electronics

MDPI

Article

Performance Prediction for Convolutional Neural Network on Spark Cluster

Rohyoung Myung and Heonchang Yu *

Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea; mry1811@korea.ac.kr
* Correspondence: yuhc@korea.ac.kr; Tel.: +82-2-3290-2392

Received: 22 July 2020; Accepted: 15 August 2020; Published: 19 August 2020

Abstract: Applications with large-scale data are processed on a distributed system, such as Spark, as they are data- and computation-intensive. Predicting the performance of such applications is difficult, because they are influenced by various aspects of configurations from the distributed framework level to the application level. In this paper, we propose a completion time prediction model based on machine learning for the representative deep learning model convolutional neural network (CNN) by analyzing the effects of data, task, and resource characteristics on performance when executing the model in Spark cluster. To reduce the time utilized in collecting the data for training the model, we consider the causal relationship between the model features and the completion time based on Spark CNN's distributed data-parallel model. The model features include the configurations of the Data Center OS Mesos environment, configurations of Apache Spark, and configurations of the CNN model. By applying the proposed model to famous CNN implementations, we achieved 99.98% prediction accuracy about estimating the job completion time. In addition to the downscale search area for the model features, we leverage extrapolation, which significantly reduces the model build time at most to 89% with even better prediction accuracy in comparison to the actual work.

Keywords: convolutional neural network; feature engineering; machine learning; performance prediction

1. Introduction

The convolutional neural network (CNN) models have received considerable attention from industries and research areas in recent years, as they can achieve state-of-the-art accuracy for essential artificial intelligence-based services, such as object recognition [1,2], image classification [3,4], and signal processing [5,6]. It is attributed to the fact that the CNN's deep structure enables hierarchical learning. However, it is also important that the CNN models learn an extensive amount of data. As a result, various CNN models are operated in clusters rather than on a single machine, because they require large amounts of computation for achieving the prediction accuracy.

Cluster environments, such as Apache Mesos [7] and Hadoop Yarn [8], have attracted attention, as they provide an environment for developing and running large-scale CNN applications on clustered resources with a general distributed programming framework. These platforms enable the users to flexibly take advantage of resources according to the specific requirements of the applications. Therefore, these platforms provide a suitable processing environment for CNN's large-scale input data processing and highly decentralized processing model.

For these reasons, the cluster environment that enables flexible resource configurations is considered as an alternative to the traditional local server architecture during the implementation and operation of CNN models. The type of service often depends on distributed programming platforms, such as Apache Spark [9]. Spark is a fast and universal engine, which can be a good indicator of future large-scale distributed engines, and its usage has been steadily increasing in recent years. Spark

facilitates the implementation of various applications from machine learning algorithms to modern deep learning models.

In a cluster environment, the cost of using the resources depends on the volume of resources and the time spent on their usage. As the volume of resources and duration of usage increase, the cost also increases. Therefore, an efficient configuration of the resources according to the service level agreement is required. The user must determine the total resources to be used in combination with the detailed options suitable for user requirements, such as the total number of cores, memory size, disk capacity, and number of processors. These choices must be carefully considered, as they can significantly affect the completion time of the applications. Estimating the performance of an application in a given resource configuration is important to support an efficient resource usage and management.

In this study, considering the size of the input data, resource configuration, volume of the resource cluster, and Spark configuration, we created a performance prediction model for CNN models [10–12] on Spark to accurately estimate the job completion time of the applications. The existing performance prediction models include analytical performance models [13–17], which are based on the analysis of the general process of CNN models, and machine learning-based performance models [18–22], which select the features that are supposed to be related to the completion time. The latter collects the sample data from tedious experiments and applies the results to the machine learning techniques.

As Spark handles applications through directed acyclic graphs (DAGs) for resilient distributed dataset (RDD) [23], the traditional analytical performance models estimate the processing time in a top-down fashion about DAGs. The DAGs are comprised of ShuffleMapStages and ResultStage. ShuffleMapStages perform a series of independent parallel tasks. They also handle data exchange between executors. In detail, ShuffleMapStage is comprised of MapPartitionTasks and a ShuffleTask. MapPartitionTasks independently perform a series of tasks on a partition, which is a subset of the entire data. And a ShuffleTask delivers the results of the tasks to the next level of executors. After the last task of ShuffleMapStages, ResultStage returns the processing results to the Spark driver. As the same Spark application creates a DAG with the same structure, the completion time of the application depends on the size of the data, platform settings, and resource configuration [24]. In addition, depending on the combination of hardware specification and a cluster platform, the completion time of the application is different even if the same platform setting and resource configuration are used. Therefore, unless the environment is precisely the same, it is impossible to accurately predict the performance of CNN on Spark with existing analytical performance models.

Further, studies on machine learning-based performance models [18–22] perform the applications multiple times to collect sample data, select features that are supposed to be related to the performance, and train machine learning models to create performance models. These models achieve high prediction accuracy without any specific insight into the internal process of applications. However, collecting samples to train model parameters is time-consuming, because the applications must be executed entirely tens of times. In general, the more samples collected, the easier it is to predict the performance of the models accurately. However, such a process requires a longer time to collect data as well as train the models. The model features are chosen empirically, which poses a limitation.

In this study, we created a performance model in hybrid fashion that takes advantage of both methods. Therefore, by analyzing the general procedure of CNN models we explain how the performance-related features affect the completion time. Based on the analysis results, highly related features are selected to create a performance model with high performance prediction accuracy of 99.98%. In comparison to existing studies, our model reduces the search space as it uses only the small number of effective features. The model also demonstrate that the prediction accuracy is high even with a low sampling rate for the input data. The time to collect data and to train the model are reduced by up to 89%. The main contributions are summarized as follows:

- In a distributed parallel processing environment, CNN models are analyzed by correlating the effect of data characteristics, job characteristics, and system resource characteristics on processing time with respect to the executing procedure of Spark.

- We proposed a two-level-hierarchy performance prediction model that leverages the relationship between model features and preprocessed processing time in the first level and then predicts the actual processing time of the applications in the second level.
- In bare metal cluster with distributed parallel processing environment, our performance model achieved a prediction accuracy of 99%. We could reduce the total time required for building the performance model up to 89% compared to the existing works.

Section 2 covers the background. Section 3 analyzes the features of the performance model and its causal relationship with the performance of Spark CNN. The experimental setup for the proposed performance prediction model is introduced in Section 4. The results of various experiment scenarios of the model are described in Section 5. Related researches are covered in Section 6. Section 7 concludes the paper.

2. Background

In this section, we show how Spark manages memory and how CNN is implemented in a distributed way. The memory usage is presented to clarify how batch processing affects heap usage. Besides, we also show how distributed, and parallel CNN is processed at Spark to examine the influence of resource types other than memory.

2.1. Heap Usage Patterns of Batch Processing in Java

Memory allocation and retrieval for applications in the Java environment is managed by the garbage collector of the Java Virtual Machine (JVM). As Java considers most objects to be obsolete in a short time, the heap is divided into a young generation for short-term objects and an old generation for long-term objects [25]. When a memory allocation request is made to the JVM, it first allocates the object in the young generation area. When more than a certain percentage of the young generation is used, the minor Garbage Collection (GC) is called to remove objects that do not reach a reference from the root set [26]. The objects that survive from minor GC more than a certain number of times are promoted to the old generation area. Similarly, when more than a certain percentage of the old generation region is used, major GC is called to remove objects that do not reach references from the root set. As Spark is based on Java, the memory management of Spark relies on GC policies. The driver acts as the core of the Spark engine, while the executors act as the processor. The objects that are created during the execution of a Spark application can be divided into long-term objects created for job management and short-term objects that lose links from the root set, after the related partition is processed. The short-term objects are allocated to the heap, when partitioning starts and can be recovered from the garbage collector upon completion. As GC is performed independent of the application's processing, the heap area coexists with objects that are not required at runtime. During GC, the long-term objects remain, and after partitioning is completed, the short-term objects for the partition are reclaimable, though the objects for the partition being processed are not recoverable. Therefore, the larger the partition size, the larger the memory size required to safely process the short-term objects.

2.2. CNN Distributed Processing in Sparks

In Spark, CNN's distributed processing uses model replication-based data parallelism [27], as shown in Figure 1. The Spark driver sends a copy of the CNN model to the executors to execute. The driver partitions the input data by dividing it according to the Spark base partition size (128 MB). The driver's task scheduler assigns partitions to the executors, and the executor transforms the partitions assigned to it into a structure that conforms to the input form of the CNN model. The executors process the partitions according to the CNN model and return the results to the driver. Each executor has a copy of the model, and therefore, the partitions can be parallelized independently. The Spark driver then

completes the process by iterating over a replica of the CNN model to the executors. The performance studies on model-parallel based CNNs in Spark are continuing and will be addressed in future studies.

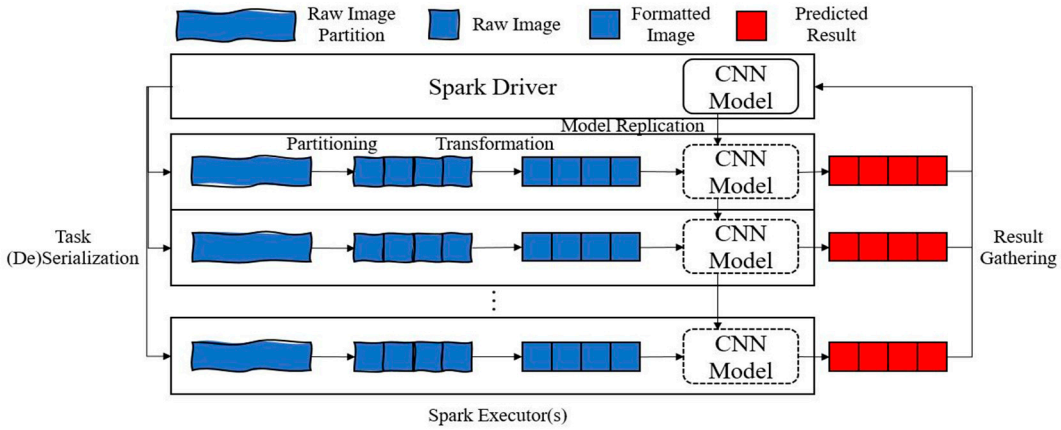


Figure 1. Processing procedure of Spark CNN based on model replication-based data parallelism.

3. Performance Model Feature Profiling

This section describes the process of selecting the features for use in the performance prediction model. A time to collect data for building the performance model depends on which model features are used and how the range of each model feature is defined. As the number of model features and the range of the features expand, the time for searching the feature map increases exponentially. This study examined the influence of each feature on the CNN's completion time in a Spark to determine which model parameters to use. It also reduces the search range for each model feature.

The features used in the existing performance prediction model and those to be used in this study are shown in Table 1.

Table 1. Features of the existing models and our model.

	Features
Existing Features [18,19]	- Ratio of data size to number of cores - Log of number of cores - Data size - Sampling ratio of input data for training prediction model - Number of cores - The number of executors - Size of partition
Virtual Resource Configuration Features	- The number of cores per executor - Size of memory per an executor

In all related works [18–22], the data size, number of cores, and some features related to Spark were used as performance model features. However, the features related to virtual resource configuration in the spark cluster environment were not considered. Therefore, in this study, we used not only the size of input data, number of executors, and the number of cores per executor, but also the memory size per executor as performance model features. Note that the effect of each feature on the completion time was evaluated based on the experimental results, as discussed in Section 5.

3.1. Feature 1: Input Data Sampling Rate

In the overview of processing data-parallel distributed network structure based on model replication, there are preprocessing steps of transforming an image into a fixed format, feedforward

step of calculating output of CNN layers, and backpropagation step of updating CNN model parameters [28]. In the first step, each image is converted into a structure defined in the CNN model. As all images are converted to the same structure, a processing time per image is the same.

In the feedforward step, CNN multiplies the input value with weights of each layer, adds a bias, and convey the result to the activation function which is to be used as an input in the next layer. As these routines are performed in the convolution layer, pooling layer, and the fully connected layer that constitute the CNN, a processing time per image in this step is also the same.

In the backpropagation step, the model parameters are updated to minimize the error between the predicted and actual values of the CNN model. The step proceeds with the reverse direction of feedforward step. Each layer of CNN calculates model parameter variation based on input values of the current layer and output values of the next layer. At last, the model parameters from the last layer to the first layer are updated. As the routines of the backpropagation step are always the same, the processing time is always the same.

As a result, the processing time for a particular sample of the CNN model is consistently the same in the same environment. Therefore, instead of using all of the CNN's input data to collect training and test data for the performance model, we collected samples using only a portion of the input data and then used extrapolation to generate a regression model for performance.

To observe the impact of the sampling ratio on processing time, we grouped the results, when the features (number of executors and cores, memory size per executor, and size of a partition) are the same, except for the sampling ratio. For each group, we divided the completion time by the maximum time of the group for scaling (e.g., scaled time $t_i' = t_i/\max(t_1, \ldots, t_n)$, where t_i is the target, and n is the number of the values of sampling ratio) Table 2 shows the minimum, average, and maximum values of the groups for each sampling ratio.

Table 2. Effect of sampling rate on scaled completion time.

	Inception V4			Resnet 50			Xception		
Sampling Rate (%)	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**
1	0.0102	0.0104	0.0106	0.0105	0.0107	0.0109	0.0104	0.0108	0.0110
10	0.095	0.098	0.100	0.097	0.100	0.101	0.096	0.097	0.098

When the other model parameters are the same, the sampling rates and ratio of the processing time of the sampling rates to that of the entire input are almost the same. To demonstrate that a lower sampling rate ensures a higher prediction accuracy of processing time and also reduces the time required for data collection, we presented the prediction accuracy of the completion time, when different sampling rates were used. This is discussed further in Section 5.

3.2. Feature 2: Executor Memory Size

When processing CNNs in Spark, we considered the influence of memory size on the processing time. Spark runs in a Java environment. As Spark runs in a Java environment a Spark application conforms memory management policy of JVM. Out of Memory Error (OOME) occurs when there is not enough memory to create the essential object while running a Spark application. As discussed in Section 2.1, the size of the executor memory should be considered along with the size of the partition, because the maximum memory usage of the executor increases as the size of the partition increases. In this study, we used the appropriate partition size for the executor memory size through a grid search. When processing CNN in Spark, we measured the processing time by increasing the memory size of an executor to examine the correlation between the processing time and memory size of an executor. The completion times with the same parameters, except for the memory size per executor, were grouped, and the times for each group are scaled as same as the scaling of the sampling rate. Table 3 shows the minimum, average, and maximum value for all groups in the CNN model. As shown in Table 4, when allocating more memory to the executor than the memory that prevents OOME,

the variation of the scaled completion time of all models is tiny. The memory size of the executor has little effect on the completion time. Therefore, appropriate memory size should be allocated to the executor for preventing OOME and for saving memory.

Table 3. Effect of the memory size of executor on Scaled Completion Time (SCT).

	Inception V4			**Resnet 50**			**Xception**		
	Min	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**
SCT	0.98	0.99	1	0.95	0.97	1	0.93	0.97	1

Table 4. Effect of the number of images per partition on the SCT of each CNN model.

	Inception V4			**Resnet 50**			**Xception**		
	Min	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**
SCT	0.99	0.99	1	0.94	0.97	1	0.98	0.99	1

3.3. Feature 3: Number of Images in Partition (Partition Size)

The size of CNN's input data is calculated as the product of the number of partitions and size of the partition (number of images in the partition). Conventionally, if the number of partitions is very high, the number of tasks increases, which incurs additional overhead in assigning tasks to threads [29]. We observed the completion times by increasing the number of images in a partition. The completion times with same parameters, except for the size of the partition, are grouped, and the times for each group are scaled as same as the scaling of the previous features. Table 4 shows the minimum, average, and maximum values for all groups in each CNN model. The variation of the scaled completion time of all models is tiny. As a result, if the number of images per partition is in a specific range that does not generate the OOME, it does not significantly affect the performance. However, larger partitions might possibly increase the minimum memory requirements of the executor. Therefore, we utilized a grid search to select the partition size suitable for the memory size of the executor.

3.4. Number of Cores and Executors

In general, it is known that the throughput increases in a quasi-linear fashion, as the number of cores increases in parallel processing or additional executors are added. We analyzed the correlation of throughput with completion time by increasing the number of cores per executor and the number of executors. The core is a thread that executes the parallel processing in the executor. Therefore, as the number of cores increases, parallelism can be increased at the executor level. As the number of executors denotes the number of parallel processors, parallelism can be increased at the cluster level.

To analyze the relationship between the number of cores and executors and the completion time at application level, the completion time was measured by varying the number of cores and number of executors. The completion time of the result that has the same feature values except for each of the two is grouped, and the completion time for each group is scaled in the same way of previous features. Table 5a shows the averaged values of all groups of each CNN model corresponding to the number of cores. Table 5b shows the averaged values of all groups of each CNN model corresponding to the number of executors.

Table 5. Effect of the number of cores on the SCT of each CNN model.

(**a**)												
Model	**Inception V4**				**Resnet 50**				**Xception**			
# of Cores	2	4	6	8	2	4	6	8	2	4	6	8
SCT	1	0.66	0.52	0.47	1	0.65	0.51	0.47	1	0.65	0.48	0.41
(**b**)												
Model	**Inception V4**				**Resnet 50**				**Xception**			
# of Executors	1	2		3	1	2		3	1	2		3
SCT	1	0.51		0.34	1	0.50		0.34	1	0.50		0.33

In addition, we measured the average processing time of a single task by varying the number of cores and number of executors to determine the influence of these two variables at partition level. Grouping and scaling were performed in similar ways. Table 6a shows the averaged scaled time of all groups with respect to the number of cores. Table 6b shows the averaged scaled time of all groups with respect to the number of executors.

Table 6. Effect of the number of cores on the SCT of each CNN model.

(**a**)												
Model	**Inception V4**				**Resnet 50**				**Xception**			
# of Cores	2	4	6	8	2	4	6	8	2	4	6	8
SCT	0.53	0.70	0.83	1	0.53	0.68	0.81	1	0.60	0.78	0.87	1
(**b**)												
Model	**Inception V4**				**Resnet 50**				**Xception**			
# of Executors	1	2		3	1	2		3	1	2		3
SCT	1	0.99		0.99	0.99	0.99		1	1	0.99		0.99

It is observed from Table 5 that the completion time is inversely related to the number of cores. However, at task level, the processing time per task increases, as the number of cores per executor increases, even though the executor handles the same tasks, as listed in Table 6. Therefore, it is clear that the number of cores exhibit a nonlinear proportional relationship with the overall throughput of executor. An increase in processing time per task with an increase in the number of cores while processing the same task implies that Spark demonstrates a low parallelism at executor level. This aspect will be analyzed in the future work through hardware level profiling.

4. Performance Model

In this section, we present two performance models for estimating the completion time of Spark CNNs in a cluster environment. In addition, we present various scenarios to verify the performance of the models. Scenarios use not only the features used in previous studies but also those related to resource configuration that need to be defined in a cluster environment. The scenarios also include extrapolation using sampling rates as a feature to reduce the time utilized to build the model. As the two performance models presented learn features based on machine learning techniques, we considered the various machine learning techniques that are used for each scenario. The performance model building procedure is described in Figure 2. When a user submits the CNN model, Training Data Collector configures, and aggregates feature vectors according to the scenario. The Training Data Collector transmits the vectors to Spark Cluster, and the cluster executes the CNN model according to the submitted feature vectors iteratively. The Spark Cluster transmit output vectors to the Hierarchy

LV1 model. In Hierarchy LV1, machine learning models are trained with the hyper-parameter search. The model which has the highest prediction accuracy and output vectors are transmitted to Hierarchy LV2. The Hierarchy LV2 trains machine learning models with submitted data.

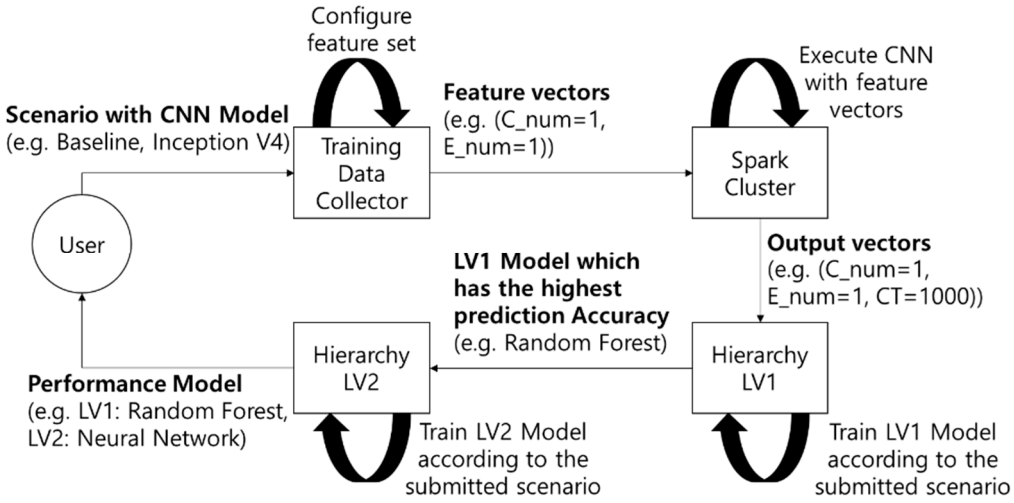


Figure 2. The performance model building procedure.

4.1. 1-Level-Hierarchy Performance Model

The first performance model predicts the processing time by applying the machine learning techniques to the model to learn the feature set, as shown in 1-level-hierarchy.

1-level-hierarchy: features→Completion Time

Various machine learning models are used, because it is not known whether the relationship between the completion time and features is linear or nonlinear. We used the L1 normalization (Lasso) and L2 normalization (Ridge) for linear regression models and Random Forest and artificial neural networks for nonlinear regression models. The features used in 1-level-hierarchy and the search range for each feature are shown in Table 7.

Table 7. Definition and range of all features.

Parameter	Range
E_mem: Memory size per executor	Minimum (MIN) memory to maximum (MAX) memory by memory scale
(E_num, C_total): (Number of executors, Total number of cores)	(1, MIN core to MAX cores by core scale), (2, 1 ∗ MAX cores + MIN core to 2 ∗ MAX cores by core scale), . . . (MAX executors, (MAX executors-1) ∗ MAX cores + MIN core to MAX executors ∗ MAX cores by core scale)
C_num: Number of cores per executor	For mod (C_total/E_num) of executors: Ceiling (C_total/E_num), Others: Floor (C_total/E_num) ∗ mod is modular function, Ceiling/Floor is round up/down function at the first decimal place.
P_size: Number of images per partition	MIN partition size to MAX partition size by partition size scale
SR: Sampling rate	MIN sampling rate to MAX sampling rate by sampling rate scale

∗ is product of the values.

E_mem, which is the memory size per executor, is determined by increasing the memory scale from the minimum memory size (MIN memory) to the maximum memory size (MAX memory) for the executor. Both E_num that denotes the number of executors, and C_total that denotes the total number of used cores affect resource configuration of an executor. E_num is increased by one from 1

to the maximum number of executors (MAX executors). MIN core and MAX core are the minimum and the maximum number of cores available on the executor in a cluster environment, respectively. C_total with E_num ranges from the product of (E_num-1) and maximum number of executor cores (MAX cores) plus minimum core number (MIN core) to the product of E_num and MAX cores by core scale. The number of cores from MIN core to MAX cores can be added to C_total, whenever E_num is increased. C_total is distributed to the executors as evenly as possible. The larger the number of cores in the executor, the longer the time required to process a single task that is shown at Table 6. If C_total is a multiple of the number of executors, it can be evenly distributed among the executors; however, some cores remain, if it is not a multiple. In this case, the remaining cores should be allocated to the executors one by one. The remainder of C_total divided by E_num executors is used to round up the quotient of C_total divided by E_num (first case of C_num). The executors obtain as many cores as the rounded down quotient of C_total divided by E_num (second case of C_num). P_size, which denotes the partition size, is set by increasing the partition size from the minimum number of partitions (MIN partition size) to maximum number of partitions (MAX partition size) by partition size scale. SR, which is the sampling rate, is determined by increasing the sampling scale from the MIN sampling rate to the MAX sampling rate. This method is similar to that used to determine the partition size. The actual setting of the features with the minimum, maximum, and scale values used in the experiment is described in Section 5.

4.2. Scenarios for 1-Level-Hierarchy Performance Model

This section describes the performance measurement scenarios of the features and the machine learning techniques used in the 1-level-hierarchy model. Spark CNN is executed in a cluster environment, and therefore, not only the total number of cores but also the number of executors should be considered. Thus, we used features from existing works as well as the reciprocal of the number of executors shown in the first row of Table 8 as baseline scenario.

Table 8. Performance Model Verification Scenarios, Usage Parameters, and Machine Learning Techniques by Scenario.

Case	Features	Machine Learning
Baseline	(C_num, 1/E_num)→CT	LASSO, RIDGE, RF, MLP
Scenario 1	(Baseline features, E_mem, SR)→CT	
Scenario 2	(Baseline features, SR)→CT	
Scenario 3-1	*hierarchy lv1-1*: (1/C_num)→CT_C_num, (1/E_num)→CT_E_num *hierarchy lv2-1*: (CT_C_num, CT_E_num)→CT	Model(s) in *hierarchy lv1*, *hierarchy lv2-1*, *hierarchy lv2-1*: LASSO, RIDGE, RF, MLP
Scenario 3-2	*hierarchy lv1-1*: (1/C_num)→CT_C_num, (1/E_num)→CT_E_num *hierarchy lv2-2*: (CT_C_num ∗ CT_E_num)→CT	
Scenario 4-1	*hierarchy lv1-2*: (1/C_num)→CT_C_num, (1/E_num)→CT_E_num, (SR)→CT_SR *hierarchy lv2-1*: (CT_C_num, CT_E_num)→CT	
Scenario 4-2	*hierarchy lv1-2*: (1/C_num)→CT_C_num, (1/E_num)→CT_E_num, (SR)→CT_SR *hierarchy lv2-2*: (CT_C_num ∗ CT_E_num ∗ CT_SR)→CT	

As the baseline does not use SR as a model feature, the completion times of CNN with respect to all the input images are used as the training and validation data. This results in an increase in the time required to collect samples to build the 1-level-hierarchy model. In Scenario 1, E_mem, P_size, and SR were added to the baseline features.

As SR is added to the feature set in Scenario 1, we extracted the images according to SR and used them as the input data for CNN. The completion times of the CNN with respect to the extracted images are used as training data. To apply the extrapolation to the regression model, the completion times of CNN with respect to all the input images are used as the validation data.

Finally, in Scenario 2, as mentioned in Section 3, only SR is added as a feature in the baseline scenario (not E_mem and P_size that demonstrate minimal effect on processing time). As each scenario uses two linear models and two nonlinear models, we evaluated the performance of a total of 12 cases, as listed in Table 8.

4.3. 2-Level-Hierarchy Performance Model

During the analysis of the impact of features on the completion time of Spark CNN, it was established that the processing speed of Spark CNN was only affected by the combination of feature values, when the hardware specification of the nodes in a cluster is homogeneous. Each of these features affect the completion time independently. Thus, we did not use multiple features in one machine learning model at once to predict the processing time. We suggested the 2-level-hierarchy performance model, which learns the individual features through machine learning techniques and the results of previous models back into machine learning techniques. *hierarchy lv1* represents a performance model at the first level of learning given features one by one. *hierarchy lv2-1*, *2-2* represents the performance model at the second level, which learns the completion time based on the results of the models at the first level.

hierarchy lv1: LV1_C_num(1/C_num)→CT_num, LV1_E_num(1/E_num)→CT_E_num, LV1_SR(SR)→CT_SR

hierarchy lv2-1: LV2-1(LV1_C_num(1/C_num), LV1_E_num(1/E_num), LV1_SR(SR))→CT

hierarchy lv2-2: LV2-2(LV1_C_num (1/C_num) ∗ LV1_E_num (1/E_num) ∗ LV1_SR (SR))→CT

hierarchy lv1 produces three regression models. C_num, E_num, and SR, which were correlated with the completion time, are trained individually by linear and nonlinear machine learning techniques. For each feature, we select the model with the highest prediction accuracy among the trained models. Thus, three regression models are created at this level: LV1_C_num, LV1_E_num, and LV1_SR. CT_C_num is the value obtained by grouping samples with the same value for the parameters, except for C_num, and dividing the completion time of each sample by the highest completion time in the group.

Similarly, CT_E_num and CT_SR imply that samples with the same feature values, except for E_num and SR, are grouped, and CT is divided by the highest processing time in the group. It could be viewed as a feature preprocessing in a broad sense. From a preprocessing point of view, the scaled times are relative values of the impact of each feature on CT for the resource with the highest completion time.

hierarchy lv2-1 uses the results of the three models created in *hierarchy lv1* as features. This model learns CT by linear and nonlinear machine learning techniques. The model, which demonstrates the highest prediction accuracy out of all the results obtained, is denoted as LV2-1. In *hierarchy lv2-1*, to understand the effect of combinations of features on CT, the model learned the impact of each feature on completion time instead of naive feature values.

hierarchy lv2-2 multiplies the results of the three models produced by *hierarchy lv1* and uses the obtained result as a feature. Similarly, the feature is trained by various machine learning techniques to select the best model as LV2-2. The reason for proposing *hierarchy lv2-2* is as follows. The executor always demonstrates a constant completion time during the processing of a specific task, unless there is performance interference from other tasks. The features that affect completion time are independent of each other. Assuming that the throughput of an executor can be expressed as a constant, it can be estimated by multiplying the rate of change provided by each feature to the baseline throughput,

when multiple features change. Even when *hierarchy lv2-2* uses extrapolation than *hierarchy lv2-1*, the prediction accuracy of completion time is higher.

4.4. 2-Level-Hierarchy Performance Model Scenario

The scenarios for verifying the performance of the 2-level-hierarchy model are as follows. Scenario 3-1 uses 1/C_num and 1/E_num as the features in *hierarchy lv1*, as shown in the fourth row of Table 8, to learn the preprocessed time CT_C_num and CT_E_num for each feature. Four machine learning techniques are used for each feature. For each feature, the model with the highest accuracy is selected, and therefore, two models are created at this level. In *hierarchy lv2-1*, the CT is trained using the two results of *hierarchy lv1* as the features. Four machine learning techniques are used to select one of the models with the highest prediction accuracy. Scenario 3-2 is generally the same as Scenario 3-1. However, as it uses *hierarchy lv2-2*, the completion time is trained using the product of the results of *hierarchy lv1*. Though Scenario 4-1 is similar to Scenario 3-1, a model has been added to train CT_SR using SR as a feature in *hierarchy lv1* to apply extrapolation. During the training, we train the model using some images extracted according to SR, and when we verify the model, we use the processing result of the entire data (SR = 100). Scenario 4-2 demonstrates the same process as Scenario 4-1, however, *hierarchy lv2-2* is used instead of *hierarchy lv2-1*.

5. Experimental Setup

This section presents the specific settings for the experiment. Section 5.1 provides a specification of the cluster environment and a description of the underlying software for Spark CNN. Section 5.2 introduces the datasets and CNN models used in the experiment. Section 5.3 presents the process of setting the values of the features and other detailed settings of the experiment. Section 5.4 describes the hyper-parameter setting of the machine learning techniques used, and finally, Section 5.5 describes the two metrics used for performance evaluation.

5.1. Experimental Environment

The cluster of experiments comprised one master and three worker nodes. Each node in the cluster used a core i7-6700, 16 GB of DDR4 memory, 256 GB of SSD, and a 1 Gbps network. We used Ubuntu 18.04 as the operating system and Mesos 1.8.0 for building container-based clusters. To implement CNN models in Spark, we used Spark 2.3.0, Spark Deep Learning Package 1.5.0, and Python packages, such as TensorFlow and Keras, for dependencies. The machine learning techniques, such as Lasso, Ridge, RF, and MLP, were used to build the performance model, and they were implemented with the APIs of Python sklearn 0.21.3.

5.2. Workloads

The yelp dataset [30] was used as the input data of Spark CNN. The yelp dataset comprises 200,000 images (114.8 GB), and the metadata for each image consists of photo_id, business_id, caption, and label. In this study, with model replication on Spark, Inception V4 [10], ResNet50 [11], and Xception [12] were adopted to classify the input images. When using SR as a feature, a random sampling of SR values was used for extracting images. To exclude the data replication and transmission time from the execution time of the CNN models, the input data was replicated for each node in advance. The DAG of each CNN model was always the same regardless of the SR, and it comprises two jobs and two stages per job.

5.3. Feature Values

Table 9 shows the range of the features used in the experiments. Each feature includes a minimum, maximum, and scale, as mentioned in Section 4. The executor memory size was set in 2 GB increments from 2–14 GB. The number of executors was set in increments of 1–3. The number of executor cores was

set in increments of 2–8, and the partition size was set in increments of 20–100 images. The partition size could be set from one image to the total number of images; however, if the number of images was very less, an empty task was generated. Therefore, we set the minimum value of the partition size to 20. In addition, when the partition size is very large, allocating the maximum memory of a machine to executor can still cause OOME. Therefore, we empirically determined the range of partition size, and the number of samples collected for the experiments was 1260. This was obtained from the product of the number of all possible feature values.

Table 9. Values of the features for Spark CNN.

Features	Range
E_mem	MIN memory: 2 GB, MAX memory: 14 GB, memory scale: 2 GB (If OOME occurs due to the partition size and number of executor cores, the minimum memory size is increased (e.g., 4–14 GB by 1 GB))
E_num	MIN exector:1, MAX executor: 3, executor scale: 1
C_num	MIN core: 2, MAX core: 8, core scale: 2
P_size	MIN partition size: 20, MAX partition size: 100, partition size scale: 20 (Partition size refers to the number of images contained per partition)
SR	1%, 10%, and 100% of CNN input images

Core Isolation and JVM Setup

When using Mesos as the cluster manager, the Spark's executor is created as a container. When the container is initialized, the number of cores, memory size, and storage size of the container can be configured. The created container uses the allocated memory size and storage size as the upper limit and the number of cores as the lower limit. Thus, containers completely utilize the available CPU during application runtime. While Spark CNN executes only the tasks that are related to the experiment, it causes the container to use more cores than the ones it has been allocated. To limit the upper bound of the number of cores of the container, we applied Stress [31] with the "-c"option (i.e., stress -c 2). Stress forces the number of available CPU cores on the node to adopt an integer value in the options.

In addition, when creating executors with the default Spark option, the initial heap memory size of the executors was 2 MB, which caused a major GC to expand the heap area continuously [32]. Therefore, the value of E_mem was assigned to JVM's -Xms option to render the minimum heap usage of the executor equal to the maximum usage from the beginning. Depending on the utilization of the memory heap, the size of the heap was expanded or shrunk, resulting in major GC. The heap size was fixed using the "-XX: -UseAdaptiveSizePolicy" option to prevent GC from changing the heap size.

5.4. Hyper-Parameter Tuning

In the 1-level-hierarchy performance model and the 2-level-hierarchy performance model, Lasso, Ridge, RF, and MLP were used as machine learning techniques, as listed in Table 8. Therefore, it was necessary to deduce a set of hyper-parameters that can maximize the prediction accuracy of performance models. The hyper-parameter values applied to each machine learning technique in this study are shown in Tables 10–12.

Table 10. Hyper-parameters of Lasso and Ridge regression.

Lasso and Ridge Regression	
Hyper-Parameter	**Values**
Penalty alpha	1×10^{-15}, 1×10^{-10}, 1×10^{-8}, 1×10^{-3}, 1×10^{-2}, 1, 5, 10, 20

Table 11. Hyper-parameters of Random Forest.

Random Forest	
Hyper-Parameter	**Values**
# of estimators	50, 100, 150
Max features	auto, sqrt, log2
Max depth	50, 100, 150

Table 12. Hyper-parameters of Multi-Layer Perceptron.

Multi-Layer Perceptron	
Hyper-Parameter	**Values**
Solver	Lbfgs, adam, sgd
Activation functions	Sigmoid, ReLU
Max iteration	5000, 10,000, 20,000
Learning rate alpha	0.01, 0.05
Hidden layer size	50, 100, 150

Each table includes a hyper-parameter search for each machine learning technique. Of all the cases, the one-leave-out cross-validation was applied to the training data, and the machine learning method and hyper-parameter set with the lowest mean squared error were used for validation. Lasso and Ridge use L1-norm and L2-norm as penalties to prevent overfitting the training data. The values of Lasso and Ridge used for each hyper-parameter are shown in Table 10. Table 11 shows the number of trees (# of estimators) and values of Max features to increase the accuracy of Random Forest prediction. The auto of the Max feature enables building a tree using all features, while sqrt and log2 imply building a tree using many features, as these functions are applied to the maximum number of features. The Max depth values are also provided to prevent overfitting. In the MLP model, Table 12, a search is performed on the Solver, Activation function, Max iteration (number of training iterations), Learning rate alpha, and the size of the hidden layer to prevent overfitting and under-fitting and improve prediction accuracy.

5.5. Performance Metrics

For obtaining the performance metrics, we used the mean absolute percent error (MAPE). It is used to determine the accuracy of the model and the time utilized in building the highest predictive model. MAPE measures the relative error between the correct and predicted values.

$$MAPE\ (\%)\ =\ \frac{100}{N}\sum_{k=1}^{N}\left|\frac{y_k-\hat{y}_k}{y_k}\right|$$

N denotes the number of samples, and y_k denotes the accurate answer value. $\hat{y}_k$ denotes the predicted value of the model with respect to the input value. In the 1-level-hierarchy performance model and *hierarchy lv2-1* and *lv2-2* in *2-level-hierarchy* performance model, y_k denotes the completion time, and $\hat{y}_k$ is the completion time predicted by the model. In the case of the *hierarchy lv1* of the *2-level-hierarchy* performance model, y_k and $\hat{y}_k$ are preprocessed time (e.g., CT_C_num, CT_E_num, and CT_SR) and the predicted preprocessed time, respectively.

The total time (T) to build the model comprises the sum of the time (t_1) utilized in collecting the training and validation data of the model, time (t_2) utilized to perform the hyper-parameter tuning, and to train the model.

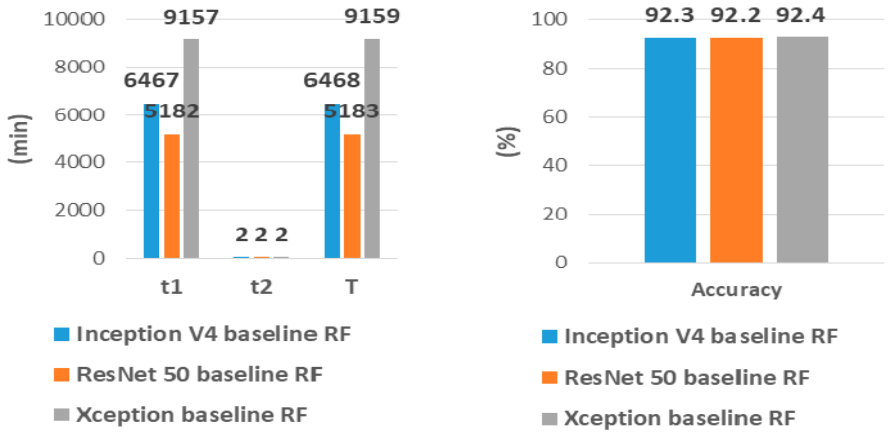
6. Experimental Result

In this section, we present the results of all the scenarios of the performance prediction model in a real container cluster-based spark environment. For each performance model scenario, we present the completion time prediction accuracy (100% -MAPE) for Spark CNN models and the total time taken to run the scenario. The execution time of a scenario is divided into the data collection time for training (t_1) and the time for the model generation (t_2).

6.1. Results for Baseline

The baseline uses the first row of Table 1 as features in the level-1-hierarchy model. The features required for the model are E_num and C_num, when the SR is 100%. The number of samples required to train the model is the product of the number of classes of the two features. Since E_mem and P_size are unnecessary, the samples generated by Table 9 must be preprocessed. Therefore, samples having the same E_num and C_num use the average completion time. In our study, according to Table 9, we have 35 cases, which is the product of E_mem and P_size for every 12 samples. A total of 12 samples are used for model training by grouping 35 cases of each sample and by averaging the CTs for each group. For predicting the accuracy of the model, we leverage leave-one-out-cross validation for Lasso, Ridge, RF, and MLP.

Figure 3a presents the used machine learning, accuracy, t_1, t_2, and T with the highest accuracy among all the performance models of the baseline for each CNN model. Figure 3b presents the highest prediction accuracy for each CNN model. Since the number of samples used for training is small, t_2 of the CNN models takes 2 min. Nevertheless, the prediction accuracy for each model is 92.3%, 92.2%, and 92.4%, an average of over 92%. In addition to RF, Lasso and Ridge averaged 89% and 90% for all CNN models, and MLP showed 49% accuracy on average.



(**a**) (**b**)

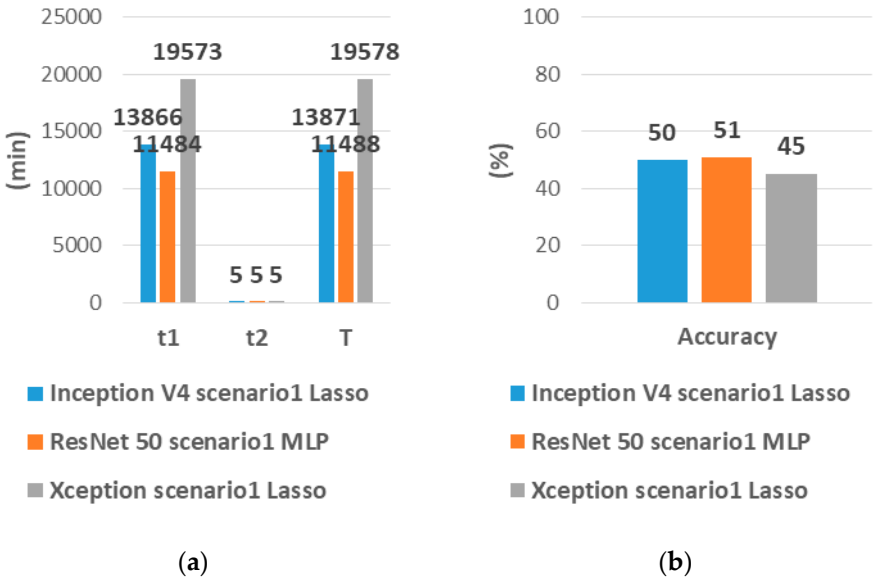
Figure 3. (**a**) Training data collection time t_1, training time t_2, and total time T of baseline scenario, (**b**) For each CNN model, the highest processing prediction accuracy of baseline scenario.

If only C_total of the two samples is the same and E_num and C_num are different, the CTs of the samples are also different, so the MLP has the lowest accuracy even by increasing the number of training iteration. (e.g., if C_total of two samples is 12, but E_num and C_num are 2, 6, or 3 and 4, respectively) t_1 of each model is 6467, 5182, 9157 min. Even though the number of samples is small, it takes hundreds of minutes to execute a CNN model and collect one sample.

6.2. Results for Scenario 1 & 2

Scenario 1 uses not only the baseline features but E_mem, P_size, and SR (1%, 10%) as features. Thus, 840 samples are generated for model training, which is the product of the classes of all features. These samples are used as training data. The remaining 420 samples with 100% SR are used as test data. We apply the training data to four machine learning techniques to create a model and present the model with the highest prediction accuracy and related metrics of the model.

Figure 4a presents the used machine learning, accuracy, t_1, t_2, and T with the highest accuracy among all the performance models of scenario 1 for each CNN model. Figure 4b presents the highest prediction accuracy for each CNN model. Scenario 1 has the lowest accuracy of all scenarios. Each CNN model has 50%, 51%, and 45% accuracy when machine learnings are Lasso, MLP, and Lasso. Besides, although it takes less time to collect each sample used in the training process, t_1 is also the highest because most samples are used for training. Scenario 1 confirms that using features that do not affect performance will reduce performance.

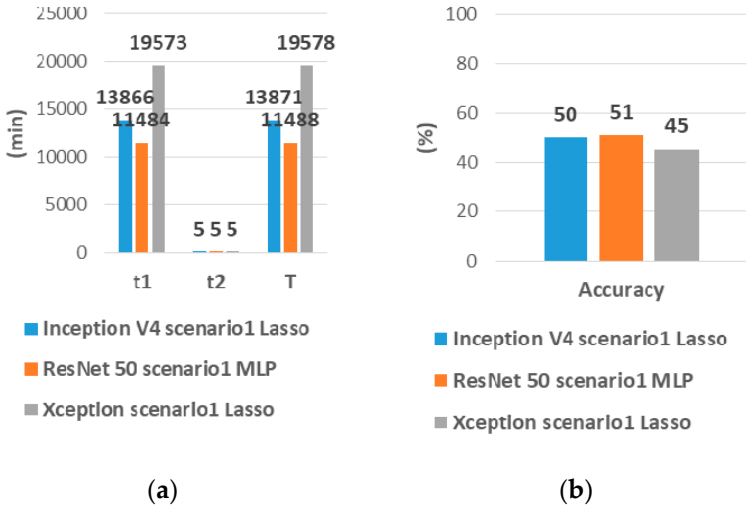


(**a**) (**b**)

Figure 4. (**a**) Training data collection time t_1, training time t_2, and total time T of scenario 1, (**b**) For each CNN model, the highest processing prediction accuracy of scenario 1.

In scenario 2, the total number of samples for model training per CNN is 24 by multiplying the number of baseline samples by the number of classes of SR (1%, 10%). The samples with 1% and 10% SR are considered as training data, and samples with 100% SR are used as test data. As shown in scenario 1, the best models of the four machine learning methods are selected, and detailed results are presented. Scenario 2 has more samples for training than the baseline, but the time required to collect a single sample is shorter than the baseline. So, T of scenario 2 is much shorter than that of baseline and scenario 1.

Figure 5a presents the used machine learning, accuracy, t_1, t_2, and T with the highest accuracy among the cases of all performance models of scenario 2 for each CNN model. Also, in Figure 5b, the highest prediction accuracy for each CNN model is presented. Scenario 2 has only an average T of 11% of the baseline. However, except for features that have less performance impact, accuracy is 55%, 51%, and 45% with MLP, MLP, and Lasso. Compared to the baseline, accuracy was reduced by nearly 40%, even though only SRs were added to the performance model. Scenario 2 demonstrates that accuracy can be lowered by applying data extrapolation to existing performance models (i.e., increasing the number of images used in CNN).



(**a**) (**b**)

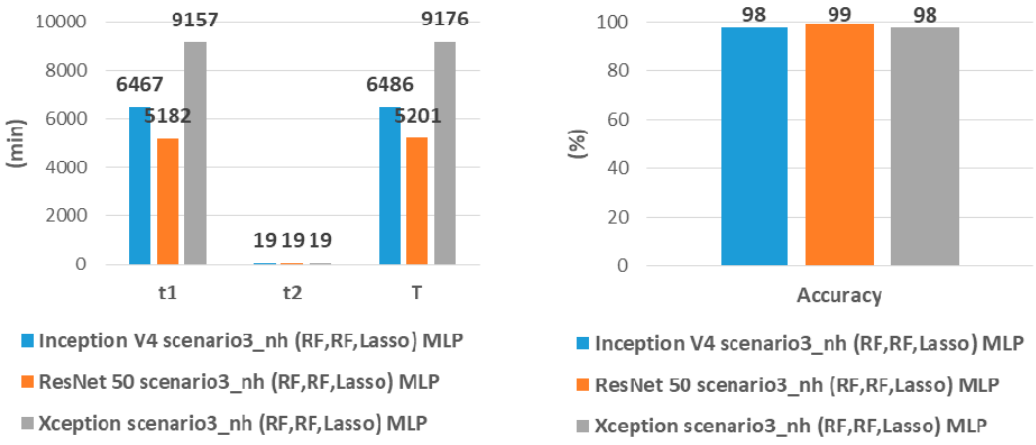
Figure 5. (**a**) Training data collection time t_1, training time t_2, and total time T of scenario 2, (**b**) For each CNN model, the highest processing prediction accuracy of scenario 2.

6.3. Results for Scenario 3 & 4

Scenario 3-1 uses C_num and E_num as features in *hierarchy lv1*. Therefore, when the SR is 100%, the number of samples used in *hierarchy lv1* is equal to the product of the classes of C_num and E_num. The cases caused by E_mem and P_size are processed in the same way as the baseline, so a total of 12 samples are used for the model training. In *hierarchy lv1*, four machine learning techniques are used to learn the relationship between each feature and the preprocessed time (i.e., CT_C_num, CT_E_num). Each machine learning technique is trained by using all 12 samples. *hierarchy lv2-1* leverages the relationship between CT and the predicted value of the machine learning model with the highest accuracy among the results of *hierarchy lv1*. The number of samples of *hierarchy lv2-1* is the same as the number of samples of *hierarchy lv1*, so that is 12. Moreover, we apply leave-one-out-cross validation for *hierarchy lv2-1* and present the average accuracy. Scenario 3-2 is similar with Scenario 3-1. However, since *hierarchy lv2-2* instead of *hierarchy lv2-1* is utilized, the model uses only one feature, which is the product of the results of *hierarchy lv1*.

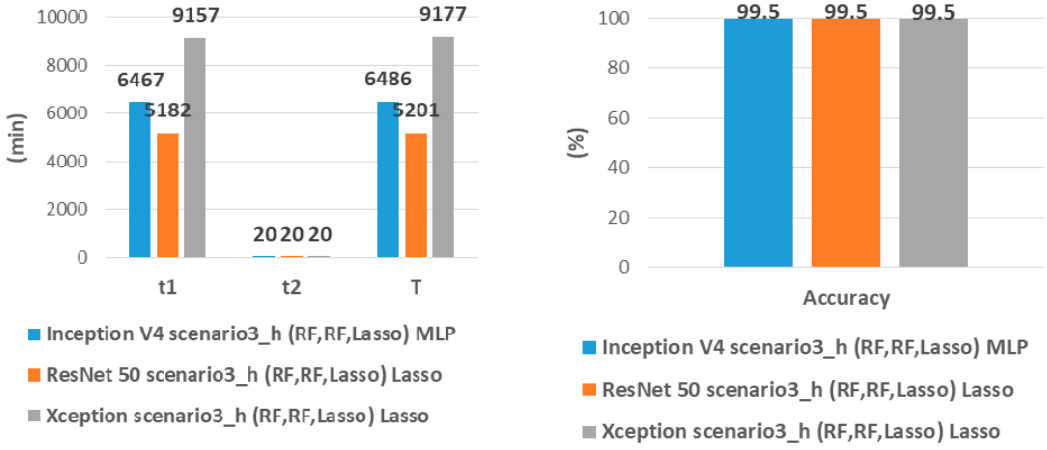
Figure 6a describes the t_1, t_2, and T of scenario 3-1. The prediction accuracy is presented in Figure 6b. The combination of the most accurate machine learning techniques for all CNN models in *hierarchy lv1* (RF, RF) averages over 99%. The most accurate machine learning technique for all CNN models in *hierarchy lv2-1* is an MLP of more than 98% on average. Because the relationship between the preprocessed features and the actual processing time is nonlinear, RF and MLP are more accurate than linear machine learning techniques, and MLP has the best performance. t_1 is the same as the baseline, but t_2 takes 20 min because of the 2-level-hierarchical model. However, when comparing the T of the scenario 3-1 and the T of the baseline, there is only a difference of 0.2% of the time, so the total time difference is insignificant. Figure 7b presents the t_1, t_2, and T of scenario 3-2. The prediction accuracy is shown at Figure 7b. The output of *hierarchy lv1* is similar to scenario 3-1. However, the most accurate machine learning technique for all CNN models in *hierarchy lv2-2* is Lasso, which averages over 99%. The accuracy is higher than the baseline and the scenario 3-1. The most accurate machine learning in *hierarchy lv2-2* is Lasso, which means that the relationship between the product of the preprocessed times and the CT is linear. t_1, t_2, and T are similar to scenario 3-1.

Scenario 4-1 is generally similar to scenario 3-1. However, in *hierarchy lv1*, we further learn the relationship between SR and CT_SR. Therefore, the number of samples used for learning is 24, considering the case of 1% and 10% SR. Then, we append the relationship between the predicted results and CT in *hierarchy lv2-1*. Scenario 4-2 is similar to scenario 4-1 but utilizes *hierarchy lv2-2* for learning the relationship between CT and product of the results of *hierarchy lv1*.



(**a**) (**b**)

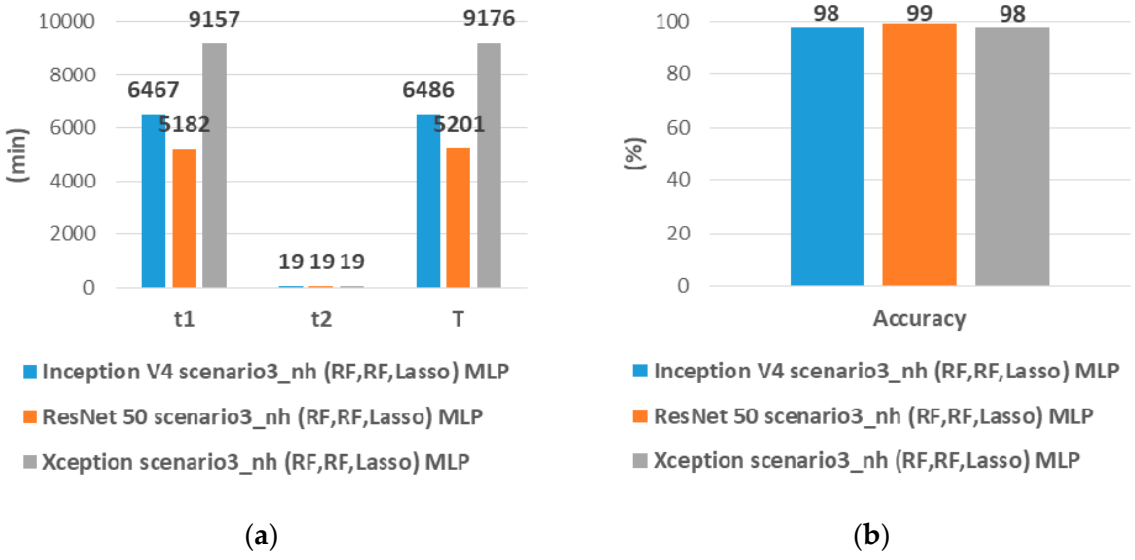
Figure 6. (**a**) Training data collection time t_1, training time t_2, and total time T of scenario 3-1, (**b**) For each CNN model, the highest processing prediction accuracy of scenario 3-1.



(**a**) (**b**)

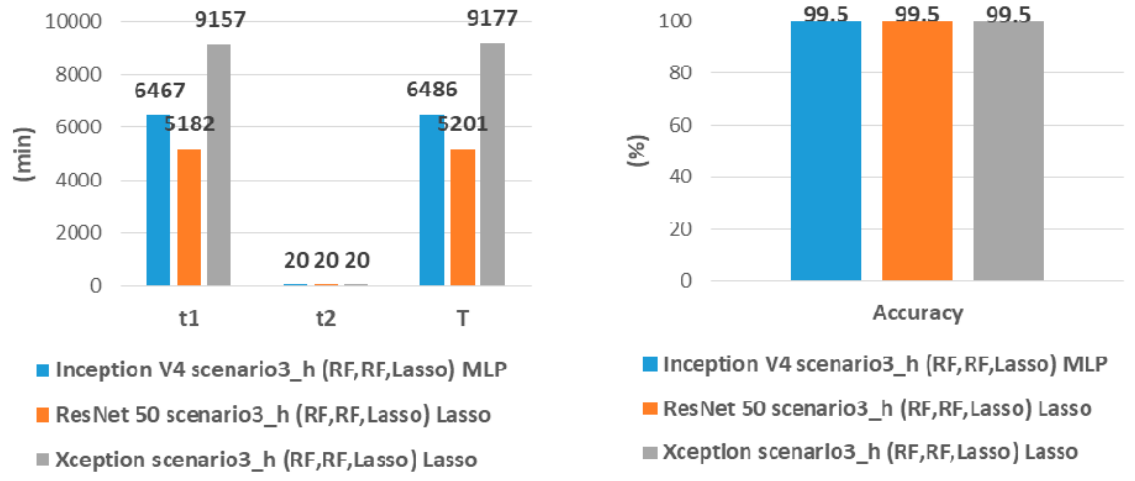
Figure 7. (**a**) Training data collection time t_1, training time t_2, and total time T of scenario 3-2, (**b**) For each CNN model, the highest processing prediction accuracy of scenario 3-2.

Figure 8a presents the results of scenario 4-1 and the prediction accuracy is compared in Figure 8b. The model of feature SR and CT_SR added to *hierarchy lv1* has the highest accuracy of Lasso, or Ridge, because the relationship between them is linear. Although the average accuracy of 99.7% was predicted in *hierarchy lv1*, the accuracies of *hierarchy lv2-1* for all CNN models are 57.3%, 57.1%, and 54.3% which were lower than the baseline, scenario 3-1, and scenario 3-2. It is the same phenomenon that the accuracy is lowered when applying the trained model to large data size (i.e., SR = 100) such as scenario 2. In terms of time, *t2* is increased to 45, 45, 45 min, but t_1 is reduced to 710, 570, 1007 min, so T is the smallest of all scenarios. Finally, Figure 9a describes the results of scenario 4-2 and Figure 9b presents the prediction accuracy of scenario 4-2. In scenario 4-2, the predicted results of *hierarchy lv1* as well as *hierarchy lv2-2* average 99.8%, which is the highest among all scenarios. The proposed model has the highest accuracy, even when data extrapolation is applied (i.e., training data has SR with 1%, 10%, but test data has SR with 100%). In terms of time, it is the same as in Scenario 4-1, which is 89% reduction from the baseline.



(**a**) (**b**)

Figure 8. (**a**) Training data collection time t_1, training time t_2, and total time T of scenario 4-1, (**b**) For each CNN model, the highest processing prediction accuracy of scenario 4-1.



(**a**) (**b**)

Figure 9. (**a**) Training data collection time t_1, training time t_2, and total time T of scenario 4-2, (**b**) For each CNN model, the highest processing prediction accuracy of scenario 4-2.

7. Related Work

Performance analysis and prediction of applications running on cluster-based general-purpose distributed processing framework have been performed from two perspectives. The most traditional methods rely on analytical models [13–16] and simulations [33,34].

Recent studies have used machine learning to predict this performance. A pioneering study of performance prediction in sparks is a regression model [8] proposed by Spark creators. The model used features based on the size of the input dataset and number of cores as the parameters of the regression model. The estimation of these model parameters is based on non-negative least squares. They also used the Optimal Experiment design for obtaining a high prediction accuracy from small training samples. However, we predicted performance by adding resource configuration parameters related to memory and the number of spark executors in the container-based cluster environment. To reduce the time involved in collecting the samples and building the performance model, we analyzed the processing procedure of the CNN models on Spark and applied extrapolation to the model to reduce the search space of the features and hyper-parameters.

Ref [19] proposed the platform, which predicts the performance of Spark SQL queries and machine learning applications by utilizing the pre-runtime features and run features associated with each stage of the Spark application. The authors reported a 25% prediction error for the machine learning applications. However, this prediction accuracy is not high when compared to those of the machine learning-based performance prediction models, and no prediction is conducted for the latest deep learning models, such as CNN. In this study, we obtained a high prediction accuracy of 99% for popular CNN implementations.

Similar to this study, [21] searched for optimal settings that could minimize the resource usage costs. This method selects the best configuration out of the resource configuration candidates in the public cloud environment. However, we considered the fundamental correlation of each configurable resource component with low-level time metrics of Spark. Besides, as the applications are performed in a container-based cluster environment, the range of resource configuration boundaries is not limited, and the computation power of the processor can be considered.

Ref [35] used a strategy to generate training data for the performance model. The model was used to predict the Spark settings with the minimum processing times for a given application. However, our work predicted the performance of CNN implementations in a given environment, including resource configurations and configurable Spark parameters.

The latest research [34] uses CNN configuration parameters, hardware characteristics, and even network characteristics as input values of the prediction model. Besides, through the temporal prediction of each layer of CNN, high prediction accuracy was achieved at a microscopic point of view. In our study, the goal of performance prediction for CNN is the same, but the difference is that resource configuration parameters are used as input values in consideration of an environment in which virtual resources can be configured, such as the cloud. For example, the memory size of the virtual resource, the number of virtual cores, and the number of virtual resources is used as resource configuration parameters. Also, [34] created a prediction model based on the parametric equation for each step of CNN processing. On the other hand, this study applied machine learning techniques for predictive models. Although it is also an advantage of machine learning-based performance prediction, the prediction model in this paper has the difference that it can be applied even if the hardware characteristics change or the platform on which the CNN runs is changed.

Ref [22] researched to predict the processing time of applications capable of parameter tuning in a single node environment. Correctly, for various applications, including Apache web server and SQLite, various configuration options such as binary configuration options and numeric configuration options were embedded and used as input values of the processing time prediction model. As a prediction model, processing time was predicted using DeepPerf, which was created by applying L1 or L2 normalization to DNN, and tuning parameters. The difference in this study is that the processing time is predicted in a distributed parallel processing environment considering that distributed processing and parallelization occurs when the volume of the application increases. Also, in this paper, to train a predictive model at a lower cost, it is possible to apply a nonlinear regression-based extrapolation even if the size of the data set increases.

8. Conclusion and Future Work

In this study, we investigated the model that could predict the performance of CNN models implemented in a Spark cluster. Firstly, we analyzed the processing of model replication-based CNN implementations in Spark and explained the impact of conventionally used features on the performance of the model. We presented a 2-level-hierarchy performance prediction model based on the fact that each feature independently affected the completion time. The proposed model showed a high prediction accuracy of 99% for all CNN models that trained on a small amount of data selected through sampling and validated against the original input data (even when data extrapolation was applied). Through various validation scenarios, we showed that the existing performance model demonstrated a high predictive accuracy of 92% for the input data collected from the original data. However,

when data extrapolation was applied to the existing model, Inception V4, ResNet50, and Xception only demonstrated a prediction accuracy of 55%, 50%, and 45%, respectively.

In future works, we will analyze, at the hardware level, the cause for the decline in parallelism with an increase in the number of cores of executors processing Spark CNN models (discussed in Section 4). Further, we will develop a system that recommends a suitable cluster volume and executor resource configuration that can handle Spark CNN models at the lowest cost in a given time by applying a 2-level-hierarchy performance prediction model, even when the processor is a graphics processing unit.

Author Contributions: Conceptualization, R.M.; methodology, R.M.; software, R.M.; validation, R.M., H.Y.; formal analysis, R.M.; investigation, R.M.; resources, R.M., H.Y.; data curation, R.M.; writing—original draft preparation, R.M., H.Y.; writing—review and editing, R.M., H.Y.; visualization, R.M., H.Y.; supervision, H.Y.; project administration, H.Y.; funding acquisition, H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2018-0-01405) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00480, Developing the edge cloud platform for the real-time services based on the mobility of connected cars).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Li, P.; Chen, X.; Shen, S. Stereo R-CNN Based 3D Object Detection for Autonomous Driving. In Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 16–20 June 2019; pp. 7636–7644.
2. Cai, Z.; Vasconcelos, N. Cascade R-CNN: High Quality Object Detection and Instance Segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.* **2019**, *1*. [CrossRef] [PubMed]
3. Sun, Y.; Xue, B.; Zhang, M.; Yen, G.G.; Lv, J. Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification. *IEEE Trans. Cybern.* **2020**, 1–15. [CrossRef] [PubMed]
4. Gong, Z.; Zhong, P.; Yu, Y.; Hu, W.; Li, S. A CNN With Multiscale Convolution and Diversified Metric for Hyperspectral Image Classification. *IEEE Trans. Geosci. Remote. Sens.* **2019**, *57*, 3599–3618. [CrossRef]
5. Mammone, N.; Ieracitano, C.; Morabito, F.C. A deep CNN approach to decode motor preparation of upper limbs from time–frequency maps of EEG signals at source level. *Neural Netw.* **2020**, *124*, 357–372. [CrossRef] [PubMed]
6. Kwon, S.; Mustaqeem. A CNN-Assisted Enhanced Audio Signal Processing for Speech Emotion Recognition. *Sensors* **2019**, *20*, 183. [CrossRef]
7. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.H.; Shenker, S.; Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11), Boston, MA, USA, 30 March–1 April 2011.
8. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing ACM, Santa Clara, CA, USA, 1 October 2013.
9. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
10. Szegedy, C.; Ioffe, S.; Vanhoucke, V.; Alemi, A. Inception-v4, inception-resnet and the impact of residual connections on learning. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, California, CA, USA, 4–9 February 2017.
11. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, Las Vegas, NV, USA, 27 June 2016.
12. Chollet, F. Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017.

13. Nelson, R.; Tantawi, A. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Trans. Parallel Distrib. Syst.* **1988**, *37*, 739–743. [CrossRef]
14. Mak, V.; Lundstrom, S. Predicting performance of parallel computations. *IEEE Trans. Parallel Distrib. Syst.* **1990**, *1*, 257–270. [CrossRef]
15. Ipek, E.; De Supinski, B.R.; Schulz, M.; McKee, S.A. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2005.
16. Ardagna, D.; Barbierato, E.; Evangelinou, A.; Gianniti, E.; Gribaudo, M.; Pinto, T.B.M.; Guimarães, A.; Da Silva, A.P.C.; Almeida, J.M. Performance prediction of cloud-based big data applications. In Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering ACM, Berlin, Germany, 1 April 2018.
17. Pei, Z.; Li, C.; Qin, X.; Chen, X.; Wei, G. Iteration Time Prediction for CNN in Multi-GPU Platform: Modeling and Analysis. *IEEE Access* **2019**, *7*, 64788–64797. [CrossRef]
18. Venkataraman, S.; Yang, Z.; Franklin, M.; Recht, B.; Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In Proceedings of the 13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16), Santa Clara, CA, USA, 13–17 March 2016.
19. Mustafa, S.; Elghandour, I.; Ismail, M.A. A machine learning approach for predicting execution time of spark jobs. *Alex. Eng. J.* **2018**, *57*, 3767–3778. [CrossRef]
20. Pan, X.; Venkataraman, S.; Tai, Z.; Gonzalez, J. Hemingway: Modeling distributed optimization algorithms. *arXiv Prepr.* **2017**, arXiv:1702.05865.
21. Alipourfard, O.; Liu, H.; Chen, J.; Venkataraman, S.; Yu, M.; Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017.
22. Ha, H.; Zhang, H. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, MN, Canada, 25–31 May 2019; Institute of Electrical and Electronics Engineers (IEEE); pp. 1095–1106.
23. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauly, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation USENIX Association, Berkeley, CA, USA, 25–27 April 2012.
24. Maros, A.; Murai, F.; Da Silva, A.P.C.; Almeida, J.M.; Lattuada, M.; Gianniti, E.; Hosseini, M.; Ardagna, D. Machine Learning for Performance Prediction of Spark Cloud Applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD) IEEE, Milan, Italy, 8–13 July 2019.
25. Reiss, C. *Understanding Memory Configurations for In-Memory Analytics*; Diss; University of California: Berkeley, UC, USA, 2016.
26. Ehringer, D. *The Dalvik Virtual Machine Architecture*; Techn. Report (March 2010)4.8 (2010); David Ehringer: New Hampshire, MN, USA, 2010.
27. Deep Learning Pipelines for Apache Spark. Available online: https://github.com/databricks/spark-deep-learning.git (accessed on 12 December 2019).
28. 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 ACM, Sydney, Australia, 10–13 August 2015.
29. Paul, A.K.; Zhuang, W.; Xu, L.; Li, M.; Rafique, M.M.; Butt, A.R. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER) IEEE, Taipei, Taiwan, 12–16 September 2016.
30. Yelp Open Dataset. Available online: https://www.yelp.com/dataset (accessed on 12 December 2019).
31. Stress (1)—Linux Man Page. Available online: https://linux.die.net/man/1/stress (accessed on 12 December 2019).
32. The Java® Virtual Machine Specification. Available online: https://docs.oracle.com/javase/specs/jvms/se9/html/index.html (accessed on 22 July 2020).
33. Bertoli, M.; Casale, G.; Serazzi, G. JMT: Performance engineering tools for system modeling. *ACM SIGMETRICS Perform. Eval. Rev.* **2009**, *36*, 10–15. [CrossRef]

34. Wang, K.; Khan, M.M.H. Performance prediction for apache spark platform. In Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), New York, NY, USA, 24–26 August 2015; pp. 166–173.
35. Nguyen, N.; Khan, M.M.H.; Wang, K. Towards automatic tuning of apache spark configuration. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD) IEEE, San Francisco, CA, USA, 2–7 July 2018.