The more resources allocate to it, the faster it grows. Even when Job-3 joins the system, resources for Job-2 will not reduce too much since it is still in the WL. In comparison, in NA, every job has the same priority and they compete for the resource whenever it becomes available. When Job-2 converges in FlowCon, more resources will be moved to newer jobs due to a smaller value of the limit, which results in a loss when compares to NA at the time point 320s. We find a different trend in Figure 14. There, at the first 2 iterations, Job-6, in FlowCon, records sightly lower values of growth efficiency. This is because FlowCon needs more time to update the configuration when there are 5 active jobs in the system. It should be noted that the time for resource usages is shorter than the completion time in Figure 12 for a specific container. The difference is caused by our calculation methods. We compute completion time whenever the container is marked as exited and we record the resource usage whenever the Algorithm 1 is called, where the interval could become larger due to the exponential back-off in Algorithm 2.

Figure 15 and Figure 16 present the CPU usage of FlowCon and NA in the same experiment. From Figure 16, we can clearly see the jitter with NA. The jitter is a result of uncontrolled resource competition: whenever there is an idle slot, the system will allocate resources to the first job in the queue. FlowCon also produces jitter, however, the resource usage for each container is much smoother by comparison. This is due to the fact that FlowCon employs a soft, upper resource limit to the containers, and therefore the room for free competition is reduced. The majority of jitter in FlowCon occurs in the interval between 0s to 200s. In this interval, jobs are submitted to the system randomly. After one container joins the system, resource assignment for each container will be updated to reflect this change in the system's status.

5.5.2 15-Job Experiments. We further increase the number of jobs to 15. Again, jobs are randomly submitted to the system during the interval 0s to 200s. As the number of concurrent jobs increases, so does the degree of competition for resources. The results are presented in Figure 17. There, we find the same trend as previous ones:

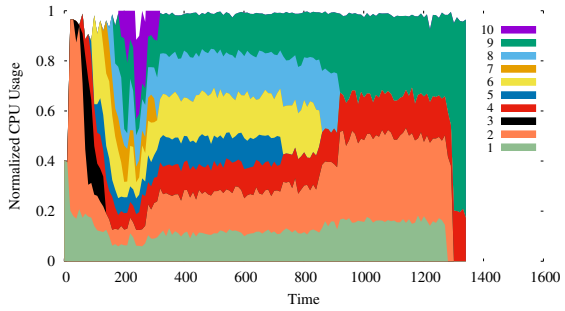

Figure 15: CPU usage of FlowCon (α = 10%, $itval$ = 20, 10 jobs)

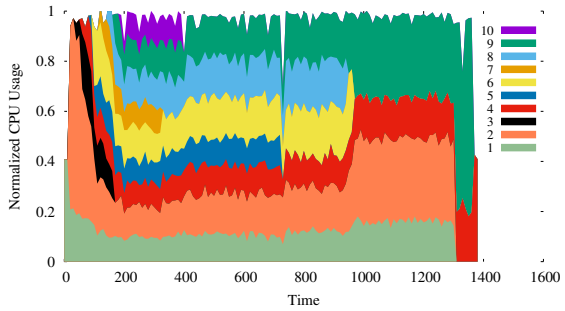

Figure 16: CPU usage of *NA* (10 jobs)

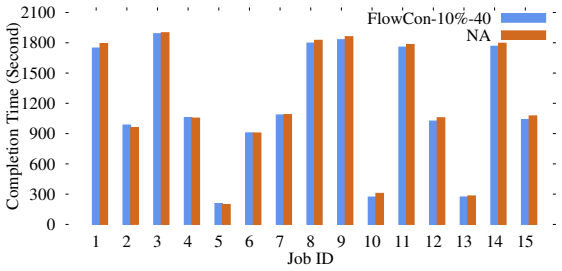

Figure 17: Fifteen jobs with random submission

FlowCon reduces makespan from 1980.1s to 1950.9s (1.5%). Comparing with Figure 9 and Figure 12 highlights the makespan reduction. The more competition in the system, the greater the challenge for elastic configuration at runtime. Furthermore, when considering individual jobs in Figure 17, FlowCon reduces the completion time for 11 jobs out of the 15 jobs. For those 4 jobs, completion time (in seconds) increases: (1) Job-2, from 959.4 to 985.1; (2) Job-4: from 1059.6 to 1053.3; (3) Job-5: from 196.2 to 207.5; and (4) Job-6, from 906.4 to 907.9. However, we can see these increments are quite small, e.g., Job-5's completion time increases the most, only by 5.7%. In comparison, in the other 11 jobs, the degree of reduction ranges from 1.2% to 11.9% and the largest degree of reduction occurs in Job-10, from 308.1s to 271.4s.

6 RELATED WORK

Learning from massive datasets to drive businesses is drastically restructuring our economy. Thus, industry players who wish to thrive in this data-driven environment are compelled to leverage machine learning-based tools to stay ahead of the competition and provide superior experiences to their clients. Efficient platforms for building such tools are thus of central importance to individuals who wish to build these models quickly and at low-cost. FlowCon provides such a paradigm to build DL models in shared computing centers (or clouds), and our experiments have shown that it performs more efficient than current frameworks.

Because of the massive utility of deep learning, developing new models is an extremely active research area with numerous new approaches being proposed continually [25, 33]. However, *very few* of the models are developed explicitly to be efficient, rather most attention is given to their accuracy and quality. Some recent advances include Xception [18], which has been presented to replace Inception modules using depthwise separable convolution and has been demonstrated to have faster convergence on the well-known JFT dataset. Deep convolutional generative adversarial networks (DCGANs) [30] present another cutting-edge method which, in this recent application, learns a hierarchy of representations from object parts to scenes in both the generator and discriminator. StarGAN [17] allows simultaneous training of multiple datasets with different domains within a single network, which results in superior quality of translated images compared to existing models. These models are exceptionally powerful but are extremely resource intensive to train. Some attention has been given to developing explicitly *resource efficient* models. For example, in the work [22], the authors evaluate a series of models under constrained time cost, and propose models that are fast for practical applications yet are more accurate than existing fast models. However, models like these are currently exceptionally rare and it is up to computing platform providers to efficiently schedule training jobs, which as mentioned above is a very understudied facet of cloud computing resource allocation optimization which FlowCon aims to address directly.

This work studies the resource scheduling for containerized applications in cloud computing. The most of the relevant research solutions to container deployment are server consolidation or VM placement [28, 29, 34, 35, 37], which often focus on minimizing the total number of servers or the waste of resources. On the other hand, to efficiently use cloud resources, a lot of task scheduling solutions have been proposed. For example, the work [19] focuses on optimizing the makespan, the total average waiting time and the used hosts. Moreover, a lot of efforts have also been put on the designs of cloud scheduling systems. For example, the work [26] proposes a dependency-aware and resource-efficient scheduling which can achieve low response time and high resource utilization. In contrast to all these techniques, we focus on handling containerized deep learning applications rather than a general workload.

In terms of distributed machine (deep) learning, most frequently researchers and engineers train their models in a large cloud or cluster environment. Generally, when more resources are given to a specific job, the training time will decrease [21, 24, 27]. However, since resources are limited, providers must have a well-defined methodology for deciding which training jobs to prioritize. A very

limited amount of work has been done on this topic to-date. Recently, the authors of Gandiva [36] developed a new cluster scheduling framework that leverages the intra-job relationships of multiple experimental variations of the same deep learning job running concurrently to improve efficiency by prioritizing or killing some subsets of the set of related jobs being trained in a GPU cluster. Also working with deep learning on GPUs, the work [23] analyzes the effects of various scheduling decisions, including gang-scheduling and failure handling during training, on the resource utilization in a large multi-tenant GPU cluster. The most directly rated to our proposed solution is SLAQ [38]. The approach schedules concurrent machine learning training jobs based on quality improvement for resource usage, by allocating cluster resources iteratively. However, SLAQ fails to allocate the resources at real-time.

7 CONCLUSION

In this work, we have proposed `FlowCon`, which aims to facilitate dynamic resource allocation for containerized deep learning training jobs at runtime. We have presented the detailed system design of `FlowCon`, and conducted extensive experiments with six different deep learning models on two frameworks, Pytorch and Tensorflow, in a cloud computing environment. Our experimental results have shown that `FlowCon` is very efficient. Specifically, compared to a current system, it has achieved significant performance improvement in the presence of various deep learning workloads, by up to 42.06% reduction in completion time for individual jobs without sacrificing the overall system makespan.

ACKNOWLEDGMENTS

Long Cheng thanks the support of the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 799066.

REFERENCES

[1] Convolutional Neural Network. https://en.wikipedia.org/wiki/Convolutional_neural_network
[2] Docker. https://www.docker.com/
[3] Google AdSense. https://www.google.com/adsense
[4] GRU. https://en.wikipedia.org/wiki/Gated_recurrent_unit
[5] Keras. https://keras.io/
[6] Kubernetes. https://kubernetes.io/
[7] LSTM. https://en.wikipedia.org/wiki/Long_short-term_memory
[8] MNIST. http://yann.lecun.com/exdb/mnist/
[9] NSF Cloudlab. https://cloudlab.us/
[10] Pytorch. https://pytorch.org/
[11] Pytorch Examples/Dataset. https://github.com/pytorch/pytorch
[12] Recurrent Neural Network. https://en.wikipedia.org/wiki/Recurrent_neural_network
[13] TensorFlow. https://www.tensorflow.org/
[14] Tensorflow Examples and Datasets. https://github.com/floydhub/dl-docker
[15] VAE. https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
[16] Mathias Berglund, Tapani Raiko, Mikko Honkala, Leo Kärkkäinen, Akos Vetek, and Juha T Karhunen. 2015. Bidirectional recurrent neural networks as generative models. In *Advances in Neural Information Processing Systems*. 856–864.
[17] Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. 2018. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*. 8789–8797.
[18] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*. 1251–1258.
[19] Lucio Grandinetti, Ornella Pisacane, and Mehdi Sheikhalishahi. 2013. An approximate ϵ-constraint method for a multi-objective job scheduling in the cloud. *Future Generation Computer Systems* 29, 8 (2013), 1901–1908.
[20] Hank H Harvey, Ying Mao, Yantian Hou, and Bo Sheng. 2017. Edos: Edge assisted offloading system for mobile devices. In *2017 26th International Conference on Computer Communication and Networks*. 1–9.
[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at Facebook: a datacenter infrastructure perspective. In *Proc. IEEE International Symposium on High Performance Computer Architecture*. 620–629.
[22] Kaiming He and Jian Sun. 2015. Convolutional neural networks at constrained time cost. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*. 5353–5360.
[23] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. *Multi-tenant GPU clusters for deep learning workloads: Analysis and implications*. Technical Report. MSR-TR-2018.
[24] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proc. 15th International Conference on Information Processing in Sensor Networks*. 23.
[25] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. 2017. A survey on deep learning in medical image analysis. *Medical Image Analysis* 42 (2017), 60–88.
[26] Jinwei Liu and Haiying Shen. 2016. Dependency-aware and resource-efficient scheduling for heterogeneous jobs in clouds. In *Proc. 2016 IEEE International Conference on Cloud Computing Technology and Science*. 110–117.
[27] Y. Mao, V. Green, J. Wang, H. Xiong, and Z. Guo. 2018. DRESS: Dynamic RESource-Reservation Scheme for Congested Data-Intensive Computing Platforms. In *Proc. 11th IEEE International Conference on Cloud Computing*. 694–701.
[28] Ying Mao, Jenna Oak, Anthony Pompili, Daniel Beer, Tao Han, and Peizhao Hu. 2017. Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *Proc. 36th IEEE International Performance Computing and Communications Conference*. 1–8.
[29] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. 2013. Adaptive resource configuration for cloud infrastructure management. *Future Generation Computer Systems* 29, 2 (2013), 472–487.
[30] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).
[31] Tara N Sainath, Oriol Vinyals, Andrew Senior, and Haşim Sak. 2015. Convolutional, long short-term memory, fully connected deep neural networks. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*. 4580–4584.
[32] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proc. 15th Annual Conference of the International Speech Communication Association*.
[33] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117.
[34] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. 2010. Resource allocation algorithms for virtualized service hosting platforms. *J. Parallel and Distrib. Comput.* 70, 9 (2010), 962–974.
[35] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. 2011. SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In *Proc. 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 195–204.
[36] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *Proc.13th USENIX Symposium on Operating Systems Design and Implementation*. 595–610.
[37] Zhen Xiao, Weijia Song, and Qi Chen. 2013. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1107–1117.
[38] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. SLAQ: Quality-driven scheduling for distributed machine learning. In *Proc. 2017 ACM Symposium on Cloud Computing*. 390–404.
[39] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. 2015. Conditional random fields as recurrent neural networks. In *Proc. IEEE International Conference on Computer Vision*. 1529–1537.