

Performance Models for Distributed Deep Learning Training Jobs on Ray

Federica Filippini
IBM Research & Politecnico di Milano
Milano, Italy
federica.filippini@polimi.it

Boris Lublinsky
IBM Research
Dublin, Ireland
blublinsky@ibm.com

Maximilien de Bayser
IBM Research
Rio De Janeiro, Brazil
mbayser@br.ibm.com

Danilo Ardagna
Politecnico di Milano
Milano, Italy
danilo.ardagna@polimi.it

Abstract—Deep Learning applications are pervasive today, and efficient strategies are designed to reduce the computational time and resource demand of the training process. The Distributed Deep Learning (DDL) paradigm yields a significant speed-up by partitioning the training into multiple, parallel tasks. The Ray framework supports DDL applications exploiting data parallelism by enhancing the scalability with minimal user effort. This work aims at evaluating the performance of DDL training applications, by profiling their execution on a Ray cluster and developing Machine Learning-based models to predict the training time when changing the dataset size, the number of parallel workers and the amount of computational resources. Such performance-prediction models are crucial to forecast computational resources usage and costs in Cloud environments. Experimental results prove that our models achieve average prediction errors between 3 and 15% for both interpolation and extrapolation, thus demonstrating their applicability to unforeseen scenarios.

Index Terms—Distributed training, Performance models, Ray

I. INTRODUCTION

Deep Learning (DL) applications are pervasive today, and applied to tackle a variety of complex problems in heterogeneous fields ranging from, e.g., image classification [1] to speech recognition [2]. The complexity of the training process dictates the need to devise efficient strategies to reduce the computational time and resource demand. The Distributed Deep Learning (DDL) paradigm [3] moves in this direction, aiming to speed-up the learning process by splitting the training procedure in multiple tasks executed in parallel. DDL is implemented according to two main approaches: *data parallelism* [4], where the training process is partitioned by dividing the input data in non-overlapping samples, and *model parallelism* [5], where the neural network is partitioned by layer while the input dataset is unique. Both approaches, albeit yielding a significant speed-up when the input data size or the number of neural network layers are large, introduce communication overhead due to the need of synchronizing partial results during the training. Nowadays, several frameworks are available to support DDL training applications, particularly data parallelism (e.g., PyTorch DistributedDataParallel [6], TensorFlow tf.distribute.Strategy API [7], Horovod [8]). They offer different levels of flexibility and scalability, but usually require the developer to specify the correct architecture description for the cluster where the training runs. This allows

to efficiently design the parallelization schema, but is often a complex task. Ray [9], integrated with popular Machine Learning (ML) frameworks as PyTorch [10] and TensorFlow [11], is becoming popular to support the deployment and scaling of both general-purpose and ML-based workflows. DDL applications can easily run on Ray-based clusters, exploiting Ray modules to enhance the scaling with minimal user effort.

This work aims at evaluating the performance of DDL training applications developed and executed on a cloud Ray cluster. Starting from suitable profiling data, Machine Learning-based performance models are developed to predict the execution times of various tasks executed in this environment, varying the dataset size, the number of parallel workers and the amount of resources they receive. As discussed in other literature proposals (see, e.g., [12]), this is crucial to forecast also the computational resources usage and the corresponding costs related to the DDL training in cloud environments. To the best of our knowledge, this is the first attempt to develop performance models for DL applications based on Ray. Experimental results prove that the average errors obtained for both interpolation and extrapolation predictions are between 3 and 15%, making the models suitable for practical applications.

The rest of the paper is organized as follows: Section II relates our work to the literature, Section III introduces the problem we tackled and the main frameworks we considered, and Section IV describes the experimental setup. Finally, the results we obtained are presented in Section V, while conclusions are drawn in Section VI.

II. RELATED WORK

To the best of our knowledge, this is one of the first attempts to analyze the performance of Distributed Deep Learning (DDL) applications based on Ray and to develop Machine Learning (ML)-based models for execution time predictions. Thus, we overview some recent works in related technological fields, such as ML-based performance modelling for Big Data and Deep Learning applications.

The authors in [13] proposed supervised ML models to predict the performance of applications executed on Apache Spark [14], a well-known framework for big data analysis, achieving very good results with respect to the Ernest approach, developed by the Spark inventors [15]. Spark applications were considered also in [16], which proposes a ML-based

prediction platform for SQL queries and ML applications exploiting stage-related features and a previous knowledge of the application profile. This platform is implemented as an additional Spark component, and it is designed to predict the execution time of both individual and complex operators. Moreover, [17] couples ML approaches and the queuing theory to predict the performance of big data applications running on clouds. Initial analytical models provide a baseline to train the ML algorithm, which is validated, as in our work, testing both its interpolation and extrapolation capabilities.

Considering Convolutional Neural Network (CNN) applications, [12] develops linear regression models to predict the training times on GPU-as-a-service systems, exploiting features related both to the CNN properties and the hardware characteristics of the GPUs where the jobs are deployed. On the other hand, [18] focuses on linear regression models for CNN applications executed on edge devices, which are then exploited in determining the optimal allocation minimizing the latency between the data-gathering phase and the subsequent decision-making one. Similarly, linear models are used in the Schedulix framework [19] to estimate execution latency of serverless applications in the public cloud, deployed according to the Function as a Service paradigm.

On different, but related topics, the authors of [20] compare and assess the good prediction capabilities of popular ML techniques applied to a workload analysis on HTTP servers. A framework leveraging ML models as logistic regression and Support Vector Machine is proposed in [21] to predict failures of legacy network nodes in the context of hybrid Software-Defined Networking architectures. Finally, [22] exploit heterogeneous ML models together with anomaly detection to properly configure a cloud-based Internet of Things device manager, meeting Quality of Service constraints.

III. PROBLEM DESCRIPTION

This section introduces the main components considered in this work, i.e., the PyTorch module used to implement the Distributed Deep Learning training applications under study (Section III-A) and the Ray framework (Section III-B).

A. The DistributedDataParallel Module

The PyTorch DistributedDataParallel (DDP) module [6] supports data parallelism in Deep Learning model training, splitting the input across the specified nodes. The model to be trained is replicated on all processes involved in the computation, which may be executed on a single or multiple machines. Each process receives and tackles a portion of the input data, while the gradients produced at each step are gathered through collective communications and averaged across all nodes as shown in Figure 1.

When initializing a DDP object, the problem configuration and model state are broadcasted from the process with rank 0, so that all the model replicas start from the same initial point. Each process owns a Reducer, which has the goal of synchronizing gradients during the backward pass. As shown in Figure 1, gradients are organized and reduced in

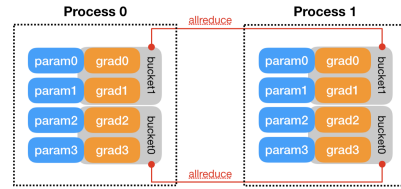


Fig. 1: DistributedDataParallel reduction process

buckets to improve communication efficiency. Starting from the same initial state and averaging gradients at every iteration guarantees that all model replicas on the DDP processes stay synchronized during the training.

B. The Ray Framework

Ray [9] has been developed to easily and transparently support the deployment and scaling of ML-based and general Python workflows on heterogeneous clusters. It is a unified framework that includes three main layers, designed to effectively manage different stages and aspects of the applications development. These are: (i) the *Ray cluster*, which includes a set of worker nodes and can possibly autoscale according to the applications resource requirements, (ii) the *Ray Core*, which enables to scale generic Python applications, and (iii) the *Ray AI Runtime (AIR) Toolkit*, which is built on top of Ray Core and is specifically devoted to ML and Artificial Intelligence (AI) workflows.

Tasks submitted to a Ray cluster require a specific amount of computing resources. The requests are managed by a distributed scheduler, through a component, denoted as *raylet*, deployed on each node in the cluster. Each raylet tracks the resources local to its node and receives information about the availability in other nodes. When a resource request arrives, the raylet can grant the resources locally, if available, scheduling the corresponding task for execution, or it can provide the address of a remote raylet at which the resource request should be offloaded [23]. The process is performed automatically and, if the required resources are not available in any cluster node, an additional component named *autoscaler* may provision additional workers according to the demand.

Ray AIR [24], [25] has been developed to support AI and ML applications. Its main goal is to enable and simplify ML workflows scaling, providing a unified ecosystem for data management, training, hyperparameter tuning and inference. In particular, the Ray Train module [26] enables scaling ML training workflows based on the most popular frameworks (e.g., PyTorch [10] and TensorFlow [11]), and it is integrated with other Ray libraries for, e.g., hyperparameter tuning and inference, to help users in developing and managing the entire workflow. Specific classes of trainers, checkpointers and predictors are implemented to integrate with the different ML and DL frameworks.

The experiments that will be described in the next sections are based on the definition of a *TorchTrainer*, which is configured by providing, on one hand, test parameters as the batch size, learning rate and number of epochs, and, on the

other hand, parameters related to the DDL process scaling. Specifically, a *ScalingConfig* object is initialized when creating the trainer, and used to specify the number of required workers and the number of CPUs or other resource types to be assigned to each worker. A placement group is created internally for each worker, so that the training can start only when all the required resources are available in the cluster. By default, the strategy used when creating each placement group is PACK, which means that all resources bundles (i.e., the worker CPUs, GPUs and any required extra resource) are created, if possible, on the same worker node.

The trainer is serialized and copied to a remote Ray actor when the *fit()* method is called. This starts the execution of the data preprocessing step and the training loop. The method returns an object that stores metrics from the training run, as well as any checkpoints saved during the process.

IV. EXPERIMENTAL SETUP

This section describes the experiments that were executed to evaluate the performance of DDL training jobs. Specifically, Section IV-A provides details about the DDL applications and the considered datasets, while Section IV-B presents the infrastructure where the experiments were executed. Finally, the methodology exploited to collect training data and developing the performance models is detailed in Section IV-C.

A. Deep Learning Models and Dataset

The training applications considered in the following consist of two CNN models for image classification, implemented relying on the PyTorch DDP module [6]. The models are characterized by neural networks of different sizes and structure. In particular, the first one is a very small and simple network, denoted in the following as *small model*, characterized by around 400k parameters, while the second one is a slightly larger network, denoted as *large model*, including almost 3M parameters. Considering CNNs that are considerably smaller than some traditional models as, e.g., AlexNet, allowed us to obtain a lower bound on the scalability properties offered by the Ray framework. Indeed, the overheads incurred during the training due to the collective communications needed to synchronize gradients and buffers are more significant than what we would obtain with larger models. Moreover, by considering these networks we could evaluate both the CPU and GPU scalability, without encountering issues with the resource memory and too much performance degradation of the CPU training with respect to the GPU. Finally, since in this context the profiling data are more noisy than for the long training of larger models for the above mentioned overhead, this is a challenging scenario for evaluating the accuracy that can be achieved through ML-based performance models.

We considered the MNIST data set [27], consisting of 60k, 28 × 28 black and white images of handwritten digits, and a test set of 10k images. We built four different training sets, denoted in the following as *MNIST*, *MNISTx2*, *MNISTx4* and *MNISTx8* and characterized by 60k, 120k, 240k, and 480k images, respectively, by replicating the base MNIST data.

Moreover, we considered three different batch sizes (of 64, 128, and 256 images).

B. Infrastructure

The experiments were executed on a Ray cluster hosted on AWS, including one head node and multiple worker nodes provisioned on-demand by the Ray autoscaler as described in Section III-B. The head node is based on a Virtual Machine (VM) instance of type *m5.large*, i.e., a general-purpose instance featuring 2 virtual CPUs (vCPUs) and 8 GiB of memory [28]. The worker nodes are configured as follows:

cpu_8 resource pool: up to 4 nodes, exploiting a *m4.2xlarge* VM (8 vCPUs and 32 GiB of memory).
cpu_16 resource pool: up to 4 nodes, exploiting a *m4.4xlarge* VM (16 vCPUs and 64 GiB of memory).
cpu_72 resource pool: up to 16 nodes, featuring a *c5n.18xlarge* VM (72 vCPUs and 192 GiB of memory).
gpu_4 resource pool: one node, exploiting a *p3.8xlarge* VM (4 NVIDIA Tesla V100 GPUs, 32 vCPUs and 244 GiB of memory) and supporting NVLink for peer-to-peer GPU communication.

Note that, since *raylet* needs some resources to run, 2 vCPUs on each worker node are always reserved to it, so that the maximum number of available CPUs is considered to be 4, 14 and 70 for the three CPU-only resource pools, respectively.

C. Methodology

This section details the methodology followed to collect data related to the DDL model training and to process these data in order to build the performance models that will be described in the following.

1) *Data Collection*: The DDL training tasks exploiting the models and dataset described in Section IV-A were executed multiple times varying the dimension of the training data and the batch size, the number of parallel workers, and the resources assigned to each worker. In the first set of tests, we focused on the *cpu_72* resource pool, varying the number of parallel workers and CPUs per worker. In the second set, we investigated the impact of different CPU types by varying the chosen resource pool, fixing to 2 or 4 the number of parallel workers and assigning to each worker all CPUs available in a node. Finally, a third set of tests was performed by running experiments on the GPU-accelerated node, exploiting 2, 3 or 4 parallel workers with one dedicated GPU each. The grid of all configuration parameters is reported in Table I. The experiments on the *cpu_72* resource pool, which were later used to build our performance models, were repeated twice in order to enlarge the set of collected data and obtain robust results with respect to possible variations due to unpredictable overheads in the cluster. The reported numbers of CPUs and GPUs in Table I are intended per-worker.

For each experiment, we collected the execution times of different stages, as shown in Figure 2:

train_epoch and *validate_epoch*: the time required by each worker to run the training and validation, respectively, of a single epoch. If, e.g., 3 epochs are run on 8

TABLE I: Configuration Parameters

CPU-only experiments on the <i>cpu_72</i> resource pool				
Model	#Images	Batch Size	#Workers	#CPUs
small	60k,120k,240k	64,128,256	2,4,8,16,32,64	1
small	480k	64,128,256	2,4,8,16	1
small	60k,120k,240k,480k	64,128,256	2,4,8,16,32	2
small	60k,120k,240k,480k	64,128,256	2,4,8,16	4
large	60k,120k,240k,480k	64,128,256	2,4,8,16,32,64	2
large	60k,120k,240k,480k	64,128,256	2,4,8,16,32	4
large	60k,120k,240k,480k	64,128,256	2,4,8,16	8

CPU-only experiments varying resource pool				
Model	#Images	Batch Size	#Workers	pool
large	60k,240k	64,128,256	2,4	<i>cpu_8</i>
large	60k,240k	64,128,256	2,4	<i>cpu_16</i>
large	60k,240k	64,128,256	2,4	<i>cpu_72</i>

GPU-accelerated experiments				
Model	#Images	Batch Size	#Workers	#GPUs
large	60k,120k,240k,480k	64,128,256	2,3,4	1

workers, this results in 24 data points for training and 24 for validation.

training_loop: the time required by each worker to run the complete training loop, including training and validation for all the specified epochs.

fit: the time required to run the *trainer.fit* method, including the dataset loading on each worker, the data and model preparation, and the training loop. Note that this includes also the time required to create the placement groups for all workers, and thus the time the Ray Autoscaler needs to provision new nodes if needed.

train: the time required to run the external training function, including the initialization of the trainer with the proper configuration and scaling parameters and the *trainer.fit* method.

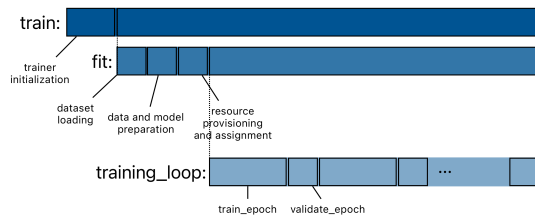


Fig. 2: Training stages

2) *Performance Models Building*: ML-based performance models were built through the *aMLLibrary*, an open-source tool for the automatic generation of regression models proposed in [29]. The models were trained and validated considering the execution times collected on the *cpu_72* resource pool, since this set of tests is the one that incorporates the highest variability. The set of ML methods and corresponding hyperparameters, the list of basic features we considered and the validation techniques we exploited are reported in Table II. The best set of hyperparameters was selected in all scenarios considering K-Fold validation technique, with number of folds equal to 5. The basic features reported in the table were chosen as in other literature proposals [13], [15]. The list was extended, exploiting the feature augmentation technique implemented by *aMLLibrary*, by considering the inverse and logarithm of n , j , *num_CPUs* and *num_images*, and, when interpolating and extrapolating on the number of images in the dataset, by computing products up to the second degree. In particular, the logarithm of the number of used resources

is particularly significant in parallel applications, since it effectively captures the behavior of the weighted average computation, usually based on reduce-like operations [13] (see Section III-A). Finally, the choice of the best model was performed considering all the three validation techniques offered by *aMLLibrary*, i.e., HoldOut, Interpolation and Extrapolation, defining the training and test sets as illustrated in Table II. In particular, HoldOut randomly extracts a given percentage of observations to be used as validation set, while Interpolation and Extrapolation are specifically tailored to build models able to predict values in areas of the feature space that were not sufficiently explored. It is relevant to note that, while the interpolation capabilities of ML models are widely known from the literature [30], good extrapolation results are generally harder to achieve. However, ML models that can accurately generalize to unforeseen scenarios have significant practical applications, e.g., the capacity planning of the cluster.

TABLE II: Machine Learning Models Configuration

Model Parameters		
Model Name	Hyperparameter Name	Values
XGBoost	min_child_weight	1;3
	gamma	0;1
	n_estimators	50;150
	learning_rate	0.01;0.05;0.1
	max_depth	1;5;10
Ridge Regression	alpha	0.01;0.1;1
Decision Tree	criterion	mse, friedman_mse, mae
	max_depth	3;5;10;20
	max_features	auto, sqrt, log2
	min_samples_split	0.01;0.1;0.2;0.5
	min_samples_leaf	0.01;0.05;0.1;0.2;0.3
Random Forest	n_estimators	10;50;200
	criterion	mae
	max_depth	3;10
	max_features	auto
	min_samples_split	0.1;0.5
	min_samples_leaf	1;4

Basic Features	
Feature Name	Description
<i>num_images</i>	Number of images in the training dataset
n	Number of parallel workers
j	Number of CPUs per worker
<i>num_CPUs</i>	Total number of used CPUs (i.e., $n \cdot j$)
b	Batch size
e	Number of epochs
l	Learning rate

Validation Techniques		
Name	Variable	Training and Test set
HoldOut	-	leave-out ratio: 0.2
Interpolation	<i>num_CPUs</i>	training: $f4;8;32;128g$; test: $f16;64g$
Interpolation	<i>num_images</i>	training: $f60k;240k;480kg$; test: $f120kg$
Extrapolation	<i>num_CPUs</i>	training: 32; test: > 32
Extrapolation	<i>num_images</i>	training: 240k; test: > 240k

V. EXPERIMENTAL RESULTS

This section reports the results achieved with all experiments described in Section IV. In particular, Section V-A is focused on the analysis of all data collected as reported in Section IV-C1, while Section V-B discusses the accuracy of the performance models built as explained in Section IV-C2.

A. Execution Times Analysis

The main goal of the analysis reported in this section is to discuss the impact on the model training times due to varying Ray configuration parameters. The most significant impact on the execution times is due to increasing/decreasing the number of workers. This is, however, considerably different when training the *small* or *large* neural network model described in Section IV-A. Indeed, very simple networks as the *small* model benefit only from increasing the number of parallel workers up to 8, since then the communication overheads

become too large. Due to space limits, we will not discuss results related to the impact of exploiting different CPU-only based resource pools. Instead, Figure 3 reports the execution times when exploiting GPUs. Due to the limited size of the GPU-accelerated node, only instances with 2, 3 and 4 parallel workers could be considered. However, the execution times are always significantly lower than in the CPU-only scenario, yielding a speed-up close to 10 with respect to the results obtained on the *cpu_72* resource pool.

Figure 4 shows the distribution of the average execution times of the *train*, *fit*, *training_loop* and *train_epoch* events considering the *large* model, varying the configuration parameters. From the first row, we can observe that the execution times always drop significantly when increasing the number of workers n up to 16. Instead (third row), they increase with the number of images (i.e., the dataset size) in all scenarios. Moreover, changing the batch size (last row) does not significantly affect the execution times of the training epochs or the training loop, but it affects the overall execution time of the experiment (Figures 4a and 4b), since it is related to the data loading part. Since the total number of CPUs and the number of images are the parameters that mostly affect the execution times, these were considered for extrapolation and interpolation when building the performance prediction models; the results are shown in the next section.

B. Performance Models

This section reports the prediction results obtained by the different performance models (see Section IV-C2). The Mean Absolute Percentage Error (MAPE)¹ obtained for all events with different validation techniques is reported in Table III. In particular, the HoldOut validation technique provides a lower bound for the models accuracy, since data are randomly split in training and test set. We can note that the MAPE is very low (below 2%) for the *training_loop* and *train_epoch* events, while larger but still below 7% for the *train* and *fit* events. This is expected, since *train* and *fit* include the overheads due to the data loading, trainer initialization and resource provisioning.

As already mentioned (see Section IV-C2), interpolation and extrapolation results are more relevant in practice, since they investigate the generalization capabilities of the obtained models. The MAPE achieved when interpolating on the total number of CPUs is below 11% for all events, and the percentage error is always below 35%. This makes the models suitable for predictions in real-life scenarios, since MAPEs around 30% are usually considered as acceptable from the literature [31]. When considering the extrapolation on the total number of CPUs, the MAPE is slightly larger than in the previous cases, which is reasonable since extrapolation is a significantly more complex task, but still lower than 16%. For the extrapolation on the dataset size, augmenting the features by considering the products up to the second degree,

¹MAPE = $\frac{100}{N} \sum_{k=1}^N \frac{y_k - \hat{y}_k}{y_k}$, where N is the total number of observations, y_k denotes the real value of observation k and \hat{y}_k denotes the corresponding prediction

as mentioned in Section IV-A, allows to obtain a MAPE below 4% both for the *training_loop* and *train_epoch* events.

TABLE III: Performance Models MAPE

Validation Technique	Event			
	<i>train</i>	<i>fit</i>	<i>training_loop</i>	<i>train_epoch</i>
HoldOut	6.41%	6.71%	1.07%	1.89%
Interpolation (num_CPUs $\geq 16:64$)	5.44%	6.89%	9.68%	10.73%
Extrapolation (num_CPUs > 32)	14.54%	15.92%	12.79%	13.47%
Extrapolation (num_images $> 240K$)	-	-	3.85%	3.76%

VI. CONCLUSIONS

This work presents one of the first attempts to investigate the performance of DDL training applications developed exploiting the Ray framework. The results of an extensive experimental campaign collecting the execution times of different training tasks were used to develop ML-based models for performance predictions, achieving a MAPE below 15% in all the considered scenarios. Future developments may include the extension of the ML-based performance models to analyze the impact of exploiting GPUs for the training, and to better investigate the performance variations due to concurrent tasks execution on the same worker nodes.

REFERENCES

- [1] G. Huang, Z. Liu *et al.*, “Densely connected convolutional networks,” in *IEEE CVPR*, 2017, pp. 2261–2269.
- [2] T. G. Dietterich, “Ensemble methods in machine learning,” in *Multiple Classifier Systems*. Springer Berlin Heidelberg, 2000, pp. 1–15.
- [3] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surv.*, vol. 52, no. 4, 2019.
- [4] C. Shallue, J. Lee *et al.*, “Measuring the effects of data parallelism on neural network training,” *JMLR*, 2018.
- [5] B. M. Forrest, D. Roweth *et al.*, “Implementing neural network models on parallel computers,” *Comput. J.*, vol. 30, no. 5, p. 413–419, 1987.
- [6] Pytorch distributeddataparallel module. [Online]. Available: <https://pytorch.org/docs/stable/notes/ddp.html>
- [7] Tensorflow distributed training. [Online]. Available: https://www.tensorflow.org/guide/distributed_training
- [8] Horovod. [Online]. Available: <https://horovod.readthedocs.io/en/stable/>
- [9] Ray. [Online]. Available: <https://docs.ray.io/en/releases-2.0.0/index.html>
- [10] Pytorch. [Online]. Available: <https://pytorch.org>
- [11] Tensorflow. [Online]. Available: <https://www.tensorflow.org>
- [12] M. Lattuada, E. Gianniti *et al.*, “Performance prediction of deep learning applications training in GPU as a service systems,” *Clust. Comput.*, vol. 25, no. 2, pp. 1279–1302, 2022.
- [13] A. Maros, F. Murai *et al.*, “Machine learning for performance prediction of spark cloud applications,” in *IEEE CLOUD*, E. Bertino, C. K. Chang *et al.*, Eds., 2019, pp. 99–106.
- [14] Apache spark. [Online]. Available: <https://spark.apache.org>
- [15] S. Venkataraman, Z. Yang *et al.*, “Ernest: Efficient performance prediction for Large-Scale advanced analytics,” in *USENIX NSDI 16*, 2016, pp. 363–378.
- [16] S. Mustafa, I. Elghandour *et al.*, “A machine learning approach for predicting execution time of spark jobs,” *Alexandria Engineering J.*, vol. 57, no. 4, pp. 3767–3778, 2018.
- [17] E. Ataie, A. Evangelinou *et al.*, “A hybrid machine learning approach for performance modeling of cloud-based big data applications,” *Comput. J.*, vol. 65, no. 12, pp. 3123–3140, 2022.
- [18] S. Disabato, M. Roveri *et al.*, “Distributed deep convolutional neural networks for the internet-of-things,” *IEEE Trans. on Computers*, vol. 70, no. 08, pp. 1239–1252, 2021.
- [19] A. Das, A. Leaf *et al.*, “Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications,” in *IEEE CLOUD*, 2020, pp. 609–618.
- [20] D. F. Kirchoff, M. Xavier *et al.*, “A preliminary study of machine learning workload prediction techniques for cloud applications,” in *EMPDP*, 2019, pp. 222–227.

