IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 24, 2019, accepted February 14, 2020, date of publication March 10, 2020, date of current version March 20, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2979812*

# ATCS: Auto-Tuning Configurations of Big Data Frameworks Based on Generative Adversarial Nets

**MINGYU LI[1,2], ZHIQIANG LIU[1], XUANHUA SHI[1], (Senior Member, IEEE), AND HAI JIN[1], (Fellow, IEEE)**

[1]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
[2]School of Mathematics and Computer Science, Liupanshui Normal University, Liupanshui 553004, China

Corresponding author: Hai Jin (hjin@hust.edu.cn)

This work was supported in part by the National Key Research and Development Plan under Grant 2017YFC0803700, and in part by the NSFC under Grant 61772218 and Grant 61832006.

**ABSTRACT** Big data processing frameworks (e.g., Spark, Storm) have been extensively used for massive data processing in the industry. To improve the performance and robustness of these frameworks, developers provide users with highly-configurable parameters. Due to the high-dimensional parameter space and complicated interactions of parameters, manual tuning of parameters is time-consuming and ineffective. Building performance-predicting models for big data frameworks is challenging for several reasons: (1) the significant time required to collect training data and (2) the poor accuracy of the prediction model when training data are limited. To meet this challenge, we proposes an *auto-tuning configuration parameters system* (ATCS), a new auto-tuning approach based on *Generative Adversarial Nets* (GAN). ATCS can build a performance prediction model with less training data and without sacrificing model accuracy. Moreover, an optimized *Genetic Algorithm* (GA) is used in ATCS to explore the parameter space for optimum solutions. To prove the effectiveness of ATCS, we select five frequently-used workloads in Spark, each of which runs on five different sized data sets. The results demonstrate that ATCS improves the performance of five frequently-used Spark workloads compared to the default configurations. We achieved a performance increase of 3.5× on average, with a maximum of 6.9×. To obtain similar model accuracy, experiment results also demonstrate that the quantity of ATCS training data is only 6% of *Deep Neural Network* (DNN) data, 13% of *Support Vector Machine* (SVM) data, 18% of *Decision Tree* (DT) data. Moreover, compared to other machine learning models, the average performance increase of ATCS is 1.7× that of DNN, 1.6× that of SVM, 1.7× that of DT on the five typical Spark programs.

**INDEX TERMS** Big data, generative adversarial nets, spark, genetic algorithm, automatic tune parameters.

## I. INTRODUCTION

Currently, multiple big data applications are required to mine valuable information from the big data, but developing a unique processing framework for each application is cost-prohibitive. The typical method is to design a general big data processing framework [1] that supports multiple big data applications. To achieve high performance in every big data application, framework developers provide users with several highly-configurable parameters. For example, Spark [2] has more than 180 parameters, and can be used for applications such as machine learning [3], graphics computing [2], [3], stream computing [4], and database management [5].

Consequently, these general big data processing frameworks have high-dimension parameter spaces, and offer several selectable configurations for each application. Moreover, the interactions of parameters is cumbersome. For example, *spark.executor.memory* and *spark.default.parallelism* are two common parameters of Spark: *spark.executor.memory* is used to set the memory of the executor, and *spark.default.parallelism* is used to set the number of tasks. If *spark.executor.memory* is set to a small value while *spark.default.parallelism* is large, the performance of Spark does not significantly improve, or even decreases, because there is

The associate editor coordinating the review of this manuscript and approving it for publication was Junjie Wu.


This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

**TABLE 1.** Time cost of collecting training data.

| Workload and Input data size | KM (M Points) | | | PR (M Pages) | | |
|---|---|---|---|---|---|---|
| | 1 | 5 | 10 | 0.5 | 0.6 | 1.2 |
| Time cost (500 sets) | 2.04 (h) | 5.98 (h) | 11.50 (h) | 22.36 (h) | 23.48 (h) | 55.96 (h) |
| Workload and Input data size | MF (M Points) | | | TC (K Points) | | |
| | 0.36 | 0.49 | 0.56 | 10 | 50 | 100 |
| Time cost(500 sets) | 28.06 (h) | 28.43 (h) | 30.19 (h) | 5.48 (h) | 18.61 (h) | 52.99 (h) |

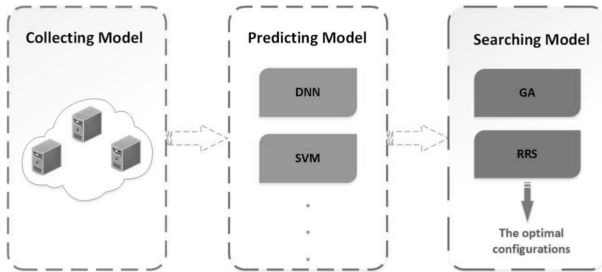



**FIGURE 1.** Machine learning (ML)-based Model.

not enough memory to run many tasks while the program is running. Also, every big data application requires a specific subset of hardware resources [6]. For example, WordCount requires a subset of resources based on CPU [7], and TeraSort requires a subset of resources based on CPU and memory [5], [7], so the optimal configuration of different workloads (e.g., WordCount, TeraSort) is nonconformity. Therefore, it is difficult for users to select the optimal configuration parameters in such a large parameter space. As a result, they often accept the default configurations, which are inefficient or even unavailable.

Generally, users want to explore a set of optimal configurations for applications, which requires rich experience and in-depth knowledge of both frameworks and applications. Manual tuning of parameters is time-consuming and labor-intensive. It is thus urgent to explore an efficient and accurate auto-tuning method for general big data processing frameworks.

Current, parameter auto-tuning methods fall into three classifications: *Program Analysis* (PA)-based, *Machine Learning* (ML)-based, and Search-based [8]. The ML-based approach has received significant attention recently, such as DAC [5] and CBM [8]. Fig. 1 illustrates the ML-based approach consisting of three components: a collecting model, a predicting model, and a searching model. The collecting data component is used to collect the execution time of different workloads with different configurations. The predicting model component uses various machine learning algorithms to predict the performance (*execution time*) of any set of configurations on a given workload. The searching model component uses a heuristic algorithm [9] (e.g., GA or RRS) to explore the optimum configuration in the parameter space. The process of the ML-based approach is to train the performance-prediction model using the training data collected by the collecting model; then, the searching model can search for optimum configurations based on the prediction result of the predicting model.

However, using the ML-based approach for parameter tuning for general big data frameworks faces several challenges. First, training an accurate performance prediction model requires significant time to collect training data; the accuracy is poor when the amount of training data is small. Table 1 reports the time cost of collecting training data, with four workloads, each having three different input sizes. We find that it requires up to 55.96 hours to collect 500 sets of PageRank training data, and most of the other workloads take more than 20 hours. If the input data size increases, more time is required. A job occasionally requires hours or more, so collecting training data is time-consuming. Second, it is difficult to explore a machine learning algorithm to build a performance prediction model that shows sufficient predictive performance on most workloads.

To overcome the challenges, we propose an auto-tuning configuration parameters system (*ATCS*) based on *Generative Adversarial Nets* (GAN) [10] that can build a performance prediction model with less training data and without sacrificing model accuracy. ATCS consists of three components: a *Random Parameter Generator* (RPG), a *Performance Prediction Generative Adversarial Nets* (PPGAN) model, and a Searching model. The function of the RPG is to randomly generate configurations within the scope allowed by cluster resources and big data frameworks. These configurations will be configured on different workloads to collect data for training PPGAN. PPGAN is a performance prediction model based on GAN. Compared to other machine learning algorithms (*DNN, SVM, and DT*), GAN is superior at fitting data distributions. GAN uses confrontation training to reduce the complexity of the model, thus reducing the amount of data required for training. Moreover, because of the huge parameter space and complex interaction between parameters, the exhaustive method is inefficient. Instead, an optimized *Genetic Algorithm* (GA) [11] is used in the searching model to search the parameter space for optimum solutions.

To prove the effectiveness of ATCS, we select five frequently-used workloads in Spark, each of which runs on five different sized data sets. The results demonstrate that ATCS improves the performance of five frequently-used Spark workloads compared with the default configurations. We achieved a performance increase of 3.5× on average, with a maximum of 6.9 ×. To obtain a similar model accuracy, experimental results find that the quantity of training data

required by ATCS is only 6% of DNN [12] data, 13% of SVM [13] data, and 18% of DT [14] data. Moreover, compared to other machine learning models, the average performance increase of ATCS is 1.7× that of DNN, 1.6× that of SVM, and 1.7× that of DT on the five typical Spark programs.

In this paper, our contributions are as follows:

1) We design and implement a parameter auto-tuning system on Spark to automatically configure Spark parameters.

2) We propose PPGAN, which can obtain a robust and high-precision performance prediction model with minimal training data compared to other machine learning models.

3) An optimized GA is used in ATCS to explore the optimum settings of big data frameworks.

We conducted extensive experiments on the challenges of auto-tuning parameters. The results demonstrates that quantity of training data required by ATCS is significantly less, and the performance more stable for different workloads, than other machine learning models, such as DNN, SVM, and DT.

The rest of this paper is organized as follows. Section II introduces important related research. Section III describes related background knowledge. Section IV explains the design details of ATCS. Section V illustrates the implementation details of ATCS. Section VI presents the experimental results. Section VII concludes the paper. Section VIII discuss some issues and future works in the paper.

## II. RELATED WORK

Automatically tuning configurations for big data frameworks have attracted significant research in recent years. Previous studies fall into three categories: Search-based [8] methods, PA-based methods, and ML-based methods.

Search-based methods consider parameter tuning a black-box optimization problem and search for parameter spaces based on specific rules (e.g., high-probability, gradient) to explore the optimal solutions. A recent related study of the search-based method is BestConfig [15], which uses the divide-and-diverge sampling method and the recursive bound-and-search algorithm to automatically tune configurations with limited resources for general systems. Kumar *et al.* [16] propose a noise-gradient algorithm, called *simultaneous perturbation stochastic approximation* (SPSA), to optimize Hadoop's performance. Moreover, several studies explore the search-based method, such as ACTS [17] and MRONLINE [18]. The search-based method is applicable to parameter optimization problems of multiple frameworks, without the need for corresponding high-level framework knowledge, but requires significant time to statistically analyze samples and iteratively search the parameter space.

The PA-based method captures performance characteristics using a fine-grained analysis of the run-time state of the program, and creates a simulator to simulate the job-execution process and predict performance. The performance prediction model created by this method is also called a cost-based model. MRTuner [19] proposes a PTC model to evaluate the cost of parallel execution between different tasks and designs an efficient search algorithm to identify the optimal execution plan. RFHOC [20] proposes to divide the map and reduce phases of the Hadoop jobs into multiple elementary operations, and then a Random Forest is used to assess the cost of each elementary operations. Herodotou and Babu [21] propose a Profiler to analyze Hadoop programs and design a What-if engine to predict the execution time of these programs. MR-COF [22] generates a cost file by monitoring and analyzing run-time behavior, and the PPM assesses the performance of Hadoop jobs based on the cost file. The PA-based method requires a fine-grained analysis of the run-time state of each stage within the job to build the simulator, which requires determining all factors that may affect performance. However, as the big data processing framework becomes more complex, this methods may not capture highly-complex run-time characteristics. Consequently, the PA-based method makes it difficult to model complex systems or migrate between different systems.

The ML-based method is used to train the performance prediction model using training data collected by the collecting model. The searching model can then explore optimum settings based on the prediction results from the prediction model, which is related to our work. For example, Yu *et al.* [5] propose a DAC composed of *Hierarchical Modeling* (HM) and a GA. HM is a prediction model built using regression trees, and GA is responsible for searching for optimal configurations in parameter space. ALOJA-ML [23] uses multiple machine learning models (eg, Regression Tree, Nearest Neighbors) to predict Hadoop program performance. Bao *et al.* [8] propose a novel comparison-based prediction model called CBM, built by a machine learning algorithm, and use *Latin hypercube sampling* (LHS) [24] to generate and search for optimal parameter configurations of distributed message systems. Bei *et al.* [25] built a Spark program performance prediction model by random forests and explore the optimum configurations using a genetic algorithm. Yigitbasi *et al.* [26] propose a SVM based method to automatically tune the configurations of Hadoop programs. Wang *et al.* [27] built a Spark program performance prediction model by binary classification and multi-classification. In the ML-based approach, we only need to consider the configuration and execution time of the workload, ignoring the details of the internal running processes. Therefore, this method can be used for parameter-tuning for several frameworks. However, to obtain an accurate performance prediction model, a significant amount of training data needs to be collected to train the model, which is very time-consuming. ATCS differs from these studies because it can obtain a high-precision performance prediction model with a small amount of training data. Moreover, ATCS facilitates migrating between different systems.

## III. BACKGROUND

### A. OVERVIEW OF APACHE SPARK

Apache Spark [28] is a fast and versatile computing engine designed by UC Berkeley AMP Lab for big

data processing. It is an open-source general parallel framework that resembles Hadoop. In Spark, *Resilient Distributed Dataset* (RDD) [29] has been proposed, that can reduce a large number of disk IO operations using iterative computations. Spark implements a fault-tolerant mechanism based on lineage. The transformation relationship of RDDs constitutes a *Directed Acyclic Graph* (DAG) [29], which is the lineage that evolves between RDDs. If part of the calculation result is lost, users only need to recalculate based on this lineage. Spark is more suitable for data mining, machine learning and other iterative intensive MapReduce algorithms. To support more big data applications and take full advantage of the resources of the cluster, Spark provides high-dimensional configurable parameters.

In Spark, since most of the calculations are done in memory, the bottlenecks of the program may be any resource in the cluster, such as CPU, network bandwidth, or memory. Therefore, the parameter tuning for Spark is mainly for configuration parameters that are playing important roles in the performance. Generally, parameters playing important roles in the performance of Spark programs are shuffle-related, storage-related, schedule-related, compression-related, or serialization-related.

For example, shuffle is an operation that has a large impact on Spark performance: *spark.shuffle.spill* and *spark.shuffle.memoryFraction* are two shuffle-related parameters. In the process of shuffle, if there are operations such as sorting and aggregation, several data structures need to be maintained in memory, which will use additional memory. If the memory is not sufficient, some data needs to be temporarily written to the external storage device and merged into the final shuffle output file; otherwise, a JVM OOM error occurs. The parameter *spark.shuffle.spill* determines whether to spill to an external storage device. If *spark.shuffle.spill* is true, when the proportion of memory used in the shuffle process out of the total memory exceeds the value of *spark.shuffle.memoryFraction*, the data in the memory begins to spill to the disk. Spill provides additional disk operations, and the value of *spark.shuffle.memoryFraction* can adjust the frequency of spill and the behavior of the GC.

Spark supports several big data applications, providing powerful libraries including SparkSQL, MLlib, GraphX, Spark Streaming, and so on. However, different applications have different requirements for cluster resources. Therefore, developers can only provide configurable parameters from which users can select. For example, the Spark Streaming program requires the job to be processed in seconds or milliseconds. It requires less memory but more CPU processing. We can set *spark.executor.cores* to increase the number of CPU cores. We can also adjust *spark.default.parallelism* to ensure that the number of parallel tasks is sufficient to fully use cluster resources. In contrast, many machine learning applications do not consider the time span of the job but require lots of memory to hold the intermediate results of the iterative calculations, thus requiring alternative parameter tuning strategies.

In Spark, different applications have different requirements for parameter configuration. Even in the same application, if the size of processed data is different, then the required parameter configuration is different.

### B. GENERATIVE ADVERSARIAL NET

GAN [10] was proposed by Ian Goodfellow in 2014. The idea of GAN comes from game theory, which is jointly improved in the mutual game process and finally reaches the Nash equilibrium [31]. As shown in Fig. 3, GAN consists of two components: the generator $G$, which is responsible for generating samples, and the discriminator $D$, which is responsible for identifying whether the sample is a fake sample (*generated by the generator*) or a real sample. The optimized target function of a GAN is shown in Equation 1 [10], where $D(x)$ is the probability that $x$ derived from the real samples rather than from $G$, $z$ represents a random noise variable, and $G(z)$ is a differentiable function represented by the generator.

During the GAN training process, we train $D$ to maximize $D(x)$ to enable discriminator $D$ to identify fake samples generated by $G$ as frequently as possible. We synchronously train $G$ to minimize $log(1-D(G(z)))$; the goal of $G$ is to generate fake samples similar to real samples to deceive $D$. The process of training $G$ and $D$ forms a dynamic ''gaming process'' and the final equilibrium point is the Nash equilibrium point. Compared with the neural network model, GAN has two different networks instead of a single network, and the training process adopts the confrontation training method; the gradient update information of $G$ in GAN comes from the discriminator $D$, which provides much stronger gradients early in learning.

$$\min_G \max_D V(D, G) = E_x \sim p_{data(x)}[\log D(x)] + E_z \sim p_{z(z)}[\log(1 - D(G(z)))] \tag{1}$$

When GANs was first proposed, several challenges needed to be solved by researchers. First, the process of training GAN is too flexible and without guidance, resulting in training that may not reach the Nash equilibrium. To solve this problem, the most straightforward method is to provide the generator clues that can help it achieve goals. For example, Conditional GAN [32] was proposed based on the original GAN: for the generator $G$, additional information is added in the generation process, which is equivalent to providing hints to G to generate high-quality samples.

During the GAN training process, gradient vanishing and mode missings issues are likely. The reason for the gradient vanishing issue is that in the min-max game of $G$ and $D$, the optimization objective function is equivalent to optimizing the *Jensen-Shannon* (JS) divergence [39]. However, if the distribution of the generated and training data differ greatly, the JS divergence is a constant, and the optimization objective function is also a constant, at which point gradient vanishing occurs. Using JS and *Kullback-Leibler* (KL) divergence [40] to measure the distribution of generated data and the distribution of training data will also cause a mode missing issue. Wasserstein GAN [33] replaces the JS divergence with

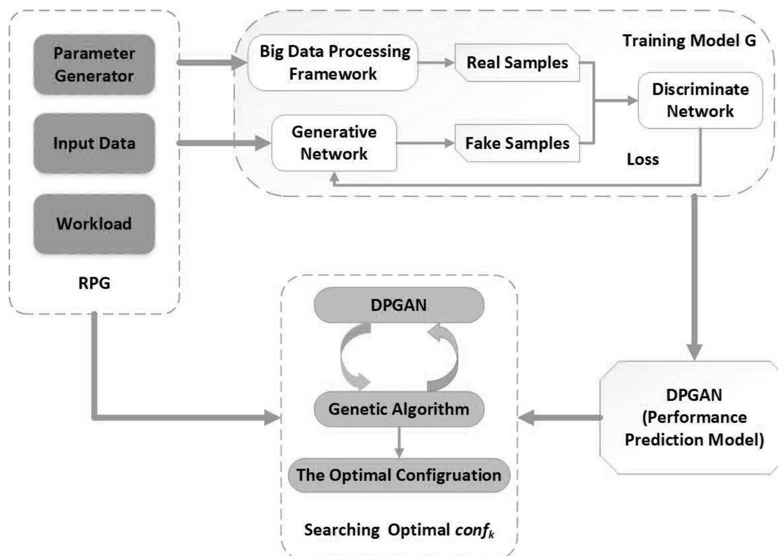



**FIGURE 2. The ATCS framework.**

Earth-Mover Distance and has solved the gradient vanishing issue and mode missing issue.

The advantage of GAN is the ability to generate false data that appears similar to real data using random noise. If the generated fake data is close enough to the real data, the generated data can be used instead of the real data. In many real-world environments, collecting real data is time-consuming and labor-intensive, and we can use GAN to generate simulation data to save costs. For example, during training a machine learning model, collecting and cleaning training data will take up most of our time. At this point, we can use GAN to create a simulator to generate training data.

In ATCS, we modified the structure of the generator in GAN and used it as a performance prediction model. We replaced the random noise of the input generator with the configuration parameters. The time was recorded from the generated data as a prediction result of the prediction model.

## IV. DETAIL OF DESIGN

ATCS is a method for automatically tuning the parameters of a general big data processing framework on a given cluster. The framework of ATCS illustrated in Fig. 2 consists of three functional components: a RPG, PPGAN, and a searching model. The subsequent content will introduce the detailed design and functionality of PPGAN and the searching model, and the RPG will be introduced in the implementation section.

### A. PERFORMANCE PREDICTION USING GENERATIVE ADVERSARIAL NETS

In this section, we will detail how to create a performance prediction model using GANs. The model's main function is to predict the corresponding performance value for any parameter configuration on a given cluster. This performance value is usually expressed by the execution time of the job. Equation 2 illustrates the definition of the performance value, where $r$ represents the hardware and software resources of the computing platform, $d$ represents the size of the data set, $w$ represents the workload, $conf$ represents the parameter configuration, and $t$ represents the execution time of jobs. In ATCS, $r$ is a constant, and we study the effect of performance on different workloads $w_m$ $(m = 1, 2, \ldots, n)$ with different parameter configurations $conf_i$ $(i = 1, 2, \ldots, n)$ and data set sizes $d_j$ $(j = 1, 2, \ldots, n)$.

$$t = f(r,\ d,\ w,\ conf) \tag{2}$$

In the performance model, there are many selectable algorithms, such as the common GAN, ANN [12], SVM [13], and *Regression Tree* (RT) [14]. When training these four machine learning models, we choose the commonly used loss function MSE. These algorithms have advantages and disadvantages, we primarily consider two aspects: the accuracy of the model and the cost of training the model.

In ATCS, we use a GAN to build the performance prediction model. This is determined by the elegant design structure of GAN, which is designed to transform a complex regression problem into two simple two-class problems. Therefore, it is possible to use minimal training data while achieving high precision. The framework of a GAN illustrated in Fig. 3, reveals a training method that differs from other neural networks. The update of $G$ is obtained from the back-propagation gradient of $D$. The principle training process of GAN was introduced in Section III.B, so here we will introduce how to apply GAN to the performance prediction model of the big data processing framework.

First, we use RPG to automatically generate a configuration $conf_k$ and then place $conf_k$ into the generator $G$.

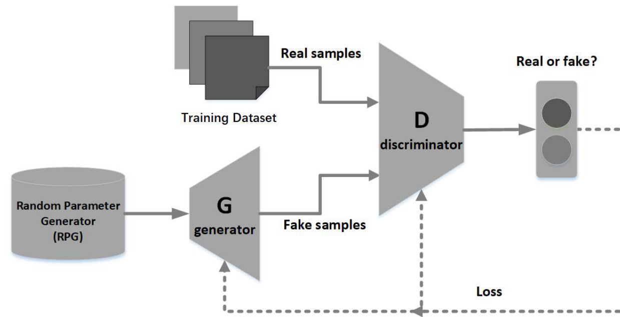



**FIGURE 3.** Performance prediction model framework.

**Algorithm 1** Mutation

**Input:** A population $P = \{conf_1, conf_2, \ldots, conf_k\}$, mutation probability threshold PM

**Output:** A new population $P' = \{conf_1{}', conf_2{}', \ldots, conf_k{}'\}$

1: len = length(*conf*);
2: define a constant C;
3: **for** each $conf \in P$ **do**
4: define a random $i \in [0, len]$;
5: define a random variable R;
6: **if** $R < PM$ **then**
7: **if** $conf[i] + C$ does not exceed the cluster resource range **then**
8: conf[i]+ =C;
9: **end if**
10: **end if**
11: **end for**
12: **return** $P$

During the confrontation training process for the model, the role of $G$ is to continuously predict the execution time of $conf_k$, then join the execution time $t$ and $conf_k$ becoming fake samples. The role of the discriminator $D$ is to continuously identify whether the input samples are real samples or fake samples. As the training progresses, when the GAN reaches equilibrium, $D$ cannot determine whether a sample is from the real sample set or the fake sample generated by $G$. At this point, the fake samples generated by $G$ can replace the real samples, resulting in a configuration $conf_k$ that can be arbitrarily placed into $G$ which can accurately predict its execution time $t$. Finally, $G$ can be used as our performance prediction model.

## B. SEARCHING MODEL

The big data processing framework has a large parameter space; for example, Spark has more than 180 parameters. It is impossible to enumerate configurations in the application because of a large number of parameters and parameter ranges that vary with the hardware environment. Therefore, a searching model component is required that uses a heuristic algorithm to explore the optimum solution in the parameter space.

Common heuristic search algorithms include: the GA, the *hill climbing algorithm* (HC) [34], the *simulated annealing algorithm* (SA) [35], the *ant colony optimization* (ACO) [36], and the *recursive random search* (RRS) algorithm [37]. The HC algorithm is simple to implement, but it is prone to arriving at the local optimal solution. The SA algorithm parameters are difficult to control, with no guarantee that they will converge to the optimal value simultaneously, generally requiring multiple attempts. The ACO algorithm requires significant calculation and time to solve. The RRS algorithm runs inefficiently, takes significant time to complete, and requires a large memory space.

In ATCS, the optimized GA is used to explore the optimum solution in Spark parameter space. We use prior knowledge of parameter tuning to optimize key operations in traditional genetic algorithms, such as crossover and mutation operations, to improve efficiency of searching for optimal configurations. A GA is a computational model that simulates the "*survival of the fittest*" mechanism in the process of biological evolution, and it is often used to search for the global optimal solution in other fields. In GA, individuals in a population can be calculated and compared simultaneously, and this potential parallelism enables GA to converge rapidly. A GA consists of three core operations: selection, crossover and mutation. All these core operations use the probability mechanism to perform the "*survival of the fittest*", so GA has good randomness, and it is not easy to fall into a local optimum. Moreover, a GA uses an evaluation function to inspire the search, that makes GA simple and robust.

In the mutation operation, we change the random mutation operation in the traditional genetic algorithm to linear addition operation within the range of available resources. The pseudo code of the mutation operation is presented in Algorithm1. In lines 6 to 8, if $conf[i]+C$ does not exceed the cluster resource range, the mutation operation will proceed in the direction of increasing resources. This can increase the subset of resources contained in each configuration, thereby improving system performance.

As presented in Algorithm 2, we will describe how to use an optimized GA to find the optimal parameter configuration. We randomly generate several parameter configurations and initialize the population $P = \{conf_1, conf_2, \ldots, conf_k\}$. The performance prediction model PPGAN can calculate the fitness value (*the execution time corresponding to each parameter configuration*) of each individual (*parameter configuration*) in the population, and then through the selection, crossover and mutations operation in the genetic algorithm to obtain a new population $P' = \{conf_1{}', conf_2{}', \ldots, conf_k{}'\}$. We then recalculate the fitness value of each individual in the new population $P'$, and continue selection, crossover, and mutation operations again. We loop this process until the optimal parameter configuration $C_i = \{c_{i1}, c_{i2}, \ldots, c_{in}\}$ that satisfies the condition is found. Throughout the execution of the algorithm, we need to determine the size of the population and the number of cycles based on prior knowledge and

**Algorithm 2** The Optimized Genetic Algorithm

**Input:** A population $P = \{conf_1, conf_2, \ldots, conf_k\}$, mutation probability threshold $PM$, crossover probability threshold $PC$, number of cycles $Count$.

**Output:** The optimal configuration $C_i = \{c_{i1}, c_{i2}, \ldots, c_{in}\}$

1: initial population $P = \{conf_1, conf_2, \ldots, conf_k\}$;
2: loading performance prediction model $PPGAN$;
3: calculate the fitness value of $P$;
4: $l = 0$
5: **while** $l < Count$ **do**
6: define random variables $C, M$;
7: selection;
8: **if** $C < PC$ **then**
9: crossover;
10: **end if**
11: **if** $M < PM$ **then**
12: mutation;
13: **end if**
14: A new population $P' = \{conf_1{}', conf_2{}', \ldots, conf_k{}'\}$
15: calculate the fitness value of $P'$;
16: $l = l + 1$;
17: **end while**
18: **return** $C_i = \{c_{i1}, c_{i2}, \ldots, c_{in}\}$

actual situations, balancinge time consumption and search performance.

## V. IMPLEMENTATION

In this section, we introduce the implementation details of ATCS on Spark 2.2.1. To improve our code developing efficiency and make full use of third-party open-source toolkits, in ATCS, we use Python to implement functional components. Moreover, PPGAN is implemented with Keras because it facilitates the design and construction of neural network models.

### A. PARAMETERS SELECTION

Spark has more than 180 parameters, but not all of them have an impact on performance, such as, *spark.app.name*, which sets the application name and has no effect on the performance of the application. There is no need to consider such parameters when turning parameters. In ATCS, by counting the frequency of each Spark parameter, we select 20 frequently-used parameters that have a large impact on performance, as presented in Table 2. These selected parameters have an impact on shuffle operations, compression operations, serialization operations, runtime behavior, network, and scheduling.

For example, *spark.executor.memory* is used to set the memory of each Executor process. The size of the Executor memory often determines the performance of Spark jobs and is directly related to common JVM OOM error. The *spark.reducer.maxSizeInFlight* is used to set the buffer size of the shuffle read task, and this buffer determines how much data can be pulled each time. The *spark.default.parallelism* is also an important parameter for setting the default number of tasks for each phase. If it is not set, the performance of Spark jobs may be affected. A common mistake made by many Spark users is not setting this parameter. When this happens, Spark will set the number of tasks according to the number of blocks of the underlying HDFS. The default is that a HDFS block corresponds to a task. Generally speaking, the default number of Spark settings is too small. Thus, the parameters of the Executor set previously will be discarded. No matter how many Executor processes exist and how large the memory and CPU are, whether one or ten tasks, 90% of the Executor process may not be executing tasks, which is a waste of resources.

### B. RANDOM PARAMETER GENERATOR

The role of the RPG is to collect enough training data for the performance prediction model. To accurately learn the performance variation characteristics of the same or different workloads under different configurations, PPGAN needs to collect abundant training data. Collecting training data is a time-consuming and labor-intensive process. For the same workload, to ensure the model accuracy, it is often necessary to collect execution times from hundreds or even thousands of different configurations. The RPG includes the *Parameter Generator* (PG) and *Input Data Size*.

PG: A typical big data processing framework, such as Spark or Hadoop, has a large parameter space. It is impossible for us to enumerate the parameters of each group. Therefore, we can use the random sampling method to generate random parameter configuration $C_i = \{c_{i1}, c_{i2}, \ldots, c_{in}\}$ in the actual range of hardware resources, with each group $C_i$ containing the $n$ parameters. For each workload, each $C_i$ can obtain a corresponding performance value $t_i$. To train the model, it is necessary to generate $m$ configurations for each workload, as presented in Equation 3, and collect the execution time of each workload on the given cluster with these configurations.

$$P_j = \{C_1,\ C_2,\ \ldots,\ C_m\} \quad j = 1, 2, \ldots,\ m \tag{3}$$

*Input Data Size*: the purpose of this component is to provide the size of each job input dataset. The input data size dose not belong to a configurable parameter of the big data processing framework. However, in automatic parameter tuning the input data size has the greatest impact on the performance of the workload. For instance, in Spark, the input dataset size is one of the foremost factors affecting performance because Spark is extremely sensitive to the dataset size. We define a configuration as in Equation 4, which consists of $P_k$ and $S_k$. $P_k$ represents the parameters generated by PG, and $S_k$ represents the size of the dataset.

$$conf_k = \{P_k,\ S_k\} \quad k = 1,\ 2,\ \ldots,\ k \tag{4}$$

## VI. EXPERIMENT

### A. EXPERIMENTAL ENVIRONMENT

In our experiments, we used three nodes to build a Spark cluster environment: one master node and two slaves nodes.

**TABLE 2.** 20 Spark configurable parameters we selected in ATCS and their descriptions.

| Configurable parameters and their descriptions | Default value | Range |
|---|---|---|
| $spark.driver.memory$: the parameter is used to set the amount of memory in the driver process. | 1GB | 1–60GB |
| $spark.driver.cores$: the parameter is used to set the number of cores in the driver process. | 1 | 1–16 |
| $spark.executor.memory$: the parameter is used to set the amount of memory to be used by each executor. | 1GB | 1–60GB |
| $spark.executor.cores$: the parameter is used to set the number of cores used on each executor. | 1 | 1–16 |
| $spark.default.parallelism$: the parameter is used to set the default number of tasks for each stage. | 32 | 1–1000 |
| $spark.storage.memoryFraction$: the parameter is used to set the percentage of RDD persistent data memory in Executor memory. | 0.6 | 0–1 |
| $spark.shuffle.memoryFraction$: the parameter is used to set the proportion of memory allocated to the shuffle read task for aggregation operations in the Executor memory. | 0.2 | 0–1 |
| $spark.driver.maxResultSize$: the parameter is used to set the threshold of the total size of the serialization results of all partitions for each Spark action . | 1GB | 0–61GB |
| $spark.broadcast.compress$: the parameter is used to decide to compress broadcast variables or not. | TRUE | TRUE or FALSE |
| $spark.reducer.maxSizeInFlight$: the parameter is used to set the buffer size of the shuffle read task, and this buffer determines how much data can be pulled each time. | 48MB | 1–1024MB |
| $spark.broadcast.blockSize$: the parameter is used to set size of each piece of a block for Torrent BroadcastFactory, in KB unless otherwise specified. | 4MB | 1–64MB |
| $spark.shuffle.io.maxRetries$: the parameter is used to set the maximum number of automatic retry that can be made if the shuffle read task fails to pull data. | 3 | 0–30 |
| $spark.shuffle.io.retryWait$: the parameter is used to set the time interval for each retry if the shuffle read task fails to pull data. | 5s | 0–30s |
| $spark.shuffle.service.index.cache.size$: the parameter is used to limit the cache entries to the specified memory footprint. | 100MB | 0–1024MB |
| $spark.shuffle.compress$: the parameter is used to decide to compress shuffle output files or not. | TRUE | TRUE or FALSE |
| $spark.shuffle.spill.compress$: if data is spilled to disk during the shuffle process, the parameter determines whether to compress the spilled data. | TRUE | TRUE or FALSE |
| $spark.rdd.compress$: the parameter is used to decide to compress RDDs or not, and the compression operation can save lots of memory, but it will consume more CPU. | FALSE | TRUE or FALSE |
| $spark.io.compression.codec$: the parameter specifies the compression algorithm used when compressing. | lz4 | lz4, lzf, snappy |
| $spark.speculation$: the parameter determines whether the speculative execution mechanism of Spark is turned on. | FALSE | TRUE or FALSE |
| $InputDataSize$: the input data size. | NULL | NULL |

Each node is equipped with an Intel Xeon CPU E5-26700 2.60GHz 16-cores processor and 64 GB memory. There are 48 cores and 192GB of memory in the cluster. The OS version is Red Hat Enterprise Linux Server release 6.2 (*Santiago*). In the cluster, the big data processing framework is Spark 2.2.1, which covers four common applications in the big data domain, including machine learning, graphics computing, SQL queries, and Spark streaming. We take the SPARK-BENCH [38] as the benchmark of our experiment.

**TABLE 3.** Workloads and data size.

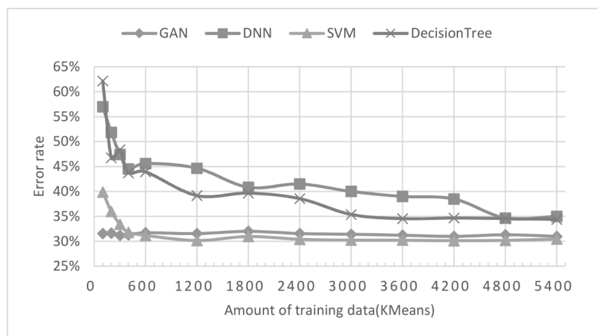
| Workload | Data size |
|---|---|
| KMeans | 1.0, 5.0, 10.0, 20.0, 160.0 (M points) |
| PageRank | 0.5, 0.6, 1.2, 1.4, 1.6 (M pages) |
| TriangleCount | 10, 20, 40, 50, 100 (K points) |
| MatrixFactorization | 0.36, 0.49, 0.57, 0.69, 0.72 (M points) |
| TeraSort | 1.0, 2.0, 3.0, 4.0, 5.0 (GB) |

### B. WORKLOAD AND ERROR RATE

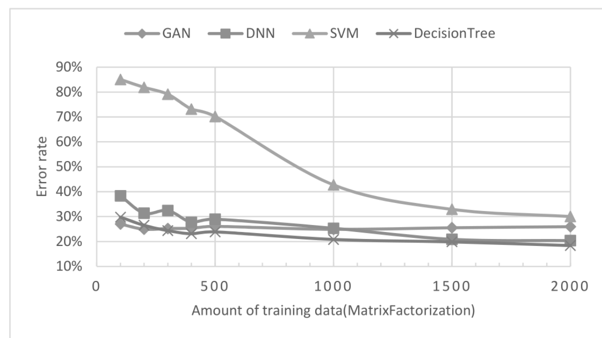
To observe the performance of Spark, we collect execution times of different workloads in different configurations and different input dataset sizes. As presented in Table 3, in ATCS we collected training data for five different workloads, each with five different input dataset sizes. These workloads are extensively used to assess the performance of the Spark framework, and they are provided by the SPARK-BENCH platform. For example, PageRank is a memory- and CPU-intensive operation, with many shuffle operations while running on Spark.

For the predictive performance model, we desire that the predicted value is close to or even equal to the true value, but this is not supported. In production practice, due to the limitations of the algorithm model itself, the predicted value we obtain tends to have a certain distance from the true value. To quantify this distance, we define the error rate as presented in Equation 5:

$$err = \frac{|t_{pre} - t_{real}|}{t_{real}} \times 100\% \tag{5}$$

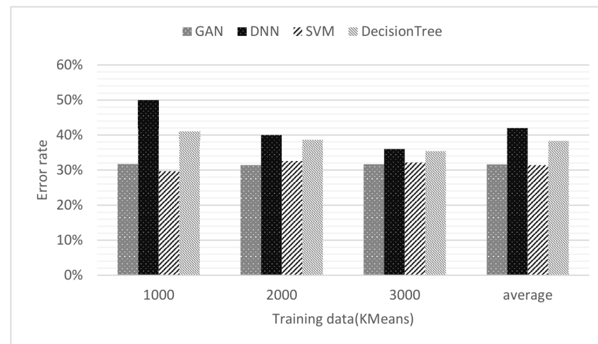



(a) Error rate varies comparison (KMeans)

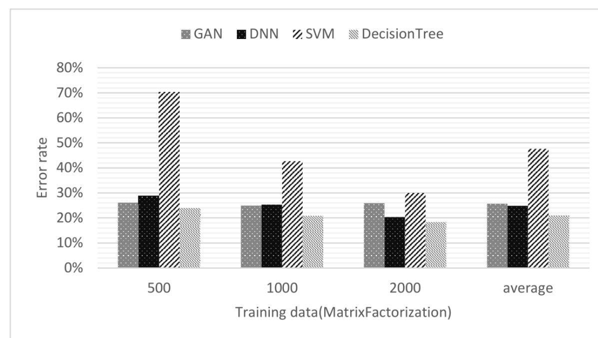



(b) Error rate varies comparison (MatrixFactorization)

**FIGURE 4. Error rate varies comparison.**




(a) Average prediction error (KMeans)




(b) Average prediction error (MatrixFactorization)

**FIGURE 5. The average prediction error of models built by GAN, ANN, SVM, and DT.**

In Equation 5, $t_{pre}$ is the predicted execution time of a job, $t_{real}$ is the actual execution time of the job, and $err$ represents the relative difference between the two. The smaller the value of $err$, the closer the predicted execution time is to the actual execution time, thus, the more accurate the prediction model.

### C. RELATIONSHIP BETWEEN TRAINING DATA VOLUME AND MODEL ERROR RATE

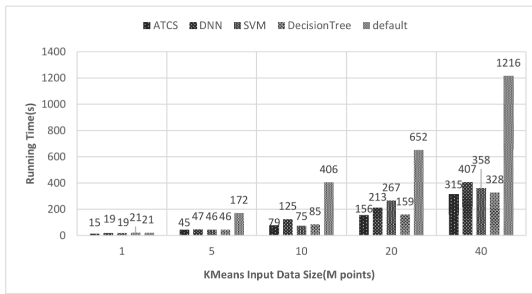
Fig. 4 illustrates the variation of the predictive model error rate with the amount of training data. As illustrated in Fig. 4 (a), for the KMeans workload, the error rates of the performance prediction models built by DNN, SVM, and DT gradually decrease as the amount of training data increases. When the training data reaches 4800, the error rates of these models becoming stabilize. The error rate of the model built with GAN is stable between 31% and 32% with training data between 100 and 5400 sets. The performance model built by DNN and DT uses dozens of times the amount of training data, but the accuracy of the model obtained is lower than the performance prediction model built by GAN. Using the performance model built by SVM, to obtain a model with comparable accuracy, the amount of training data is also six times that of GAN. Comparing Fig. 4 (a) and Fig. 4 (b), the model built by SVM in Fig. 4 (a) performs better but is worse in Fig. 4 (b). In contrast, DT, which performs poorly in Fig. 4 (a), performs well in Fig. 4 (b). The performance of DNN is worse than GAN in both Fig. 4 (a) and Fig. 4 (b). Therefore, comparing the models built by the four algorithms, we conclude that the performance of the model built with GAN is excellent and relatively stable supporting our choice.

### D. MODEL ACCURACY AND ROBUSTNESS

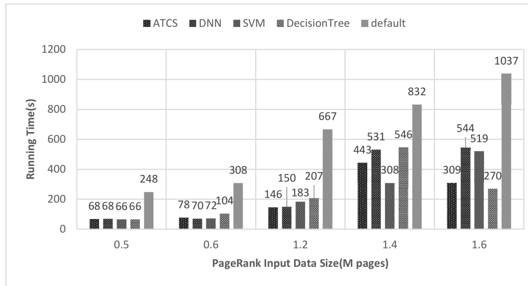
Accuracy and robustness of the performance prediction model are important indicators to consider. As illustrated in Fig. 5, we tested the error rate of the performance model built by GAN, DNN, SVM, and DT for training data quantities of 1000, 2000, and 3000. As illustrated in Fig. 5 (a), the average error rate of the performance prediction model built by GAN is 31%, which is lower than the average error rate of 42% and 38% of the performance model built by DNN and DT and equivalent to that of SVM. However, from the experimental data in Fig. 5 (a), when the training data volume is less than 600, the performance of GAN is significantly better than that of SVM. Fig. 5 demonstrates again that using GAN to build a performance prediction model, we can train a higher-precision model with a small amount of training data. As shown in Fig. 5, the prediction performance of GAN is more stable than DNN, SVM, and DT.

### E. PERFORMANCE IMPROVEMENT

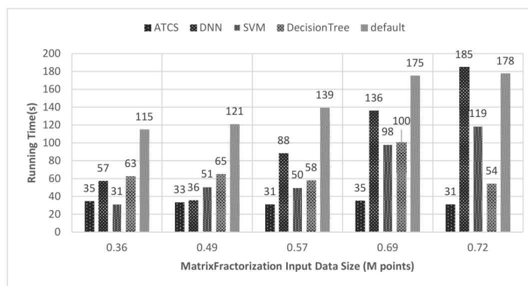
Fig. 6 shows the increases of five different workloads with ATCS over DNN, SVM, DT, and default configurations. As illustrated in Fig. 6, ATCS dramatically improves the performance of five different workloads (*KMeans, PageRank, MatrixFactorization, TriangleCount, and TeraSort*)



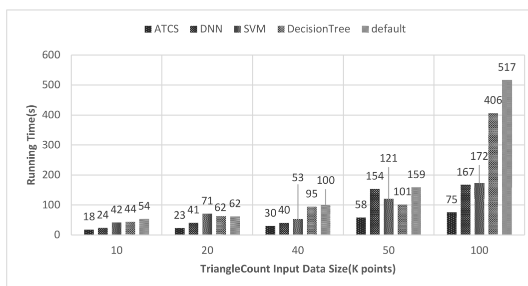

(a) Increase comparison for KMeans



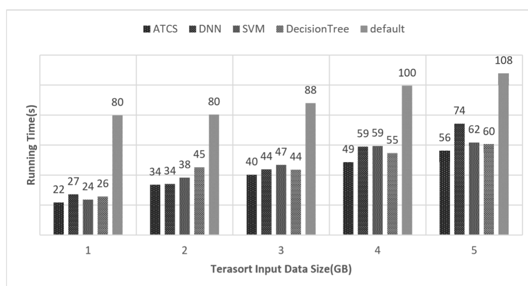

(b) Increase comparison for PageRank




(c) Increase comparision for MatrixFactorization




(d) Increase comparision for TriangleCount




(e) Increase comparision for TeraSort

**FIGURE 6.** **Increase comparison.**

compared to default configurations. We achieved a performance increase of 3.5× on average, with a maximum of 6.9×. This indicates that the default configuration performance is poor for common Spark applications because the resources of the Spark cluster cannot be fully utilized. For instance, the default value of *spark.executor.cores* is 1, and in our experiment, the total number of cores for a slave node is 16. If the default configuration is used when submitting jobs to the cluster, the remaining 15 cores will not be used. Therefore, the resources of the cluster are largely wasted.

As illustrated in Fig. 6 (a), the optimal configuration found by ATCS performs best on KMeans compared to DNN, SVM, and DT. The increase of ATCS over DNN is 1.3× on average and up to 1.6×. The increase of ATCS over SVM is 1.2× on average and up to 1.7×. The increase of ATCS over DT is 1.1× on average and up to 1.4×.

Fig. 6 (b) reports the performance improvements of the four models on PageRank. As expected, we use ATCS for parameter tuning to improve the performance of PageRank over the other three methods. The increases of ATCS over DNN, SVM, DT are 1.2×, 1.1×, and 1.2× on average.

Fig. 6 (c) illustrates the performance improvement of the four models on MatrixFactorization. As expected, DNN, SVM, and DT do not perform well. ATCS achieves an average of 3.1×, 2.1×, and 2.1× increase over DNN, SVM, and DT. The maximum increase is 5.9 ×.

Fig. 6 (d) illustrates the performance improvement of TriangleCount tuned by ATCS against that tuned by DNN, SVM, and DT. The increases of ATCS over DNN, SVM, and DT are 1.9×, 2.3×, and 3.1× on average.

As illustrated in Fig. 6 (e), on the workload TeraSort, ATCS achieves an average of 1.2×, 1.1×, and 1.2× increases over DNN, SVM, and DT.

Moreover, as illustrated in Fig. 6, the larger the input data set of different workloads, the more prominent the contribution of ATCS because there is a limit to the performance optimization of the Spark framework through parameter tuning. This upper limit is the theoretical minimum execution time for processing a job on the Spark framework. This shortest time contains the overhead of the Spark framework itself. When the input data volume of the workload is small, the overhead of the Spark framework itself accounts for a large proportion, and the parameter tuning effect is not obvious. In contrast, when the input data volume of the workload is large, the overhead of the Spark framework itself only accounts for a small part of the total execution time, and the parameter tuning effect is more obvious.

## VII. CONCLUSION

In this paper, we present ATCS, a system for automatic tuning of parameters in big data processing frameworks. To reduce the training data volume and improve the accuracy of the performance prediction model, we propose a performance prediction model based on GAN. Moreover, to collect enough

training data, we propose a RPG that can randomly generate configurations within a given cluster resource. Finally, we use an optimized GA to search for optimal configurations in the parameter space. Extensive experiments over ATCS confirm that it can reduce the amount of training data, and we achieve on average a 3.5× and up to a 6.9× performance increase compared to the default configurations. Finally, compared to other machine learning models, the average performance increase of ATCS is 1.7 × of DNN, 1.6 × of SVM, and 1.7 × of DT on the four typical Spark programs.

## VIII. DISCUSSION AND FUTURE WORK

In this paper, we have only implemented ATCS on Spark. This is because ATCS is essentially framework-independent, and it is applicable to any big data framework that requires parameter tuning. When implemented on other big data frameworks, such as Hadoop, we only need to reselect the parameters that need to be tuned, and then repeat the process of collecting training data, training the model, and searching for the optimal parameter configuration.

In the future, we will explore methods to promote the accuracy of GAN prediction and aim to study the migration of models between different workloads. This will further reduce the overhead of collecting training data and help us build a more lightweight automatic tuning system.

## REFERENCES


[1] L. Wang, S. Tasoulis, T. Roos, and J. Kangasharju, "Kvasir: Scalable provision of semantically relevant Web content on big data framework," *IEEE Trans. Big Data*, vol. 2, no. 3, pp. 219–233, Sep. 2016.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. HotCloud*, 2010, p. 10.

[3] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*. [Online]. Available: http://arxiv.org/abs/1603.04467

[4] D. Sun, G. Zhang, and W. Zheng, "Big data stream computing: Technologies and instances," *J. Softw.*, vol. 25, no. 4, pp. 839–862, 2014.

[5] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proc. ASPLOS*, 2018, pp. 564–577.

[6] A. Jacob, B. Harris, J. Buhler, R. Chamberlain, and Y. Cho, "Scalable softcore vector processor for biosequence applications," in *Proc. 14th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 2006, pp. 295–296.

[7] Z. Fadika and M. Govindaraju, "DELMA: Dynamically ELastic MapReduce framework for CPU-intensive applications," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2011, pp. 454–463.

[8] L. Bao, X. Liu, Z. Xu, and B. Fang, "AutoConfig: Automatic configuration tuning for distributed message systems," in *Proc. 33rd ACM/IEEE Int. Conf. Automat. Softw. Eng. (ASE)*, Sep. 2018, pp. 29–40.

[9] A. Caprara, M. Monaci, P. Toth, and P. L. Guida, "A Lagrangian heuristic algorithm for a real-world train timetabling problem," *Discrete Appl. Math.*, vol. 154, no. 5, pp. 738–753, Apr. 2006.

[10] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. NIPS*, 2014, pp. 2672–2680.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[12] S. Agatonovic-Kustrin and R. Beresford, "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research," *J. Pharmaceutical Biomed. Anal.*, vol. 22, no. 5, pp. 717–727, Jun. 2000.

[13] M. E. Mavroforakis and S. Theodoridis, "A geometric approach to support vector machine (SVM) classification," *IEEE Trans. Neural Netw.*, vol. 17, no. 3, pp. 671–682, May 2006.

[14] C. Zheng, V. Malbasa, and M. Kezunovic, "Regression tree for stability margin prediction using synchrophasor measurements," *IEEE Trans. Power Syst.*, vol. 28, no. 2, pp. 1978–1987, May 2013.

[15] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "BestConfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proc. SoCC*, 2017, pp. 338–350.

[16] S. Kumar, S. Padakandla, L. Chandrashekar, P. Parihar, K. Gopinath, and S. Bhatnagar, "Scalable performance tuning of Hadoop MapReduce: A noisy gradient approach," in *Proc. CLOUD*, 2017, pp. 375–382.

[17] Y. Zhu, J. Liu, M. Guo, W. Ma, and Y. Bao, "ACTS in need: Automatic configuration tuning with scalability guarantees," 2017, *arXiv:1708.01349*. [Online]. Available: http://arxiv.org/abs/1708.01349

[18] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE: MapReduce online performance tuning," in *Proc. HPDC*, 2014, pp. 165–176.

[19] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "MRTuner: A toolkit to enable holistic optimization for mapreduce jobs," in *Proc. VLDB*, 2014, pp. 1319–1330.

[20] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, "RFHOC: A random-forest approach to auto-tuning Hadoop's configuration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1470–1483, May 2016.

[21] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *J. VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[22] C. Liu, D. Zeng, H. Yao, C. Hu, X. Yan, and Y. Fan, "MR-COF: A genetic MapReduce configuration optimization framework," in *Proc. ICAPP*, 2015, pp. 344–357.

[23] J. Berral, N. Poggi, D. Carrera, A. Call, R. Reinauer, and D. Green, "ALOJA-ML: A framework for automating characterization and knowledge discovery in Hadoop deployments," in *Proc. SIGKDD*, 2015, pp. 1701–1710.

[24] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[25] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, and S. Feng, "Configuring in-memory cluster computing using random forest," *Future Gener. Comput. Syst.*, vol. 79, pp. 1–15, Feb. 2018.

[26] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema, "Towards machine learning-based auto-tuning of mapreduce," in *Proc. MASCOTS*, 2013, pp. 11–20.

[27] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *Proc. HPCC/SmartCity/DSS*, 2016, pp. 586–593.

[28] *Apache Spark*. Accessed: Sep. 5, 2019. [Online]. Available: http://spark.apache.org/

[29] M. Zaharia, M. Chowdhury, T. Das, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. NSDI*, 2012, pp. 15–28.

[30] G. Mackey, S. Sehrish, and J. Wang, "Improving metadata management for small files in HDFS," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, Sep. 2009, pp. 1–4.

[31] Q. Fan, K. Zeitouni, N. Xiong, Q. Wu, S. Camtepe, and Y. Tian, "Nash equilibrium-based semantic cache in mobile sensor grid database systems," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 9, pp. 2550–2561, Sep. 2017.

[32] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014, *arXiv:1411.1784*. [Online]. Available: http://arxiv.org/abs/1411.1784

[33] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," 2017, *arXiv:1701.07875*. [Online]. Available: http://arxiv.org/abs/1701.07875

[34] C. Hwang and S. Lin, "Hill climbing for diversity retrieval," in *Proc. CSIE*, 2009, pp. 154–158.

[35] L. Wang, S. Li, F. Tian, and X. Fu, "A noisy chaotic neural network for solving combinatorial optimization problems: Stochastic chaotic simulated annealing," *IEEE Trans. Syst. Man, Cybern. B, Cybern.*, vol. 34, no. 5, pp. 2119–2125, Oct. 2004.

[36] S. Alupoaei and S. Katkoori, "Ant colony optimization technique for macrocell overlap removal," in *Proc. 17th Int. Conf. VLSI Design*, 2004, pp. 963–968.

[37] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for network parameter optimization," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 3, pp. 44–53, Dec. 2004.
[38] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A comprehensive benchmarking suite for in memory data analytic platform Spark," in *Proc. CF*, 2015, pp. 1–8.
[39] M. L. Menendez, J. A. Pardo, L. Pardo, and M. C. Pardo, "The Jensen–Shannon divergence," *J. Franklin Inst.*, vol. 334, no. 2, pp. 307–318, 1997.
[40] J. R. Hershey and P. A. Olsen, "Approximating the Kullback Leibler divergence between Gaussian mixture models," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2007, pp. 317–320.

**MINGYU LI** received the B.S. degree in computer science and technology from the South China University of Technology, Guangzhou, China, in 2009. He is currently pursuing the M.S. degree in computer technology with the Huazhong University of Science and Technology, Wuhan, China. He is also an Experimentalist with Liupanshui Normal University. His research interests include big data and machine learning.

**ZHIQIANG LIU** received the B.S. and M.S. degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2007 and 2011, respectively. He is currently pursuing the Ph.D. degree in computer science and technology with the Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan, China. His research interests include big data and machine learning.

**XUANHUA SHI** (Senior Member, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 2006. He is currently a Professor with the School of Computer Science and Engineering, HUST. His research interests include parallel and distributed computing, multicore architecture and system software, cloud computing and big data processing, and heterogeneous parallel computing. He is the Deputy Director of the National Engineering Research Center for Big Data Technology and System (NERCBDTS), and also the Deputy Director of Parallel and Distributed Computing Institute.

**HAI JIN** (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology, in 1994. He worked at The University of Hong Kong, from 1998 to 2000, and as a Visiting Scholar at the University of Southern California, from 1999 to 2000. He is currently a Cheung Kung Scholars Chair Professor of computer science and engineering with the Huazhong University of Science and Technology. He is also the Chief Scientist of China Grid, and also the Chief Scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System and Cloud Security. He has coauthored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a Fellow of CCF and a member of the ACM. In 1996, he received the German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany, and the Excellent Youth Award from the National Science Foundation of China, in 2001.