2-2019

FC2: Cloud-based cluster provisioning for distributed machine learning

Nguyen Binh Duong TA
Singapore Management University, donta@smu.edu.sg

Cluster Computing (2019) 22:1299–1315
https://doi.org/10.1007/s10586-019-02912-6

FC^2: cloud-based cluster provisioning for distributed machine learning

Ta Nguyen Binh Duong[1]

Received: 25 September 2018 / Revised: 13 January 2019 / Accepted: 25 January 2019 / Published online: 8 February 2019

Abstract
Training large, complex machine learning models such as deep neural networks with big data requires powerful computing clusters, which are costly to acquire, use and maintain. As a result, many machine learning researchers turn to cloud computing services for on-demand and elastic resource provisioning capabilities. Two issues have arisen from this trend: (1) if not configured properly, training models on cloud-based clusters could incur significant cost and time, and (2) many researchers in machine learning tend to focus more on model and algorithm development, so they may not have the time or skills to deal with system setup, resource selection and configuration. In this work, we propose and implement FC^2: a system for fast, convenient and cost-effective distributed machine learning over public cloud resources. Central to the effectiveness of FC^2 is the ability to recommend an appropriate resource configuration in terms of cost and execution time for a given model training task. Our approach differs from previous work in that it does not need to manually analyze the code and dataset of the training task in advance. The recommended resource configuration can then be deployed and managed automatically by FC^2 until the training task is completed. We have conducted extensive experiments with an implementation of FC^2, using real-world deep neural network models and datasets. The results demonstrate the effectiveness of our approach, which could produce cost saving of up to 80% while maintaining similar training performance compared to much more expensive resource configurations.

Keywords Distributed machine learning · Cloud-based clusters · Resource recommendation · Cluster deployment

1 Introduction

In machine learning (ML), we aim to learn models from training data, and use them to make predictions on new data. A ML model has to be trained with data first before it can be used. Training ML models such as deep neural networks [1] with large amounts of data is an iterative task which requires high performance, distributed computing infrastructure to reduce the training time, which could be several days or weeks on a single system. Fast, resource-efficient ML model training is an important problem as such tasks would be repeated many times for fine-tuning of model's parameters; and users usually have budget constraints in terms of computational resource cost. Public cloud resources, such as those provided by Amazon EC2, Azure, etc., offer a compelling alternative to in-house dedicated clusters, due to the on-demand, pay-as-you-go pricing model and flexible, seemingly unlimited resource capacity.

Optimizing resource cost and performance for cloud-based distributed ML is challenging due to several reasons: (1) there are many possible configurations which could produce drastically different execution times, e.g., number workers or parameter servers [2], network latency and bandwidth, dataset or model partitioning strategies, model-specific parameters such as number of neurons and their connectivity, etc.; (2) most cloud providers offer a wide range of resource types with varying levels of performance and pricing; and (3) training large ML models with lots of data is compute-intensive and time-consuming. Indeed, ML researchers often find that setting up and maintaining a distributed computing cluster a hassle which takes away precious time from their core research activities [3].

Till date, not much research has been done to effectively bridge the gap between machine learning and distributed

✉ Ta Nguyen Binh Duong
donta@ntu.edu.sg

1 School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore

cloud computing. Most current setups require significant domain expertise and manual system tuning to achieve a desirable cluster configuration, which could be sub-optimal: recent work [4] demonstrated that a good configuration can be 20x faster in distributed model training compared to a sub-optimal configuration, while producing similar accuracy for the output models. Such performance gap could be much more for larger-scale setups. As ML model training may take days and be repeated many times to find a good set of hyper-parameters and neural network architectures, empirically exploring many possible cluster configurations is simply not practical.

Recently, cloud-based ML services such as Amazon Machine Learning [5] or Azure ML Studio [6] have been popularized. Such services offer intuitive interfaces, simple built-in ML models and algorithms for laymen to quickly harness the power of ML and big data. However, in these services, optimizing resource cost and performance with regard to distributed training still requires much manual effort. Popular ML packages like MXNet [7], TensorFlow [8], etc., focus on providing programming supports and leave tedious system management issues for end users to handle.

In this work, we investigate resource recommendation techniques to efficiently handle distributed ML model training over public cloud infrastructures. We propose FC^2 (Fast, Convenient, and Cost-effective), a system designed to handle complexity and heterogeneity inherent in public cloud resources; while providing a simple web-based interface for ML researchers and laymen to train complex, distributed ML models quickly and cost-effectively. We have made the following contributions in this paper:

– We consider the problem of distributed ML training over cloud resource. We then develop a simple but effective resource recommendation algorithm which can suggest a good cluster setup to reduce the training time and cost for a given ML model and dataset. Our approach is different from previous work in this area in that it does not need to manually analyze complex ML code and dataset to estimate the potential training time. Instead, we only make use of resource information and the scalability properties of a ML task to suggest an appropriate cluster setup.
– We develop an easy to use web/mobile interface for supporting simple cloud-based distributed ML model training. Users only have to upload their code, specify URLs to training datasets; and the appropriate resource selection, system configuration and deployment will be carried out by FC^2 automatically.
– We conduct extensive experiments with real-world deep neural network models and datasets to validate the effectiveness of our proposed approach. The results demonstrated significant cost savings of up to 80%, while maintaining similar levels of training performance in terms of execution time, compared to more expensive resource configurations.

We continue this paper with a thorough review of related work in Sect. 2. Sections 3 and 4 discuss the objective and approach of this study. Sections 5 and 6 detail the core of FC^2: the resource recommendation algorithms and their implementation. Sections 7 and 8 describe our evaluation methodology, experimental results and analysis. Section 9 concludes the paper.

2 Related work

2.1 Overview of distributed machine learning

Recently, ML models like deep neural networks have had great success in many challenging artificial intelligence problems such as speech/image/video recognition [1, 9–12], image segmentation [13], machine translation [14], or even playing complex games such as Go [15]. To be effective, these ML models need large amounts of data, as evidence suggested that model accuracy improves with regard to the increasing sizes of models and training data. For example, millions of labelled images were used to train neural networks having billions of connections resulting in very high recognition accuracy [16, 17]. In [18], hundred thousands of video clips [19] have been used to recognize many classes of human actions. AlphaGo [15] was able to beat world-class players using training data consisting of more than 100,000 recorded games played by human experts.

Usually, a distributed computing infrastructure is required to handle such large-scale model training to achieve a reasonable completion time, which could take days. This has led to the development of a few distributed ML frameworks, for example TensorFlow [8], SINGA [20], MXNet [7], Petuum [21], etc. Such frameworks are mostly based on the parameter server paradigm, in which data or model are partitioned/replicated across a set of worker nodes. A number of parameter servers are in charge of maintaining the global state of model's parameters. Distributed ML models are usually trained iteratively using stochastic gradient descent (SGD), in which workers need to exchange newly computed gradients via the set of parameter servers [2].

Existing ML frameworks focus more on providing programming supports and libraries for the development of new ML models and algorithms; and model-specific optimization to improve accuracy and training time. ML researchers still have to spend a considerable amount of time to setup and maintain systems, and to select an appropriate configuration for training, such as the number of workers/parameter servers and their corresponding resource configurations [22]. Such decisions require significant expertise in the domain of distributed systems, which many ML researchers may not have, or simply do not have enough time to investigate. As it is not cost-effective to maintain a large, dedicated computing cluster in most practical situations, on-demand cloud computing is a suitable alternative [23]. However, resources offered by public cloud providers are diverse in terms of pricing and performance [24]. Budget constraint is also another issue as cloud resources are not that cheap in the long run. For instance, the on-demand price of an AWS EC2 *p2.16xlarge* instance is more than $27 per hour (latest pricing as of August 2018, Singapore region).

We have carried out extensive literature survey, and found that not much research has been done in bridging the gap between cloud-based distributed computing and scalable machine learning. In the following, we classify existing research efforts into several categories. First, we discuss work that directly addresses the issues of performance prediction and automatic cluster deployment for training large-scale ML models. Second, we look at research dealing with performance optimization for ML training tasks in distributed and cloud-based computing clusters. Finally, we review currently popular cloud-based, on demand ML services which provide friendly interfaces and visual supports for laymen to train ML models.

2.2 Performance prediction and automatic deployment for distributed ML model training

In order to automatically provision a cluster of suitable machines for ML model training, we first need to estimate the performance for each candidate cluster configuration. Feng et al [4] presented one of the first studies in the area of automatic cluster configurations for distributed ML model training. The authors developed a scalability optimizer which could automatically choose a good configuration, i.e., number of workers and parameter servers, for distributed ML training. To do so, the optimizer will need to know the neural network architecture and other model-specific parameters, which might not be always available. Furthermore, this approach has been designed considering local, dedicated clusters, which might have a limited number of homogeneous nodes. However, cloud resource configurations are diverse and much more varied in terms of performance. Resource cost, which is a key issue in cloud deployment and multi-user systems, were also not taken into account [4]. The same authors [25] considered cloud-based setups for ML. However, it was more for fast ML model serving, not distributed model training.

More recently, [3] proposed a method for estimating the speedup ratio of distributed ML training which might be achieved when more workers are added to the system. The approach requires analyzing the ML code and calculating the amount of floating computations/parameters that are present in the model. It could be a challenging and time-consuming process when complex ML code written by unknown users are analyzed. Security and copyright reasons may also make it not possible to do so. Furthermore, the developed method was evaluated on Apache Spark using a dedicated commodity cluster, not public cloud resource.

Apache SINGA [20] is a distributed ML framework which supports both synchronous and asynchronous ML model training. It provides a number of built-in model partitioning strategies so that finding a good training configuration becomes somewhat easier, but still largely a manual process. In addition, SINGA has not considered the issues of resource cost optimisation.

Ako [22] is a recently proposed decentralized ML system supporting distributed model training. It does without parameter servers by having all nodes in the cluster as worker processes. Workers compute gradients and exchange partial updates directly with each other, subject to bandwidth availability. Ako does not require resource configuration decisions, i.e., one does not need to determine the appropriate number of workers and parameter servers to fully utilize the cluster's resource. Similarly, Horovod [26], which has been developed recently at Uber, lets workers communicate directly by organizing them in a ring. These systems do not address the issues specific to cloud-based deployments such as cost and selections of various resource types.

The authors of [27] developed a performance model for the distributed training of deep convolutional neural networks using asynchronous GPU computation with mini-batch SGD. The model considers the batch sizes, neural network architectures and worker specifications to predict the execution time given a training dataset. Such prediction can then be used to choose the fastest server configuration. This model has been designed and empirically evaluated with supercomputers consisting thousands of dedicated GPUs in mind. The authors did not show how such model would be applicable for performance prediction and automatic configuration selection on public cloud resources.

There have been some research in the area of performance prediction for applications running on public cloud infrastructures. CloudProphet [28] focused on the problem of selecting the best-performing cloud providers for a given application. It aimed to predict an application's performance when running on a chosen cloud platform, without actual deployments due to cost or security concern. On the other hand, empirical approaches including [29] evaluate the application's performance on actual cloud infrastructures, with the aim of developing automated methods to deploy and test applications using synthetic workloads in advance. RA^2 [24] predicted the execution time of cloud-based simulations via a data-driven approach. In [30], the authors used a simulation-based algorithm to predict application execution times with respect to cloud configuration changes. In the most recent work [31], a classifier has been developed to characterize the computing footprint of an application, and then to match this application with the right cloud resource. Although interesting and practical, these existing approaches have been designed specifically for web and other enterprise applications, not distributed ML model training.

2.3 Performance optimization in distributed ML clusters

Recent research have been focusing more on performance optimization techniques for training large ML models, e.g., loose synchronization methods, data filtering, communication and job scheduling, etc. We review them here as these techniques have a direct impact on training performance and resource selection techniques in distributed ML. In [32], distributed ML execution threads could use loose synchronization models and stale shared data to reduce network communication costs. In [33], we developed network optimization techniques including parameter storage, gradient and parameter filtering to reduce communication overhead and improve training time in distributed ML clusters. In [34], a dynamically-partitioned cluster management mechanism and an utilization-fairness optimizer have been implemented. Empirical performance measurements then demonstrated significant speed gains and better resource utilization in ML training clusters.

In [35], the authors considered using only ternary gradients, i.e., gradients that are quantized to ternary levels, to reduce the overhead of network synchronization. This in turn helps to accelerate distributed deep learning under data parallelism. A performance model has also been developed to study and demonstrate the scalability as well as speedups of the proposed mechanism. Similarly, [36] proposed to use just 1-bit SGD to minimize the communication overhead in distributed training of speech recognition models. In [37], investigations showed that most of the gradient exchange in distributed SGD are redundant. The authors then proposed a method called Deep Gradient Compression to reduce the network bandwidth consumption in the ML training cluster which is based on commodity Ethernet and mobile devices.

In [38], a deep learning cluster scheduler named Optimus has been proposed. The authors argued that existing cluster schedulers have not been tailored to deep learning jobs, preventing the cluster to achieve high resource efficiency and performance at the same time. Optimus aims to minimize ML task training time using online fitting techniques to predict ML model convergence during training, and to estimate training speeds with regard to resource allocations. The performance predictions then will be used to dynamically provision compute resources and place ML tasks accordingly to reduce completion time.

We note that these techniques have been demonstrated to reduce training time and improve resource utilization in ML clusters. However, none of them have directly addressed the issue of cloud-based cluster setups and automatic deployment for ML model training. We believe that our approach in this paper could nicely complement existing performance optimization techniques in the public cloud context.

2.4 Commercial cloud-based ML services

Due to the currently strong demand in easy-to-use data science tools, multi-user cloud-based ML services have been getting popular, e.g., those currently offered by Amazon ML [5], Azure ML Studio [6], Google Cloud AI [39], or BigML [40], to name a few. These services provide user-friendly interfaces and built-in ML models which are ready to be put into usage. Users can also make use of distributed GPU/CPU training capability offered to speed up the process of tuning hyper-parameter and model architectures, at a cost. There have also been some supports in deep learning cluster setup and management. For instance, Amazon took a first step in the right direction by introducing the Deep Learning AMI [41] early 2017, which is a template for creating virtual machines pre-installed with ML packages such as MXNet. Using the template, users can create on-demand deep learning clusters more easily via AWS CloudFormation [42].

We note that existing cloud-based ML services still do not really provide much controls and optimizations for distributed ML model training, especially in the case of budget-conscious users. In particular, the question of how to configure the appropriate sets of workers/parameter servers remains open. Well-known ML frameworks, e.g., MXNet [7], Petuum [21], TensorFlow [8], etc., provide excellent libraries, programming models, and ML model-specific optimisations, but they do not deal directly with distributed system setup and management issues.

3 Objective and scope

The wide variety of resource configurations, their performance levels and prices offered by public clouds provides the much-needed flexibility for end users running various applications and workloads. At the same time, this also creates difficult issues with regard to resource selection and cost management. It is well-known that ML model training needs to be done repeatedly to obtain good hyper-parameters such as biases, learning rates, etc. This process is intensive in terms of both cost and time [2]. Therefore, the choice of a suitable resource configuration would potentially yield significant improvements in training time, and vice versa. As cloud resource is typically billed per unit of time, a faster training time could translate to greater cost saving.

In this work, our aim is to alleviate the problem of cloud resource selection and configuration for distributed ML training, so that ML researchers would be able to focus solely on their ML model development tasks. We consider the training of large ML models using stochastic gradient descent (SGD) [43], which is the standard technique applied to a wide variety of models such as logistic regression or deep learning networks [16]. In gradient descent, a cost function computed using the ML model's parameters and the training data is iteratively optimized. To speed up the training, usually a data-parallel approach[1] is employed: the training dataset is partitioned over a cluster of worker nodes. Each of the node computes the gradients in parallel, and the results are aggregated at one or more server nodes which are referred to as parameter servers (PS) [44]. These servers maintain the ML model's parameters and broadcasts the latest values to all workers.

In this paper, we consider ML training clusters composed of virtual machines (VMs) acquired on-demand from public IaaS cloud providers such as AWS EC2. **The objective of this work is then two-fold**:

(1) We investigate cloud resource recommendation algorithms for training arbitrary ML models and datasets using the PS framework so that both training time and cost could be minimized.
(2) We develop an easy-to-use system to support automatic resource configuration, deployment and execution for distributed ML model training over public cloud resources.

[1] Model-parallel is another approach to speed up the training, which is beyond the scope of this paper.

4 The FC^2 approach to distributed machine learning

In this section, we describe our approach to convenient and cost-efficient distributed ML model training over resource acquired from public IaaS clouds. We start with describing the architecture of the web-based ML system. In the next section, we follow with the resource recommendation algorithms which constitute the core of our system.

Figure 1 shows the architecture and various components in our proposed system.

4.1 Web/mobile interface

The FC^2 system provides an easy to use interface so ML researchers can focus solely on their model and algorithm development. ML code could simply be packaged (e.g., in a Python wheel bundle) and uploaded via the web interface. Training data could also be uploaded or specified using external URLs. The user then can move on to specify his/her budget for the model training process; or rely on the resource recommendation algorithms to suggest an appropriate cluster setup to run the training. The ML model training could then be submitted; and results would be made available on the web interface for users to download. Trained models could also be deployed, e.g., via TensorFlow Serving, to service online classification/regression requests. Figure 2 illustrates a typical model training workflow in FC^2.

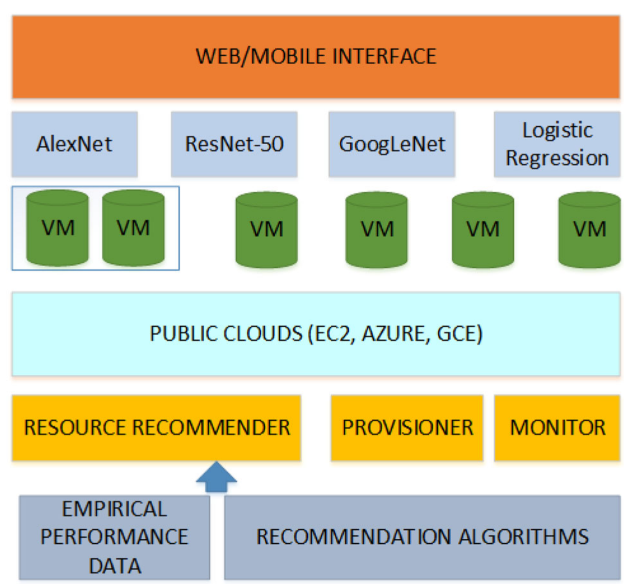

Fig. 1 An architectural overview of FC^2. The system supports fast, easy ML model training with a budget in mind

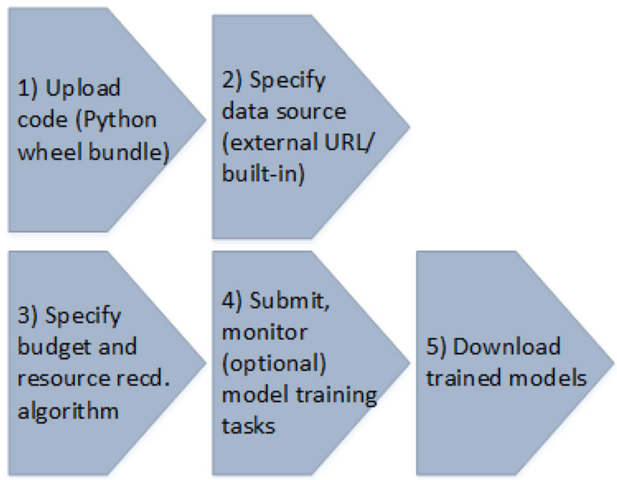

Fig. 2 A typical ML model training workflow in FC^2. Users can submit packaged code, specifying training datasets, providing information such as budget, and submitting training tasks with recommended cluster setups. Trained models will be available for downloading from the web interface

4.2 Recommender

The resource recommendation component aims to predict the most appropriate cluster setup to run a particular ML model training, given the model code and dataset specified by users. It takes input from a database which stores empirical performance data obtained from past executions. In Sect. 5, we define the resource recommendation problem, and describe several heuristic algorithms which have been implemented in our system. The recommended cluster configuration contains information such as cloud instance types, number of workers, selection of parameter servers, etc.

4.3 Provisioner

This component takes a cluster configuration from the Recommender, connects to a public cloud provider and provision the required resource. It will also automate various tasks in cluster setup for distributed ML training such as network or data storage configuration so that a ML researcher does not have to do this manually.

4.4 Monitor

This component is responsible for monitoring ML task executions and the status, e.g., network bandwith and CPU/GPU utilization, of the cluster provisioned for each training task. It also collects empirical performance data which could be necessary for the resource recommendation algorithms.

5 Resource recommendation

5.1 Problem definition

The resource recommendation problem is defined as follow. Given an indicative budget C, find a cluster setup consisting of parameter servers and workers so that the model training cost and/or time would be minimized. For simplicity, we consider cluster setups which use a single parameter server and the same cloud instance type for workers. Such setups are actually quite popular for data-parallel ML training [3].

The resource recommendation stated above is a challenging problem. Given a ML training problem (model code and dataset), there is a large number of potential cluster setups due to various cloud resource types, their performance levels and pricing offered by public cloud providers like AWS EC2. Each combination of resource types in a computing cluster may produce drastically different training time, or model accuracy. In addition, due to model and code complexity, it is difficult to derive the expected training time of a given ML model beforehand [4]. Therefore, searching for an optimal configuration which could minimize both resource cost and training time might not be possible due to time and budget limitation.

In the following sections, we describe several heuristics which aim to suggest a suitable cluster configuration quickly and efficiently. Our proposed algorithms are different from previous work such as [3] in that they do not need to analyze the code and dataset of the ML training task in advance, which could be a complex and time-consuming task. Instead, our algorithms are resource-aware, in the sense that they make use of resource information and previous empirical performance data to suggest a cluster setup.

5.2 Algorithms

We adopt a two-stage approach in recommending a cluster configuration. In the first stage, the parameter server for the training cluster will be selected. In the next stage, the algorithms will then recommend the appropriate instance type and the number of workers in the cluster.

5.2.1 Selecting the parameter server

FC^2 provides a list of suitable instance types which can be used as parameter servers. In distributed ML training, the parameter server only needs to maintain and communicate the model's parameters, so a medium-sized general purpose instance such as AWS EC2's *m*4.*large* or *m*4.2*xlarge* would be sufficient in many cases. Given the list L^p of eligible instance types, users can manually specify the type of parameter server depending on their budget. Otherwise, the resource recommendation algorithms would pick one with the largest network (bandwidth) capacity from the list.

5.2.2 Cost optimization (cost-opt)

This algorithm aims to minimize the total resource cost when running ML training tasks. At first, FC^2 automatically provides a pre-defined list L^w of CPU or GPU instance types which could be suitable for ML model training. As a model-agnostic algorithm, *Cost-Opt* would only make use of the resource pricing information to select the cheapest cloud instance types for the execution. This selection is subjected to the indicative per-hour budget C which should be set by the user in advance. For example, the user might specify that he is willing to spend around $2 per hour to train his ML model. Alternatively, a user can also set a certain limit on the number of workers in the training cluster. With a given budget C, *Cost-Opt* then calculates the total number of workers needed as follow, assuming the per-hour cost for the parameter server is c_{ps}:

`The Cost-Opt Algorithm:`

(1) Select the cheapest instance type t from L^w, and obtain its per-hour cost c_t.
(2) Calculate the number of workers needed for the cluster: $n_t = (C - c_{ps})/c_t$.

The *Cost-Opt* algorithm mainly serves as a point of comparison with other algorithms. Cheaper instance types may reduce the cost, but their potentially inferior performance may prolong the training time, leading to more cost in the end. However, our experiments demonstrate that in some cases, cheaper instances could produce similar or even better performance compared to the more expensive types.

5.2.3 Runtime optimization (time-opt)

This algorithm aims to minimize the total execution time for a ML training task by selecting the most expensive cloud instance type from a predefined list L^w for the execution. This selection is also subjected to an indicative per-hour budget C, or a maximum number of workers which should be set by the user in advance. *Time-Opt* calculates the total number of workers needed using the below algorithm if C is given:

`The Time-Opt Algorithm:`

(1) Select the most expensive instance type t from L^w, and obtain its per-hour cost c_t.
(2) Calculate the number of workers needed: $n_t = (C - c_{ps})/c_t$.

At first, the *Time-Opt* algorithm may seem not very cost-efficient. However, we note that current cloud billing models are usually per unit of time e.g., hour or second. A more expensive resource type, for example AWS EC2's *p*2 or *g*3 instances which are GPU-based, would be able to complete deep neural network training tasks, e.g., for image recognition, much faster compared to cheaper instances such as the CPU-based *m*4 instances. In this way, the total cost of using more expensive workers may not be more than that of a cluster composed of cheaper-priced workers.

We also note that for both *Cost-Opt* and *Time-Opt*, the user may also choose to specify a maximum number of workers instead of an indicative budget. In this case, these two algorithms would only need to look at the list of pre-defined instance types L and select the cheapest or most expensive type, respectively.

5.2.4 Scalability optimization (scala-opt)

In this algorithm, we find an optimized cluster configuration by exploiting the scalability properties of a distributed ML training setup based on the PS framework [2]. More specifically, *Scala-Opt* estimates the number of workers that should be deployed in a cluster using the network bandwidth utilization of the given ML task. In order to do this, *Scala-Opt* would need to collect some bandwidth utilization data first by bench-marking the particular ML task for a very short duration using the smallest cluster setup available, e.g., a cluster with only one parameter server and one worker. We note that such data collection task may increase the overall cost and time of ML model training. However, for training tasks that last days or weeks, a few minutes of added time could be considered negligible. Furthermore, a ML training task could be repeated many times, while our algorithm may need to collect the bandwidth utilization data only once. Such data could also be stored for future usage with similar ML training tasks.

For flexibility, we develop two versions of *Scala-Opt*. In the first version, a user may have the option to manually specify the instance type for workers. The algorithm will then recommend a suitable number of workers for the cluster. We refer to this version as *Scala-Opt-M*. In the second version, users may leave both the tasks of choosing instance type and number of workers for the algorithm. In the following, we mainly describe the second version under the name of *Scala-Opt*, with some notes applied for *Scala-Opt-M*. We denote L^w as the list of all possible instance types the user would like to consider as workers for his task. We denote that the parameter server's per-hour cost as c_{ps}, and its bandwidth capacity as B_{ps}. *Scala-Opt* then calculates the appropriate number of workers using the below algorithm:

`The Scala-Opt Algorithm:`

(1) Remove the most expensive instance type t from the list L, and estimate its bandwidth utilization b_t (in Mbps) for the given ML task. This step can be skipped if b_t is already available due to previous runs.

In the *Scala-Opt-M* version, a user can select the type of workers from the list L manually, so we can skip this step.

(2) Calculate the number of workers: $n_t = \min\{(B_{ps} \times p)/b_t, (C - c_{ps})/c_t\}$, where $0 < p < 1$.

(3) If n_t is smaller than the currently chosen value,[2] choose n_t as the number of workers needed. Else, repeat step (1) and (2) until all instance types in the list have been considered.

In this algorithm, we consider all possible instance types ordered according to their per-hour price. Step (2) calculates the number of workers for a given instance type t, starting with the most expensive one, subject to an indicative per-hour budget C and the bandwidth constraint B_{ps}. The parameter p, which could be set to a value close to 1, for example 0.8, is there to ensure that the bandwidth capacity of the parameter server would not be close to saturation by the workers' aggregated bandwidth. *Scala-Opt* considers the more expensive instances first since they might have much better computation performance, especially for deep neural network training. Higher-performing instances may generate more network traffic, i.e., higher values for the bandwidth utilization b_t, which in turn would reduce the number of workers calculated by this algorithm. However, there might be cases in which slightly cheaper instances could perform better. The algorithm accounts for that in Step (3), which aims to choose the smallest number of workers for a given indicative budget.

6 System implementation

We implement the FC^2 system described above using a mix of open-source tools and frameworks. The web interface has a responsive design, and has been implemented using Python/Django. Boto3[3] and Paramiko[4] are used for interfacing with AWS EC2 and to control cloud instances with SSH. Subprocess[5] is used to run ML tasks so that the system can employ some status monitoring mechanisms. When the ML training task is completed, a Python script will trigger an HTTP request from the task's cluster to update the web interface. Nethogs[6] is used to carry out bandwidth utilization measurements when running ML tasks for the first time using the *Scala-Opt* resrouce recommendation algorithm.

The system currently supports some of the most popular ML frameworks such as TensorFlow, MXNet and Apache Spark MLlib. In Fig. 3, users can upload the code for his ML model training in a Python wheel bundle, specifying the main script to be executed. They can then move on with supplying the training dataset, which could be a built-in one,[7] or via an external URL (Fig. 4). Figure 5 illustrates how a user can choose the computing resource manually or use system-recommended configurations.

7 Evaluation methodology

In this section, we describe the methodology used to evaluate the effectiveness of our proposed resource recommendation algorithms. The algorithms have been implemented into our FC^2 system.

7.1 ML model and dataset

Due to a limited budget for cloud resource, and the need to repeat the experiments many times to obtain reliable results, we mainly use the popular CIFAR-10 dataset which is available online at [45], and the TensorFlow ML framework to carry out the experiments. The CIFAR-10 dataset is a collection of small images which are frequently used to train or evaluate ML and computer vision algorithms. The dataset has 60000 colour images which are classified into 10 classes. 50000 images are used for training, and the rest are test images.

The ML model used in the experiments is a deep neural network consisting of convolution and non-linear layers, followed by fully connected layers, and a softmax classifier.[8] The model has more than a million of learnable parameters. In a distributed setting, the batch size which is the number of images processed in each time step might greatly affect the amount of computation a worker would have to carry out, as well as the network bandwidth utilization. In our experiments, we test several different batch sizes, e.g., 128, 512 and 1280, to evaluate its effect on the performance of our resource recommendation algorithms.

[2] n_t should be initialized to a very large value.

[3] https://github.com/boto/boto3.

[4] http://www.paramiko.org

[5] https://pymotw.com/2/subprocess.

[6] https://github.com/raboof/nethogs.

[7] FC^2 provides a number of the most popular training datasets via AWS Elastic File System.

[8] https://code.google.com/archive/p/cuda-convnet.

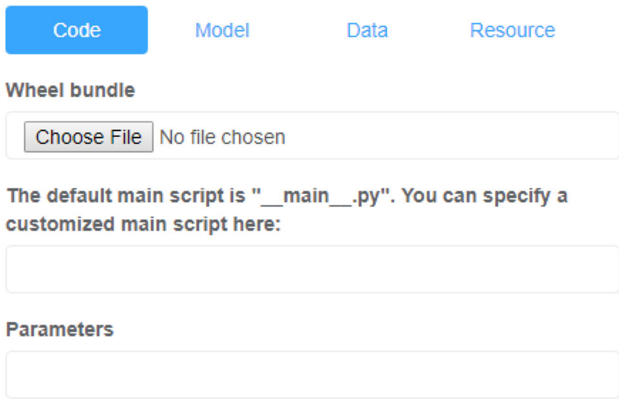

Fig. 3 Specifying ML model code in FC^2 with TensorFlow. The code should be packaged into a Python wheel bundle

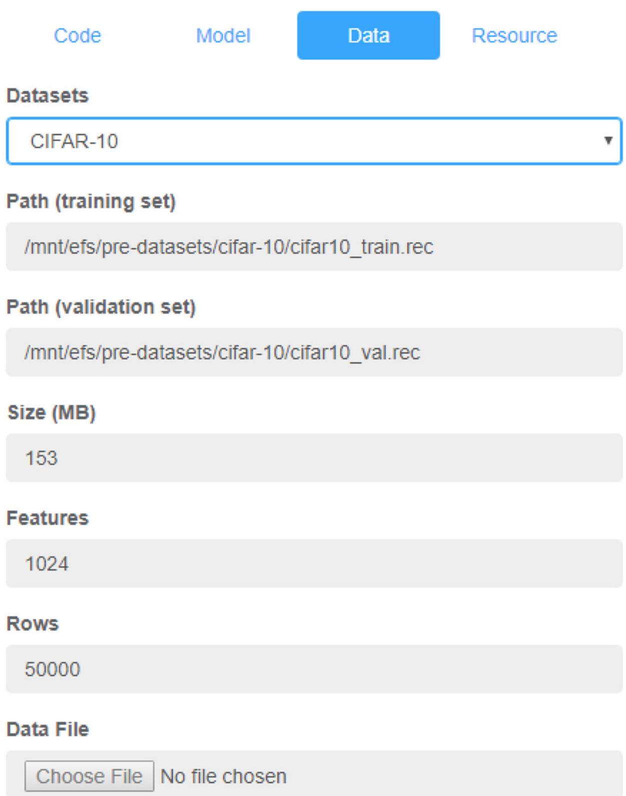

Fig. 4 Specifying training data in FC^2 with TensorFlow. Users can choose to use built-in datasets or an external URL pointint to their own datasets

7.2 Instance types and pricing

AWS EC2 provides many instance types with varying sizes and costs for different purposes. In our evaluation, we use instance types and pricing from the Singapore region. For parameter servers, we consider the general purpose *m*4 instances. Table 1 lists the prices and configurations for the considered instance types. Note that in the table, EC2

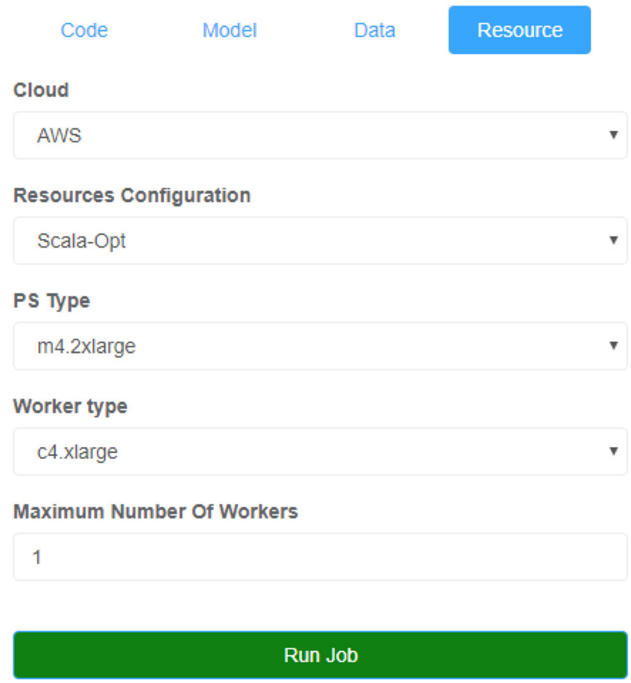

Fig. 5 Specifying computing resource for ML model training in FC^2 with TensorFlow. Users can do it manually, or rely on the built-in resource recommendation algorithms to setup the training cluster

Compute Unit (ECU) indicates the integer processing power of an AWS EC2 instance. In distributed ML training, a parameter server is mainly used for aggregating gradients computed by workers, and sending out the updated model parameters. Therefore, CPU-based instances such as *m*4 would be sufficient. The larger configurations, e.g., *m*4.10*xlarge*, tend to have much better network performance[9] at a significantly higher cost. Depending on the user's budget, an appropriate instance type could be selected from the given list. To avoid network saturation at the parameter server, we set the value $p = 0.95$ in the *Scala-Opt* algorithms.

For workers, a wide variety of EC2 instance types have been considered, namely the general purpose *m*4 and *t*2 instances, the compute-optimized *c*4, the GPU-based *p*2 and *g*3. These have been widely used for ML workloads and other enterprise applications. Table 2 lists the pricing and configurations for various instance types considered for workers in this paper. The *t*2 instances do not have a fixed level of CPU performance (variable ECU).

[9] EC2 only mentions that the network performance of these instance types is classified as High. More information is available from https://aws.amazon.com/ec2/instance-types/.

Table 1 Pricing and configuration for EC2 instance types considered for the parameter server

Instance type	Cost	ECU	Mem. (GiB)	Network
m4.large	0.125	6.5	8	Moderate
m4.2xlarge	0.5	26	32	High
m4.10xlarge	2.5	124.5	160	High

Table 2 Pricing and configuration for EC2 instance types considered for workers

Instance type	Cost	ECU	GPU	Mem.
m4.xlarge	0.25	13	-	16
t2.xlarge	0.2336	Variable	-	16
c4.xlarge	0.231	16	-	7.5
c4.8xlarge	1.848	132	-	60
p2.xlarge	1.718	12	1	61
g3.4xlarge	1.67	47	1	122

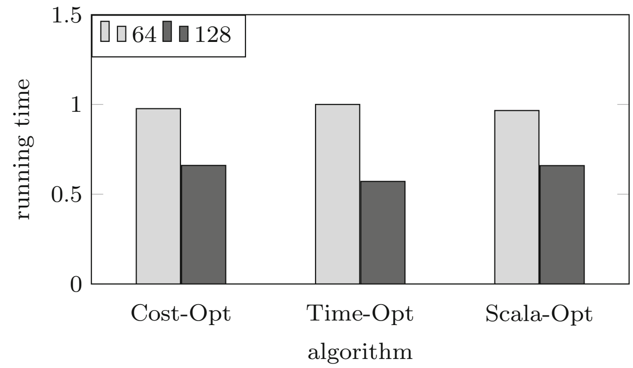

Fig. 6 Comparing model training times between resource recommendation algorithms (CPU-based instances only). We note that they have quite similar performance with various batch sizes. *Scala-Opt-M* (not shown here) also provides similar training times when users manually select either m4.xlarge or c4.xlarge for the workers

8 Results and analysis

In this section, we report the experimental results using various combinations of algorithms and configurations. We first describe the results obtained with CPU-based instances, i.e., when users have limited budget. We then move on to consider a mix of resource types ranging from cheap CPU instances to the more expensive GPU-based instances. We also look at the effect of expensive parameter servers having very high levels of network performance.

8.1 Using inexpensive CPU-based instances

For users with limited budget, they may want to opt for lower-priced CPU-based instances such as *m*4, *t*2 or *c*4. In this set of experiments, we consider only CPU-based instances for the ML training cluster. We also use an *m*4*.large* instance as the parameter server due to budget reason. We do not use larger batch sizes such as 1280 as such sizes would be too slow for CPU-based training. To set the indicative budget *C*, we choose a limit of 6 workers per cluster.[10] Figures 6 and 7 show the performance in terms of training time and cost for each algorithm, respectively. We observe that *Scala-Opt* produces similar training time to *Time-Opt* as shown in Fig. 6, albeit recommending smaller cluster sizes. More specifically, *Scala-Opt* recommends 2 and 3 workers of the instance type *t*2*.xlarge* for batch sizes of 64 and 128, respectively. At the same time, *Time-Opt* chooses 6 workers of the type *m*4*.xlarge* which is the more expensive instance type compared to *t*2*.xlarge*. This is mainly because in our experiments, the cheaper *t*2*.xlarge* instances provide better computation performance compared to *m*4*.xlarge*. Figure 6 also demonstrates that *Scala-Opt* performs similarly in temrs of training time to *Cost-Opt*, which selected 6 workers of the cheapest type *c*4*.xlarge*.

When comparing the resource cost, we observe that *Scala-Opt* results in the lowest cost compared to the other two algorithms. This is mainly because it chooses the cheaper *t*2*.xlarge* instances and a smaller number of workers. Figure 7 illustrates the cost savings. More specifically, the cost reduction of *Scala-Opt* when compared to *Time-Opt* is around 65% for the batch size of 64, and around 40% for the batch size of 128. If we let users choose instance type for workers manually, i.e., *Scala-Opt-M*, the cost reduction would be around 15% (not shown in Fig. 7) when *m*4*.xlarge* or *c*4*.xlarge* is selected.

Figure 8 and 9 provide a closer look at the training performance for various CPU-only instance types and cluster sizes. We observe that *t*2*.xlarge*, despite being relatively inexpensive, has the best performance in terms of training time. The figures also demonstrate that peak performance has been obtained from clusters of 3–4 workers. From that point, increasing the cluster size does not help much as the parameter server's network capacity has been saturated. This explains the effectiveness of our *Scala-Opt* approach, in which FC^2 estimates the bandwidth consumption of the ML task before actual training to limit the cluster size accordingly. As a result, *Scala-Opt* can provide comparable training time at a much lower resource cost.

[10] Similar results have also been obtained for larger cluster sizes.

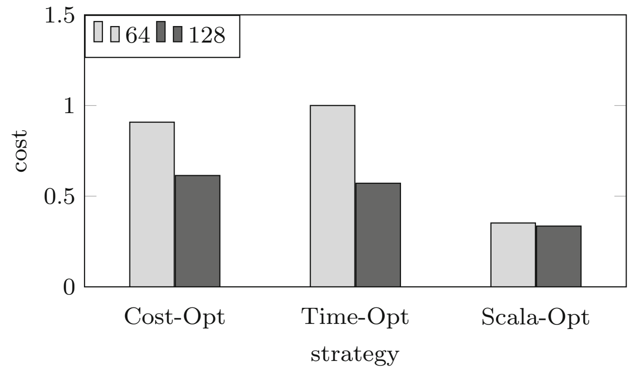

Fig. 7 Comparing resource cost incurred by each resource recommendation algorithm (CPU-based instances only). We note that *Scala-Opt* outperforms *Time-Opt* by as much as 65%. This is mainly because *Scala-Opt* uses a smaller number of cheaper workers to achieve similar training performance. For instance, when using a batch size of 64, *Scala-Opt* uses only 2 t2.xlarge workers compared to 6 m4.xlarge workers in *Time-Opt*, and 6 c4.xlarge workers in *Cost-Opt*

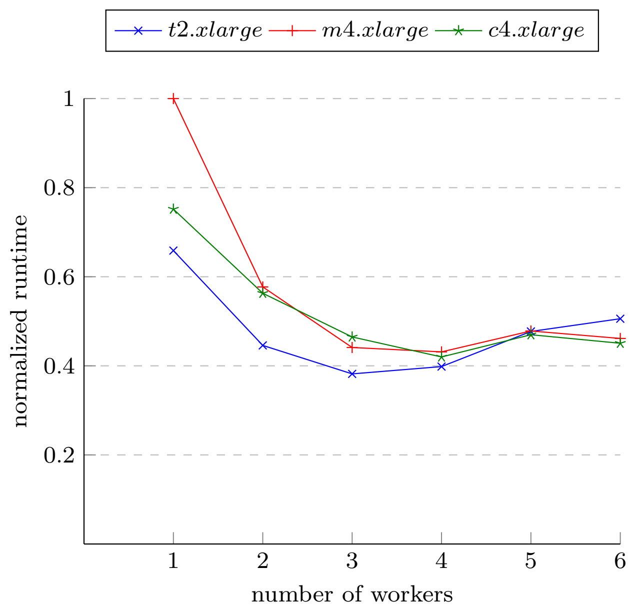

Fig. 8 Performance of CPU instances with various number of workers, batch size of 64

8.2 Using a mix of GPU and CPU instances

In this set of experiments, we consider a list of several instance types which could be used to run ML model training, namely *t*2, *m*4, *c*4 and the GPU-based *p*2. Similar to the above experiments, we assume the same instance type for the parameter server, and a limit of 6 workers per cluster. Figure 10 shows the execution time comparison between the proposed algorithms with various batch sizes used for training the neural network model. We observe very similar performance in most cases for the two algorithms *Time-Opt* and *Scala-Opt*. The *Time-Opt* algorithm would recommend a cluster of 6 *p*2*.xlarge* instances, which are the most expensive type in the list. On the other hand, *Scala-Opt* makes use of the available bandwidth information obtained via quick bench-marking to recommend smaller cluster sizes. More specifically, *Scala-Opt* recommends 2, 3 and 5 workers of the type *p*2*.xlarge* given the batch sizes of 128, 512 and 1280 respectively.

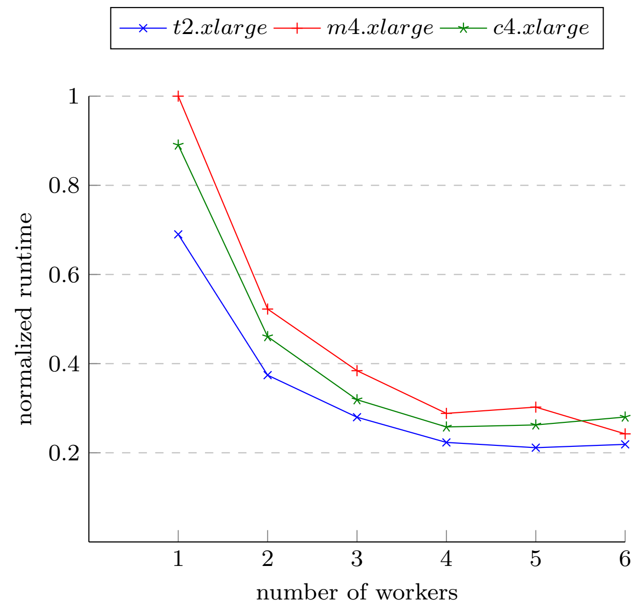

Fig. 9 Performance of CPU instances with various number of workers, batch size of 128

Figure 11 confirms that larger cluster sizes do not necessarily provide shorter training time. We note that for smaller batch size, e.g., 128, the workers could complete the computation faster. As a result, more data would be exchanged with the parameter server to update the ML model, leading to more bandwidth utilization. In the experiments, we observe that when using *p*2*.xlarge* which is GPU-based, a setup of more than 2 workers could easily saturate the network capacity of the parameter server, with a batch size of 128. Therefore, the *Scala-Opt* algorithm would recommend only 2 workers in this case.[11]. When using larger batch sizes such as 512 or 1280, the workers would take more time for computation due to the larger number of images in each batch. This would reduce network bandwidth traffic in the cluster, thus more workers could be used to speed up the computation without overloading the parameter server's network interface. For the largest batch size used in our experiments (1280), the *Scala-Opt* algorithm recommends around 5 *p*2*.xlarge* workers.

[11] The parameter p in the *Scala-Opt* algorithms is set to 0.95 to avoid bandwidth saturation at the parameter server, which has a capacity of around 450Mbps.

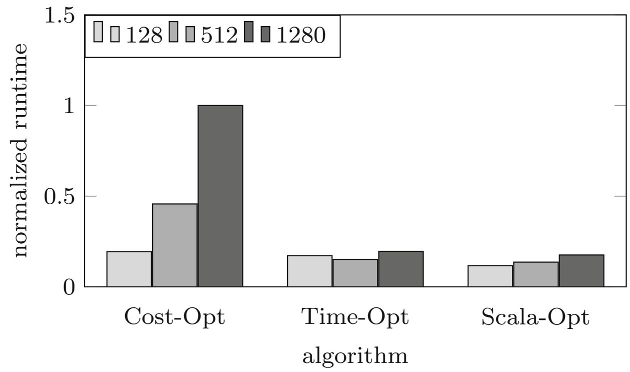

Fig. 10 Comparing execution times between resource recommendation algorithms. We note that *Time-Opt* and *Scala-Opt* have quite similar performance with various batch sizes, while *Cost-Opt* results in more time

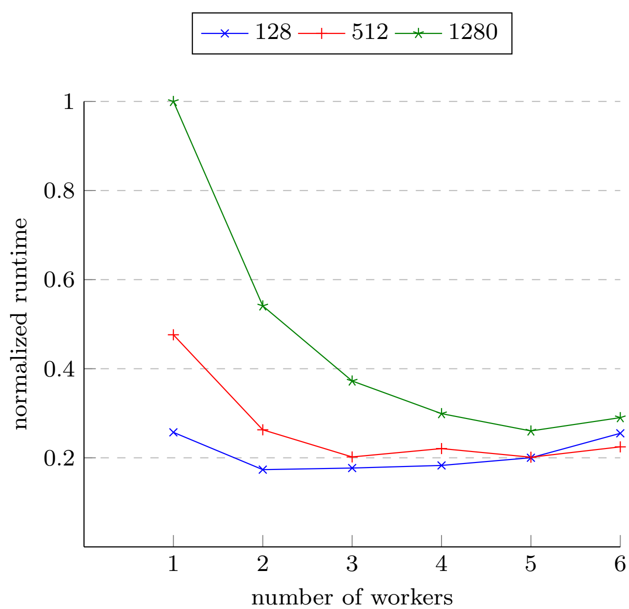

Fig. 11 Performance of p2.xlarge with various number of workers and batch sizes

It is not a surprise that *Cost-Opt*, which selects the cheapest instance type *c*4.*xlarge*, takes more time to complete the training compared to the other two algorithms. However, cheaper resource type does not necessarily reduce the total cost, as shown in Fig. 12. This is because a longer training time would lead to more cost; as cloud resource is charged per unit of time. Figure 12 also shows that the cost has been reduced significantly in *Scala-Opt* as compared to *Time-Opt*. More specifically, when the batch sizes are 128 and 512, the cost reductions are around 80% and 50%, respectively. This demonstrates the effectiveness of *Scala-Opt*, which provides almost the same level of training performance but with much less resource cost.

8.3 Using only high-performance instances

In this section, we present the results obtained when running the resource recommendation algorithms using a set of high-performance (and costly) instance types. This scenario is applicable for users with relatively higher budget. More specifically, we consider the following EC2 instance types: the GPU-based *g*3.4*xlarge* and *p*2.*xlarge*, and the CPU-based *c*4.8*xlarge*. These instance types have similar pricing as shown in Table 2. To handle these high-performance workers, a parameter server of the type *m*4.2*xlarge* which has a network capacity of around 1 Gpbs is used. Other settings and parameters are the same as in the previous experiments.

Figures 13 and 14 show the training time and cost of the three algorithms, respectively. We observe that *Cost-Opt* and *Scala-Opt* have quite similar performance in all cases, while *Time-Opt* results in more time especially for the larger batch size of 512. A closer look at Figs. 15 and 16 reveals the reason for such difference in training performance. Despite being the cheapest among the three, *g*3.4*xlarge*, which is a newer-generation instance type, outperforms the other instances namely *p*2.*xlarge* and *c*4.8*xlarge*. *Scala-Opt* has been able to make use of network utilization information to recommend only 2 and 4 *g*3.4*xlarge* workers for the clusters with the respective batch size of 128 and 512. As a result, while incurring less cost, its performance is quite similar to that of *Cost-Opt*, which uses 6 *g*3.4*xlarge* in all cases. We note that the training performance (with batch size of 128, Fig. 15) shows little improvement when increasing the cluster size beyond 2 workers, due to network saturation at the parameter server. The instance *p*2.*xlarge* has not been used in all the algorithms as it is neither the most expensive nor cheapest type. It also does not have the best performance according to the pre-run network benchmarking.

When the batch size is set to 512, more computation will be required per iteration. In this case, the gap in training performance becomes more obvious, as shown in Fig. 16. The most expensive instance type, *c*4.8*xlarge*, does not really provide the same level of performance compared to the other GPU-based instances. In the end, *Time-Opt* incurs about 65% more cost compared to *Scala-Opt*, while producing around 30% less training performance. This fact demonstrates the effectiveness of our proposed approach, and the importance of selecting the right resource type and cluster size when training large ML models.

8.4 Using large-capacity parameter server

In this set of experiments, we investigate the effect of using a parameter server with large network capacity on the

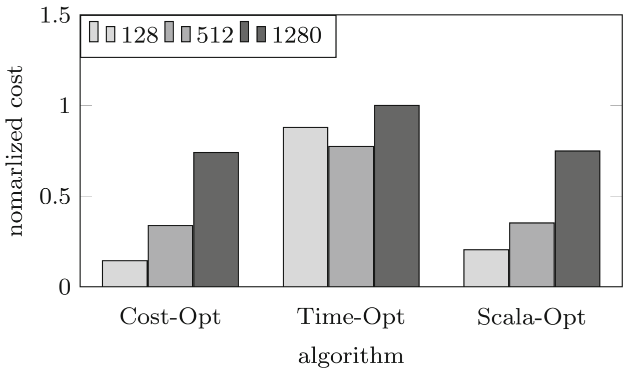

Fig. 12 Comparing resource cost incurred by each resource recommendation algorithm. We note that *Scala-Opt* outperforms *Time-Opt* by as much as 80%. The cost incurred by *Cost-Opt* is not as small as expected due to the much longer training time

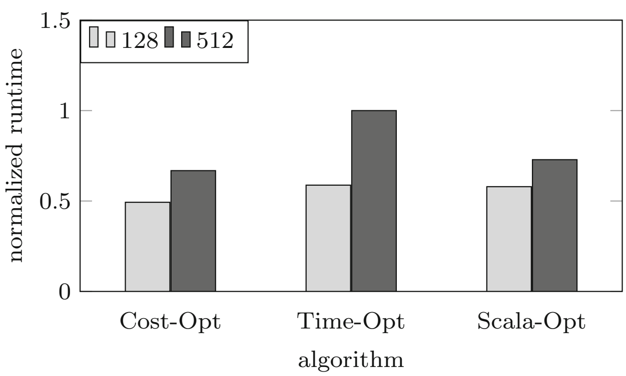

Fig. 13 Comparing execution times between all resource recommendation algorithms when using high-performance instances. We note that *Cost-Opt* and *Scala-Opt* have quite similar performance with various batch sizes, while *Time-Opt* results in more time for the larger batch size of 512

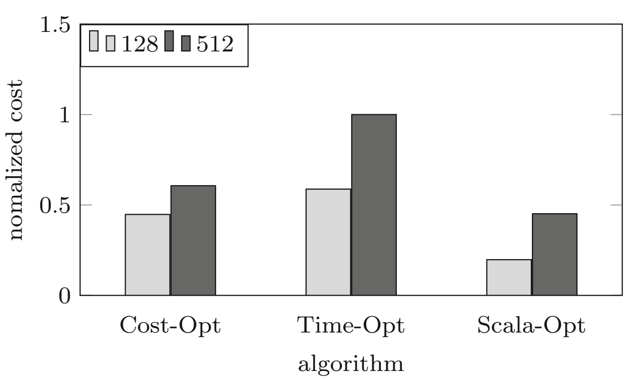

Fig. 14 Comparing resource cost incurred by each resource recommendation algorithm when using high-performance instances. We note that *Scala-Opt* outperforms *Time-Opt* by as much as 65%. The cost incurred by *Cost-Opt* is higher than that of *Scala-Opt* due to the former recommending the maximum number of workers for the training clusters

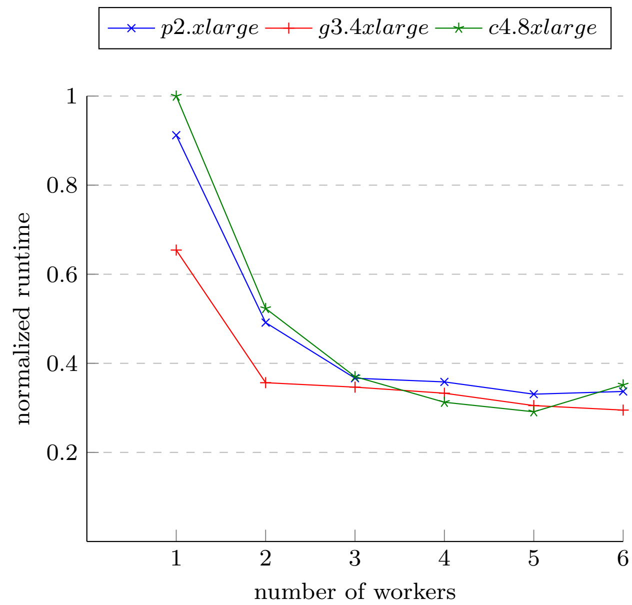

Fig. 15 Performance of high-performance instances with various number of workers, batch size of 128

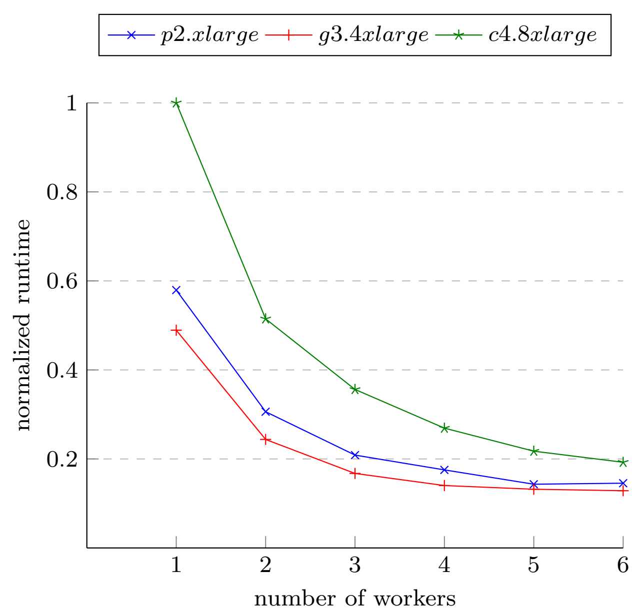

Fig. 16 Performance of high-performance instances with various number of workers, batch size of 512

proposed resource recommendation algorithms. More specifically, we use the instance type *m*4.10*xlarge* as the parameter server for all training clusters. This instance provides about 10 Gpbs in network bandwidth. We use the same high-performance instance types, i.e., *g*3.4*xlarge*, *p*2.*xlarge* and *c*4.8*xlarge*, for the workers.

Figure 17 compares the training performance produced by each recommendation algorithm. Regardless of the parameter server's capacity, we note that *Time-Opt* performs worse than the other two, mainly due to the fact that

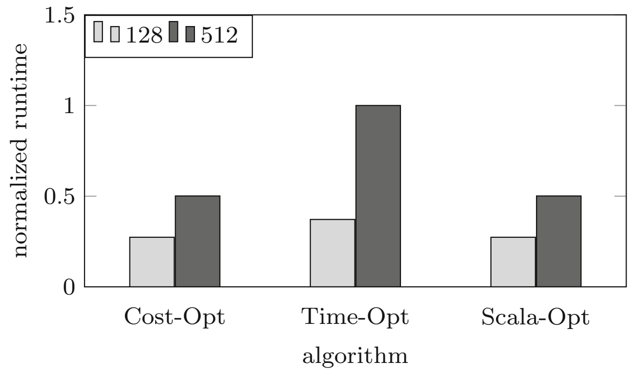

Fig. 17 Comparing execution times between all resource recommendation algorithms when using a large-capacity parameter server, the m4.10xlarge. We note that *Cost-Opt* and *Scala-Opt* have quite similar performance with various batch sizes, while *Time-Opt* results in more time for the larger batch size of 512

*c*4.8*xlarge*, although expensive, is not the best at this kind of ML model training tasks. The other thing is that *Cost-Opt* and *Scala-Opt* have almost the same level of performance. This is because they recommend the same resource type and cluster size in this case. Here, it happens that the highest performing instance type is also the cheapest one, and this explains the similarity in performance between *Cost-Opt* and *Scala-Opt*. This might not be the case all the time. In the public cloud market where new resource types are introduced and pricing adjusted quite frequently, we believe that *Scala-Opt* should be the choice for consistently recommending an appropriate cluster size and worker type (Fig. 18).

Due to the larger network capacity, adding more workers to the clusters (subjected to a pre-defined limit or budget) seems to reduce the training time more compared to the previous experiments, although the reduction get less significant as the cluster size increases. Figures 19 and 20 illustrate this effect. All the resource recommendation algorithms suggest the maximum size for the cluster.

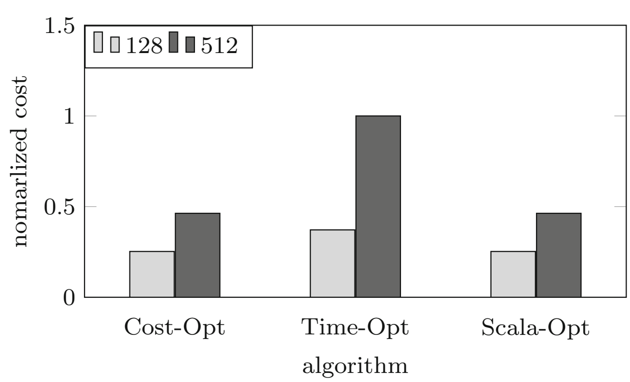

Fig. 18 Comparing resource cost incurred by each resource recommendation algorithm when using large-capacity parameter server. We note that *Scala-Opt* and *Cost-Opt* outperform *Time-Opt* significantly

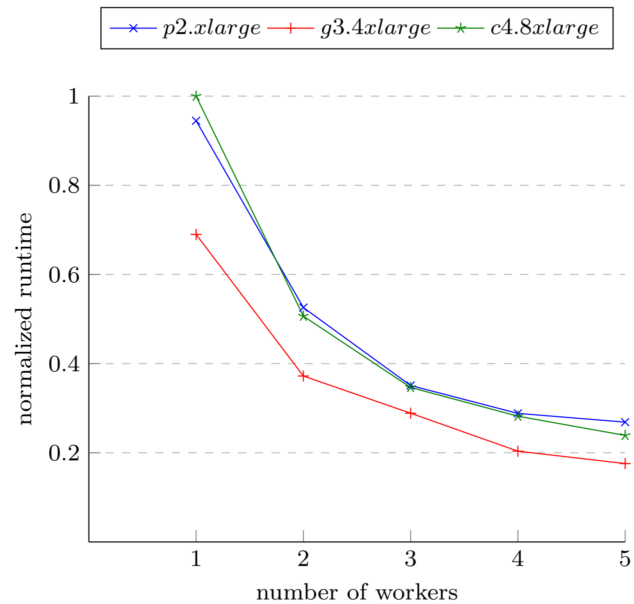

Fig. 19 Training performance when using large parameter server with various number of workers, batch size of 128

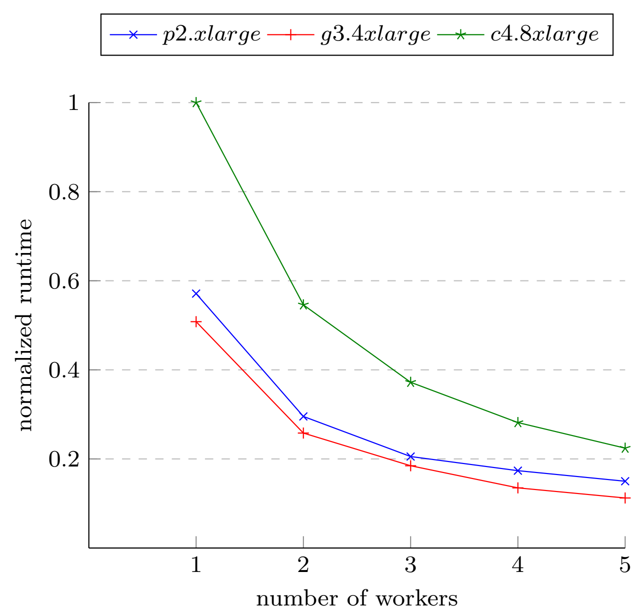

Fig. 20 Training performance when using large parameter server with various number of workers, batch size of 512

Therefore, in Fig. 18, we observe that *Cost-Opt* and *Scala-Opt* have similar cost, while *Time-Opt* incurs the most cost due to the more expensive instance type coupled with longer training time. We also note that while using a better parameter server could make it easier for selecting the right cluster size, the cost of such server would account for a significant proportion in the users' budget. In particular, the

$m4.10xlarge$ costs about 5X more than $m4.2xlarge$ which has been used in the previous experiments.

8.5 Summary

The empirical results demonstrated that the *Scala-Opt* algorithm could effectively make use of scalability properties such as network capacity of servers and bandwidth utilization of distributed ML tasks to make simple but efficient cluster configuration recommendations. We highlight two key advantages of *Scala-Opt*:

(1) In most cases, *Scala-Opt* provides similar training performance in terms of execution time compared to the other two algorithms, namely *Time-Opt* and *Cost-Opt*, but with much lower resource cost (up to 80% cost reduction). The significant savings in resource cost enable ML researchers to conduct more training for fine-tuning of models and hyper-parameters.

(2) We also observe that *Scala-Opt* consistently works well for a wide range of instance types used as workers and parameter servers. In most practical cases, it was able to select the lower cost but higher-performing resource type given the diverse options from public cloud providers. This feature is especially useful as in the current cloud computing landscape, new resource types and pricing have been introduced to the market very frequently. It is not sufficient to just rely on hardware specifications and pricing for automatic provisioning of ML clusters.

9 Conclusion

Public cloud services such as AWS EC2 provides various resource configurations with different pricing and performance levels, which make it difficult to select a suitable cluster setup to execute resource-intensive distributed ML model training tasks. In addition, popular ML frameworks such as TensorFlow or MXNet focus on programming support and model development, and leave the job of cluster configuration and deployment to end users. These issues create a gap between scalable ML and distributed computing research, which hinders the progress of ML researchers who might not be familiar with distributed system setup, or not willing to spend the time.

In this work, we have designed and developed FC^2, an easy-to-use web service which could automate the resource provisioning, configuration and execution of distributed ML training tasks. The core of our system is a set of resource-aware recommendation algorithms which can intelligently suggest appropriate cluster setups to run any ML tasks without the need to analyze complex source code, or making predictions on task running time in advance. Our proposed *Scala-Opt* algorithm instead leverages the scalability properties of a distributed ML setup to recommend cost-effective and high-performing cluster configurations. The experiments demonstrated that *Scala-Opt* could achieve similar levels of performance compared to much more expensive configurations. The cost savings produced by *Scala-Opt* could be up to 80% as demonstrated in our experiments. We are deploying the FC^2 system to serve end-user ML model training requests in our organization.

Acknowledgements The research has been supported via the Academic Research Fund (AcRF) Tier 1 Grant RG121/15.

References

1. Chan, W., Jaitly, N., Le, Q., Vinyals, O.: Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In: 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 4960–4964. IEEE (2016)
2. Li, M., Andersen, D.G., Park, J.W., Smola, A.J., Ahmed, A., Josifovski, V., Long, J., Shekita, E.J., Su, B.Y.: Scaling distributed machine learning with the parameter server. OSDI **14**, 583–598 (2014)
3. Ulanov, A., Simanovsky, A., Marwah, M.: Modeling scalability of distributed machine learning. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 1249–1254. IEEE (2017)
4. Yan, F., Ruwase, O., He, Y., Chilimbi, T.: Performance modeling and scalability optimization of distributed deep learning systems. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1355–1364. ACM (2015)
5. Amazon Machine Learning. https://aws.amazon.com/machine-learning. August 2018
6. Microsoft Azure Machine Learning Studio. https://studio.azureml.net. August 2018
7. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
8. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M.: Tensorflow: a system for large-scale machine learning. OSDI **16**, 265–283 (2016)
9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia, pp. 675–678. ACM (2014)
10. Chilimbi, T.M., Suzue, Y., Apacible, J., Kalyanaraman, K.: Project adam: building an efficient and scalable deep learning training system. OSDI **14**, 571–582 (2014)
11. Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q.V., et al.: Large scale

distributed deep networks. In: Advances in Neural Information Processing Systems, pp. 1223–1231 (2012)
12. Varol, G., Laptev, I., Schmid, C.: Long-term temporal convolutions for action recognition. IEEE Trans. Pattern Anal. Mach. Intell. **40**(6), 1510–1517 (2018)
13. Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE Trans. Pattern Anal. Mach. Intell. **40**(4), 834–848 (2018)
14. Klein, G., Kim, Y., Deng, Y., Senellart, J., Rush, A.M.: Opennmt: Open-source toolkit for neural machine translation. arXiv preprint arXiv:1701.02810 (2017)
15. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
16. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
17. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: CVPR, Vol. 1, p. 3 (2017)
18. Carreira, J., Zisserman, A.: Quo vadis, action recognition? a new model and the kinetics dataset. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4724–4733. IEEE (2017)
19. Kay, W., Carreira, J., Simonyan, K., Zhang, B., Hillier, C., Vijayanarasimhan, S., Viola, F., Green, T., Back, T., Natsev, P., et al.: The kinetics human action video dataset. arXiv preprint arXiv:1705.06950 (2017)
20. Wang, W., Chen, G., Chen, H., Dinh, T.T.A., Gao, J., Ooi, B.C., Tan, K.L., Wang, S., Zhang, M.: Deep learning at scale and at ease. ACM Trans. Multimed. Comput. Commun. Appl. (TOMM) **12**(4s), 69 (2016)
21. Xing, E.P., Ho, Q., Dai, W., Kim, J.K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., Yu, Y.: Petuum: a new platform for distributed machine learning on big data. IEEE Trans. Big Data **1**(2), 49–67 (2015)
22. Watcharapichat, P., Morales, V.L., Fernandez, R.C., Pietzuch, P.: Ako: Decentralised deep learning with partial gradient exchange. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, pp. 84–97. ACM (2016)
23. Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., Recht, B.: Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 445–451. ACM (2017)
24. Duong, T.N.B., Zhong, J., Cai, W., Li, Z., Zhou, S.: Ra2: Predicting simulation execution time for cloud-based design space explorations. In: Proceedings of the 20th International Symposium on Distributed Simulation and Real-Time Applications, pp. 120–127. IEEE Press (2016)
25. Yan, F., Ruwase, O., He, Y., Smirni, E.: Serf: efficient scheduling for fast deep neural network serving via judicious parallelism. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 300–311. IEEE (2016)
26. Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799 (2018)
27. Oyama, Y., Nomura, A., Sato, I., Nishimura, H., Tamatsu, Y., Matsuoka, S.: Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 66–75. IEEE (2016)
28. Li, A., Zong, X., Kandula, S., Yang, X., Zhang, M.: Cloudprophet: towards application performance prediction in cloud. In: ACM SIGCOMM Computer Communication Review, vol. 41, pp. 426–427. ACM (2011)
29. Cunha, M., Mendonça, N., Sampaio, A.: Cloud crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds. Concurr. Comput. Pract. Exp. **29**(1), e3825 (2017)
30. Li, H.W., Wu, Y.S., Chen, Y.Y., Wang, C.M., Huang, Y.N.: Application execution time prediction for effective cpu provisioning in virtualization environment. IEEE Trans. Parallel Distrib. Syst. **28**(11), 3074–3088 (2017)
31. Evangelinou, A., Ciavotta, M., Ardagna, D., Kopaneli, A., Kousiouris, G., Varvarigou, T.: Enterprise applications cloud rightsizing through a joint benchmarking and optimization approach. Future Gener. Comput. Syst. **78**, 102–114 (2018)
32. Cui, H., Cipar, J., Ho, Q., Kim, J.K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G.R., Gibbons, P.B., et al.: Exploiting bounded staleness to speed up big data analytics. In: USENIX Annual Technical Conference, pp. 37–48 (2014)
33. Sun, P., Wen, Y., Duong, T.N.B., Yan, S.: Timed dataflow: Reducing communication overhead for distributed machine learning systems. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), pp. 1110–1117. IEEE (2016)
34. Sun, P., Wen, Y., Ta, N.B.D., Yan, S.: Towards distributed machine learning in shared clusters: a dynamically-partitioned approach. In: 2017 IEEE International Conference on Smart Computing (SMARTCOMP), pp. 1–6. IEEE (2017)
35. Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., Li, H.: Terngrad: Ternary gradients to reduce communication in distributed deep learning. In: Advances in Neural Information Processing Systems, pp. 1509–1519 (2017)
36. Seide, F., Fu, H., Droppo, J., Li, G., Yu, D.: 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In: Fifteenth Annual Conference of the International Speech Communication Association (2014)
37. Lin, Y., Han, S., Mao, H., Wang, Y., Dally, W.J.: Deep gradient compression: reducing the communication bandwidth for distributed training. arXiv preprint arXiv:1712.01887 (2017)
38. Peng, Y., Bao, Y., Chen, Y., Wu, C., Guo, C.: Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: Proceedings of the Thirteenth EuroSys Conference. ACM (2018)
39. Google Cloud AI. https://cloud.google.com/products/ai. August 2018
40. BigML. https://bigml.com. August 2018
41. Amazon Deep Learning AMIs. https://aws.amazon.com/machine-learning/amis. August 2018
42. AWS CloudFormation. https://aws.amazon.com/cloudformation. August 2018
43. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
44. Li, M., Andersen, D.G., Smola, A.J., Yu, K.: Communication efficient distributed machine learning with the parameter server. In: Advances in Neural Information Processing Systems, pp. 19–27 (2014)

45. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Technical Report, University of Toronto (2009)

Ta Nguyen Binh Duong is currently a regular faculty (Lecturer) in the School of Computer Science and Engineering (SCSE), Nanyang Technological University (NTU), Singapore. He obtained his PhD in Computer Science from NTU Singapore. Previously, he was a Research Scientist with A*STAR Institute of High Performance Computing, and a Research Fellow with SCSE, NTU and University College Cork, Ireland. His main areas of expertise include distributed computing, machine learning, distributed simulations, and computer networking.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.