



## Article

# A Novel Reinforcement Learning Approach for Spark Configuration Parameter Optimization

Xu Huang , Hong Zhang \*  and Xiaomeng Zhai

School of Cyber Security and Computer, Hebei University, Baoding 071000, China

\* Correspondence: hzhang@hbu.edu.cn

**Abstract:** Apache Spark is a popular open-source distributed data processing framework that can efficiently process massive amounts of data. It provides more than 180 configuration parameters for users to manually select the appropriate parameter values according to their own experience. However, due to the large number of parameters and the inherent correlation between them, manual tuning is very tedious. To solve the problem of tuning through personal experience, we designed and implemented a reinforcement-learning-based Spark configuration parameter optimizer. First, we trained a Spark application performance prediction model with deep neural networks, and verified the accuracy and effectiveness of the model from multiple perspectives. Second, in order to improve the search efficiency of better configuration parameters, we improved the Q-learning algorithm, and automatically set start and end states in each iteration of training, which effectively improves the agent's poor performance in exploring better configuration parameters. Lastly, comparing our proposed configuration with the default configuration as the baseline, experimental results show that the optimized configuration gained an average performance improvement of 47%, 43%, 31%, and 45% for four different types of Spark applications, which indicates that our Spark configuration parameter optimizer could efficiently find the better configuration parameters and improve the performance of various Spark applications.



**Citation:** Huang, X.; Zhang, H.; Zhai, X. A Novel Reinforcement Learning Approach for Spark Configuration Parameter Optimization. *Sensors* **2022**, *22*, 5930. <https://doi.org/10.3390/s22155930>

Academic Editor: Rebeca P. Díaz-Redondo

Received: 26 June 2022

Accepted: 6 August 2022

Published: 8 August 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

**Keywords:** Apache Spark; parameter optimization; deep neural network; Q-learning

## 1. Introduction

Apache Spark [1] is a widely used open-source data analysis framework that utilizes resilient distributed datasets (RDDs) [2] to improve the efficiency of data processing and analysis while ensuring high fault tolerance and scalability. It provides a series of high-level components, including Spark streaming for real-time computing, Spark SQL for structured data processing, GraphX for graph computing, and MLlib for machine learning [3]. These components are applied by application developers to various fields, such as feature extraction [4], intrusion detection [5], and community discovery [6], and maintain good performance.

Apache Spark has more than 180 configuration parameters that users must manually adjust according to their application and cluster environment. Choosing the appropriate configuration parameters can not only significantly improve the performance of Spark applications and speed up the running time of Spark applications, but also improve the utilization of cluster resources. Unfortunately, some Spark's configuration parameters are numerous, and their inter-relationship are also very complex [7]. Therefore, tuning configuration parameters is a very challenging task.

Manually tuning Spark configuration parameters is cumbersome and time-consuming, and requires developers to have a deep understanding of the Spark framework, which inspired our interest in the automatic tuning of Spark configuration parameters. Generally speaking, automatic Spark configuration parameter tuning consists of two components:

the Spark application performance prediction model and the configuration-parameter-searching algorithm. Performance prediction models render the evaluation process efficient and practical instead of methods of repeatedly executing the application.

Gao et al. [8] employed SVM to build a model to predict the execution time of Spark applications. Rahman et al. [9] built a Spark performance prediction model using ANN. However, SVM, ANN, etc. are all shallow machine-learning methods that do not perform very well when the dataset has more noise or is complex and high-dimensional. In this paper, we introduce a deep-learning neural network to build a Spark performance prediction model. Deep-learning neural networks can gradually learn through multiple networks, extract complex and effective features, and have better prediction accuracy and generalization ability [10].

Recently, some methods based on reinforcement learning to find the optimal neural network model structure have been proposed. MetaQNN [11] applied the Q-learning algorithm to select various layers of a CNN that can automatically generate high-performance CNN architectures for a given task. BlockQNN [12] proposed a Q-learning algorithm to construct an optimal neural network block, and then applied the block-by-block stacking method to automatically construct the network. The above methods were mainly aimed at the construction of the neural network model structure. In this paper, we combine the Spark performance prediction model and the improved Q-learning algorithm to automatically search for optimal Spark configuration parameters in a discrete and limited parameter space. In particular, we make the following contributions in this paper:

- We designed and implemented a Spark configuration parameter optimizer that accurately predicts the execution time of Spark applications and gives the recommended configuration parameters. Experiments demonstrate that the performance of Spark applications using the recommended configuration parameters is significantly improved compared to the default configuration.
- From more than 180 Spark configuration parameters, we screened out key configuration parameters that had greater impact on the application to reduce the complexity of the model.
- We built a Spark application performance prediction model on the basis of a deep neural network, and verified the accuracy and usability of the model from multiple perspectives through experiments.
- On the basis of the Q-learning algorithm, we improved the optimal searching algorithm to explore unknown areas of the configuration parameter space, avoid local optimal solutions, and find a suitable parameter configuration for Spark applications in a short period of time.

The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 introduces the processing flow of the Spark configuration parameter optimizer and the method used in each section. Section 4 introduces the experiments and analyzes the experimental results. Section 5 concludes the paper.

## 2. Related Work

In recent years, the performance optimization of large data processing systems has been a hot-spot academic issue. Big data processing systems contain a large number of configuration parameters such as controlling parallelism, memory settings, and I/O behavior. Inappropriate parameter settings can lead to severe performance degradation [13]. Gounaris et al. [14] mapped their experience in a trial-and-error iterative improvement methodology for tuning parameters in arbitrary applications on the basis of evidence from a very small number of experimental runs. Due to the complexity between parameters, this method requires users to have rich experience in Spark configuration parameter tuning. This paper mainly studies the automatic optimization of Spark configuration parameters, that is, combining the Spark application performance prediction model and search algorithm to effectively traverse the configuration parameter space and give better configuration parameters for the application.

Research on Spark performance prediction models has always been a hot topic. Singhal et al. [15] proposed a gray box approach to estimate an application execution time on the Spark cluster for larger data using measurements on low-volume data in a small cluster. Huang et al. [16] proposed a cost optimization model for the Spark shuffle process that enables users to obtain the best compression configuration before application execution. Wang et al. [17] proposed a novel method for tuning the Spark configuration parameters on the basis of machine learning, and established a model based on binary and multiclassification. Islam et al. [18] modeled the Spark application completion time with respect to the number of executors and Spark application input or iteration. Chao et al. [19] built regression models for different stages during the run, and then built a regression model on the basis of the predicted time of each stage to predict the overall running time of the job. Cheng et al. [3] used Adaboost to build a set of stage performance models for Spark applications. At the same time, using projective sampling reduced the training samples of the performance prediction model and reduced the model overhead. Shah et al. [20] proposed an execution time estimation method called PERIDOT that estimated the dependencies between the internal stages of the application by analyzing the logs of two executions and then combined the knowledge of Spark's data partitioning mechanism to derive an analytical model that estimated the execution time of an application on the basis of resource settings and input data size. However, the above performance prediction model cannot effectively deal with the problem of high-dimensional configuration parameters, and accuracy decreased with the increase in configuration parameters.

Parameter search mainly aims to design a search algorithm to quickly find better configuration parameters from a larger configuration space. Wang et al. [17] adopted the recursive random search (RRS) algorithm to search a parameter space. Gu et al. [21] proposed a neural-network-based configuration tuning approach. In this approach, a neural network model was trained to predict the increase or decrease in configurations that determine the next search space. Ross [22] presented a sample-efficient, high-dimensional autotuner that used Bayesian optimisation with a directed-acyclic graph (DAG) surrogate model. Patanshetti et al. [23] proposed two search algorithms, grid search with finer tuning and controlled random search, which help in selecting those important parameters that affect the performance of Spark applications. We selected the next parameter to be changed by comparing the values in the Qtable, avoiding the search in the invalid configuration parameter space, and improving the search efficiency.

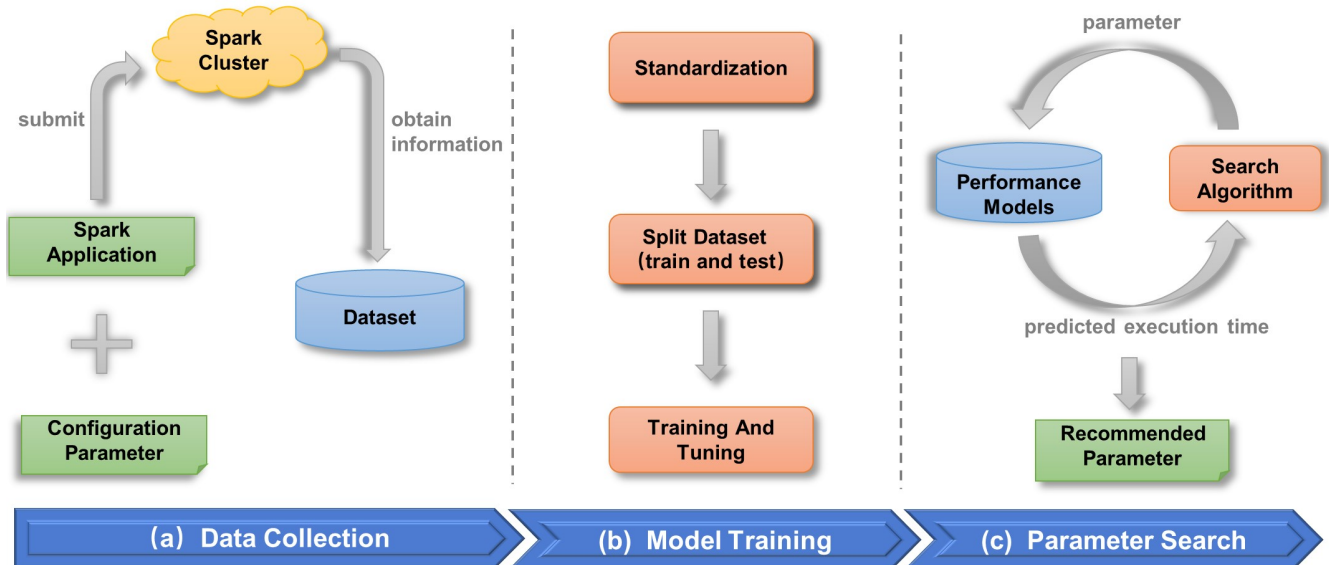
### 3. Methods

In this part, we introduce the Spark application configuration parameter optimizer in detail, which mainly consists of three phases: the collection and preprocessing of historical Spark application data; the training of a Spark application performance prediction model; and the search for better configuration parameters, corresponding to Figure 1a–c.

#### 3.1. Data Collection and Preprocessing

##### 3.1.1. Parameter Selection

In Spark configuration parameter optimization, it is very important to choose an appropriate parameter space. A large number of parameter features increase the searching time, while few parameter features may lead to sub-optimal configuration. In the Spark parameter list, some parameters (such as `spark.app.name`) have no influence on the performance of the application, which are removed directly. Lastly, on the basis of theories [24,25] and experience [26], we selected 16 parameters that had the greatest impact on the performance of the Spark system. These parameters mainly covered the allocation and use of available cluster resources (CPU, memory, and disk), data transmission and compression, scheduling, etc. Table 1 lists the name, function, default value, and value range of each configuration parameter on the basis of our Spark cluster.



**Figure 1.** The framework of the Spark configuration parameter optimizer.

**Table 1.** Default Spark parameter values with range.

Spark Parameters	Function	Default	Range/Step
spark.executor.cores	Number of cores to use on each executor	1	1–8, 1
spark.executor.memory	Amount of memory to use per executor process	1	1–8, 1
spark.executor.instances	Number of executors	2	2–8, 1
spark.driver.cores	Number of cores to use for the driver process	1	1–4, 1
spark.driver.memory	Amount of memory to use for the driver process	1 g	1–4, 1
spark.reducer.maxSizeInFlight	Maximal size of map outputs to fetch simultaneously from each reduce task	48 m	48–96, 8
spark.shuffle.compress	Whether to compress map output files	true	true, false
spark.shuffle.spill.compress	Whether to compress data spilled during shuffles	true	true, false
spark.shuffle.file.buffer	Size of the inmemory buffer for each shuffle file output stream	32 k	32–128, 16
spark.broadcast.blockSize	Size of each piece of a block for TorrentBroadcastFactory	4 m	4–24, 2
spark.broadcast.compress	Whether to compress broadcast variables before sending them	true	true, false
spark.memory.fraction	Fraction of (heap space—300 MB) used for execution and storage	0.6	0.3–0.8, 0.1
spark.memory.storageFraction	Amount of storage memory	0.5	0.3–0.8, 0.1
spark.rpc.message.maxSize	Maximal message size to allow in “control plane” communication	128 m	128–256, 32
spark.rdd.compress	Whether to compress serialized RDD partitions	false	true, false
spark.io.compression.codec	The codec used to compress internal data such as RDD partitions, event logs, broadcast variables, and shuffle outputs	lz4	Lz4, snappy

### 3.1.2. Data Preprocessing

In the data collection and preprocessing phase, we applied Hibench [27] to generate the experimental data to be processed by Spark applications, and then randomly generated parameter sets  $P$  from configuration parameter space  $D$ . The representation is as follows:

$$P = \{p_i \in D \mid 1 \leq i \leq M\} \quad (1)$$

$$C = \{c_{ij} \in p_i \mid 1 \leq i \leq M, 1 \leq j \leq N\} \quad (2)$$

where  $M$  represents the number of parameter combinations,  $N$  denotes the number of parameter types in each parameter combination,  $p_i$  represents the  $i$ -th parameter set obtained by sampling, and  $c_{ij}$  represents the  $j$ -th parameter value in the  $i$ -th parameter set.

Lastly, we executed the Spark application with each parameter set on Yarn mode on our Spark cluster, and recorded the running time of the Spark application with different parameter sets to generate the basic history dataset for training in Equation (3).

$$T = \{t_1, t_2, t_3, \dots, t_i \mid 1 \leq i \leq M\} \quad (3)$$

where  $t_i$  represents the Spark execution time under the  $i$ -th parameter set. Since the structure and format of each Spark configuration parameter are inconsistent, the value of each configuration parameter is normalized before model training by Equation (4):

$$c_{ij} = \frac{(c_{ij} - \mu_j)}{\sigma_j} \quad (4)$$

where  $c_{ij}$  is the original configuration parameter value,  $\mu_j$  is the mean value of this type of configuration parameter, and  $\sigma_j$  is the standard deviation of this type of configuration parameter. Standardization scales the value of each configuration parameter to the same numerical range, ensuring that the configuration parameters are at the same level, thereby improving the comparability of different configuration parameters.

### 3.2. Performance Prediction Model

After standardizing the base dataset for the Spark application, we generated a dataset for model training. To improve the efficiency of searching for optimal parameters, we used a performance model to evaluate the actual performance of the Spark application. In general, the performance of a Spark application [28] can be expressed as in Equation (5):

$$Perf = Func(app, input, rsrc, config) \quad (5)$$

where  $Perf$  represents the running performance of the application,  $Func$  represents the performance evaluation function,  $app$  is the Spark application,  $input$  represents the input data,  $rsrc$  denotes the cluster resource, and  $config$  is the Spark configuration parameter set.

In this paper, we optimized the execution time of Spark applications by adjusting the Spark configuration parameters, which means that  $app$ ,  $input$ , and  $rsrc$  are constants in Equation (5),  $config$  is the controlled variable, and  $Perf$  is the dependent variable based on the evaluation function. Therefore, this problem can be transformed into predicting the execution time of a Spark application given a set of configuration parameters.

We employed a deep neural network to build a Spark performance prediction model. The structure is shown in Figure 2. A deep neural network is mainly divided into three types of layers: the input layer, the hidden layer, and the output layer. These layers are fully connected, which indicates that any neuron in the  $n$ -th layer is connected with any neuron in the  $n+1$ -th layer. The connection relationship is expressed in Formula (6).

$$y_m = f\left(\sum_{i=1}^n w_{im}x_i + b_m\right) \quad (6)$$

where  $w$  is the weight,  $b$  is the bias,  $x_i$  is the neural unit, and  $f(\cdot)$  is the activation function. In our deep neural network model structure, the input layer was the input of 16 Spark configuration parameters, the hidden layer was 3 layers (the number of neurons in each layer was 12, 8, and 4), and the output layer is the execution of the Spark application time. The activation function was rectified linear unit (ReLU). The definition of ReLU is  $f(x) = \max(0, x)$ , which is a popular activation function in neural networks. It is nonlinear and can be run backwards to minimize errors. During training, we applied an efficient optimizer, Adam, which iteratively updated the weights of the network on the basis of training data. At the same time, in order to render the model more accurate and with a stronger generalization ability, we also introduced Bayesian optimization to adjust the hyperparameters of our evaluation model, such as batch sizes, learning rates, and epochs.



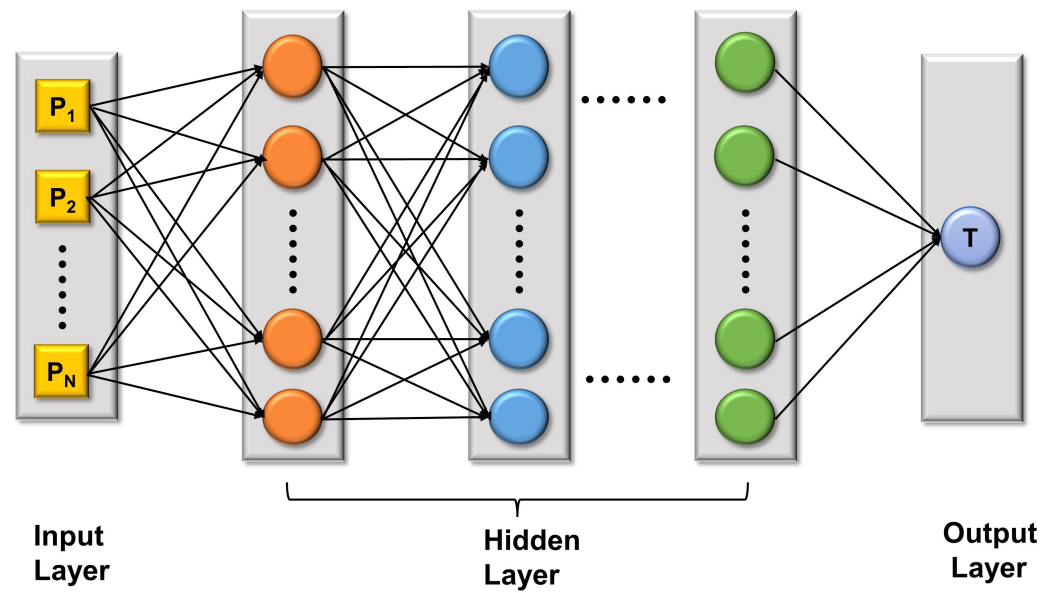


Figure 2. Deep neural network model.

### 3.3. Parameter Searching

The optimization problem of Spark configuration parameters in a discrete, finite search space can be modeled as a Markov decision process (MDP). An MDP consists of a quadruple  $\langle S, A, P_{sa}, R \rangle$ , where  $S$  is the finite state set,  $A$  is the finite motion set,  $P_{sa}$  is the state transition probability, and  $R$  is the reward function [29]. After converting the Spark configuration parameter optimization problem into MDP, the configuration parameter set at the current moment is noted as the identifier of state  $s$ . Under this configuration parameter, the Spark application prediction time is used as the value of the state  $s$ , and each execution of an action is converted into another state.

Q-learning is a value-based reinforcement-learning algorithm that can solve the above MDP problem. The decision-making agent seeks the optimal strategy by maximizing the state action value function  $Q(s, a)$ , uses  $S$  and  $A$  to construct a  $Q$  value table to store the value of  $Q(s, a)$ , and selects the action that can obtain the maximal profit in the current state according to the  $Q$  value table in each step by adopting the  $\epsilon$ -greedy strategy. The  $Q$  value update formula is shown in Equation (7).

$$Q_{i+1}(s, a) = Q_i(s, a) + \alpha \left[ r + \gamma \max_{a'} Q_i(s', a') - Q_i(s, a) \right] \quad (7)$$

where  $\alpha$  is the learning rate, and  $\gamma$  is the conversion factor. When  $\gamma = 0$ , only the immediate reward is considered. When  $\gamma = 1$ , the long-term reward is as important as the immediate reward.  $s'$  is the new state obtained after state transition,  $a'$  is the action selected according to the  $\epsilon$ -greedy policy, and  $r$  is the immediate reward by taking action  $a$  from state  $s$  to state  $s'$ .

In order to prevent the reinforcement-learning agent from conducting invalid searches among the parameter sets with poor performance, we improved the Q-learning algorithm, and did not set fixed starting and ending states in each epoch of the agent's learning. The processing of the improved Q-learning algorithm is shown in Algorithm 1. *initQtable*( $S, Q$ ) means initializing the  $Q$ -Table and randomly selecting a set of configuration parameters from the parameter search space as the identifier of  $s_0$ , the predicted execution time of the Spark application with this configuration is used as the value of the state  $s_0$ , and all the corresponding values of  $Q(s_0, a)$  are initialized to 0. *bestState* records the best state at the current moment (the initial state is  $s_0$ ). The outermost "For" loop represents the number of learning epochs, the "While" loop represents each epoch of learning, which starts from the best state at the current moment. *chooseAction*( $s$ ) uses the  $\epsilon$ -greedy strategy

to choose an action according to the Q-table. Each action means an update of configuration parameters and a transition to a new state  $s'$  after executing the action. If  $s'$  was not in the Q-table, we inserted it into Q-table and initialized all corresponding  $Q(s', a)$  values to 0.  $updateQtable()$  was used to update the Q-table according to Formula (7). If state  $s'$  is better than the best state at the current moment,  $bestState$  is updated. The ratio represents the ratio of performance improvement. If the ratio is less than 0.1, the current modified direction is not ideal, and current  $s'$  is the termination state to end this epoch.

---

**Algorithm 1** The process of the improved Q-learning algorithm.

---

**Input:**

State List  $S = [s_0, s_1, s_2, \dots, s_n]$ ;  
 Qtable List  $Q = [q_0, q_1, q_2, \dots, q_n]$ ;

**Output:**

bestConf;

```

1: initQtable(S, Q)
2: bestState =  $s_0$ 
3: for round from 1 to rounds do
4:    $s = bestState$  //Store the best state
5:   while ratio > 0.1 do
6:      $a = chooseAction(s)$  //use the  $\epsilon$ -greedy strategy to select the action
7:      $s' = getOrCreateState(s, a)$  //if the state exists, return, does not exist, create and
      return
8:      $t, t' = getTime(s, s')$  //obtain the predicted execution time of the Spark applica-
      tion corresponding to the status
9:      $r = t - t'$  //use the time difference as a timely reward
10:    updateQtable() // update Qtable according to Formula (7)
11:    if  $t' < getTime(bestState)$  then
12:      bestState =  $s'$  //store the best state
13:      ratio =  $(t - t')/t$  //compute the ratio of performance improvement
14:       $s = s'$ 
15: return bestState

```

---

## 4. Experiments

In this section, we introduce the experimental environment (including Spark cluster configuration, Spark applications, and experimental data statistical methods), evaluation metrics for Spark performance prediction models, and the analysis of experimental results.

### 4.1. Experimental Setup

In order to evaluate the effectiveness of our proposed optimizer, we implemented the experiments on a cluster of six computing machines, each with Intel(R) Core(TM) i9-10900K CPU @ 3.70 GHz processor, 20 cores, 64 GB main memory, and 1 GigE Ethernet network. Our Spark cluster was based on CentOS Linux release 8.2.2004, JDK version 1.8, Apache Hadoop version 2.7.7, and Apache Spark 3.1.3.

To test the effectiveness of our method on different Spark application types, we chose WordCount, PageRank, KMeans, and TeraSort, provided by Hibenach [27], as our benchmark applications. These four applications represent different workloads of Apache Spark. WordCount is CPU- and I/O-intensive, TeraSort is memory-intensive, and PageRank and KMeans are iteration-intensive.

After the collecting training data, we measured the execution time of each Spark application under a given configuration 9 times, and used the median as the real execution time of each application to eliminate bias caused by other factors such as the computer hardware and network.

#### 4.2. Performance Metrics

In order to evaluate the quality of the Spark performance prediction model, we used the following evaluation metrics:

**Mean absolute error (MAE):** The average value of the absolute error can reflect the actual situation of the predicted value error. The smaller the MAE value is, the higher the model accuracy is. Assuming that the real values in the test set were  $y_1, y_2, y_3, \dots, y_n$  and the corresponding Spark performance model predicted values were  $f_1, f_2, f_3, \dots, f_n$ , MAE is represented in Equation (8).

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - f_i| \quad (8)$$

**Root mean square error (RMSE):** The standard deviation of the residuals (prediction errors). The smaller the RMSE value is, the higher the accuracy of the Spark performance prediction model is. The representation of RMSE is in Equation (9).

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - f_i)^2} \quad (9)$$

**R-squared ( $R^2$ ):** The coefficient of determination can measure the fitness of one model. The closer the value of  $R^2$  is to 1, the better the fit of the Spark performance prediction model is. The mean of the true values is  $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$ . The representation of  $R^2$  is:

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - f_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \quad (10)$$

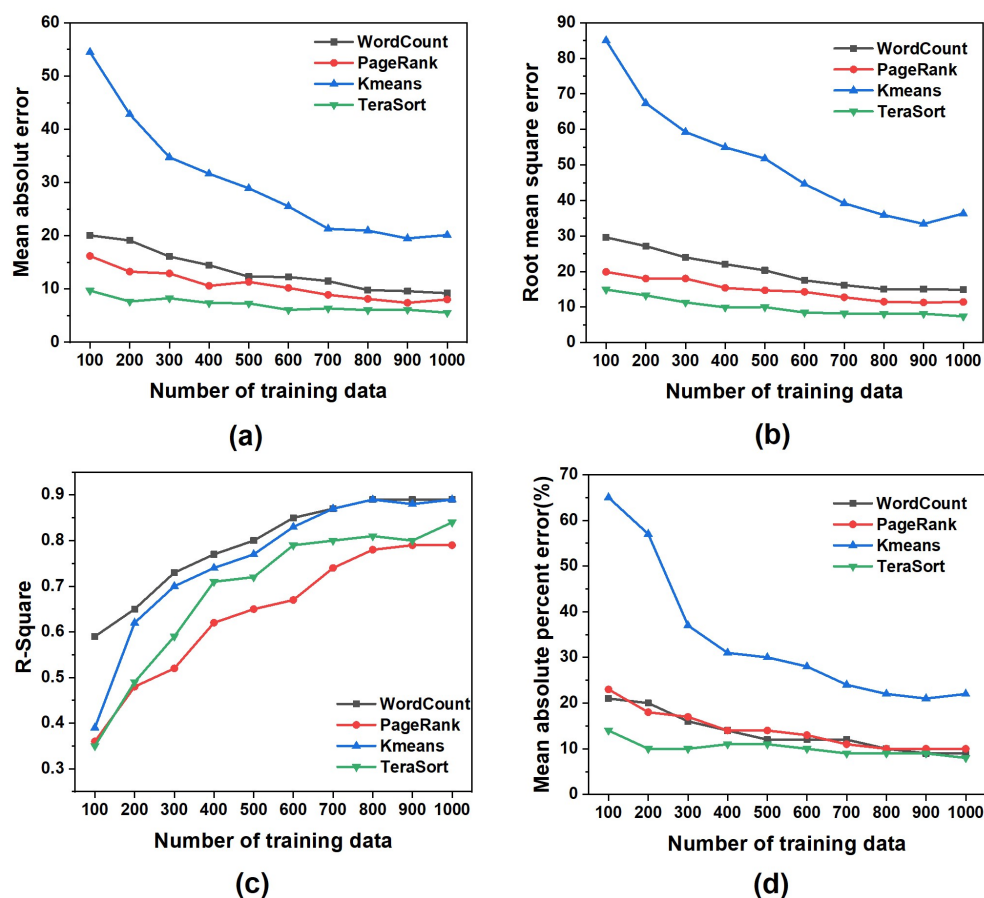
**Mean absolute percentage error (MAPE):** The deformation (percentage value) of the MAE. The smaller the value of MAPE is, the better the accuracy of the Spark performance prediction model is. The representation of MAPE is shown in Equation (11):

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{y_i - f_i}{y_i} \right| \quad (11)$$

#### 4.3. Experimental Results

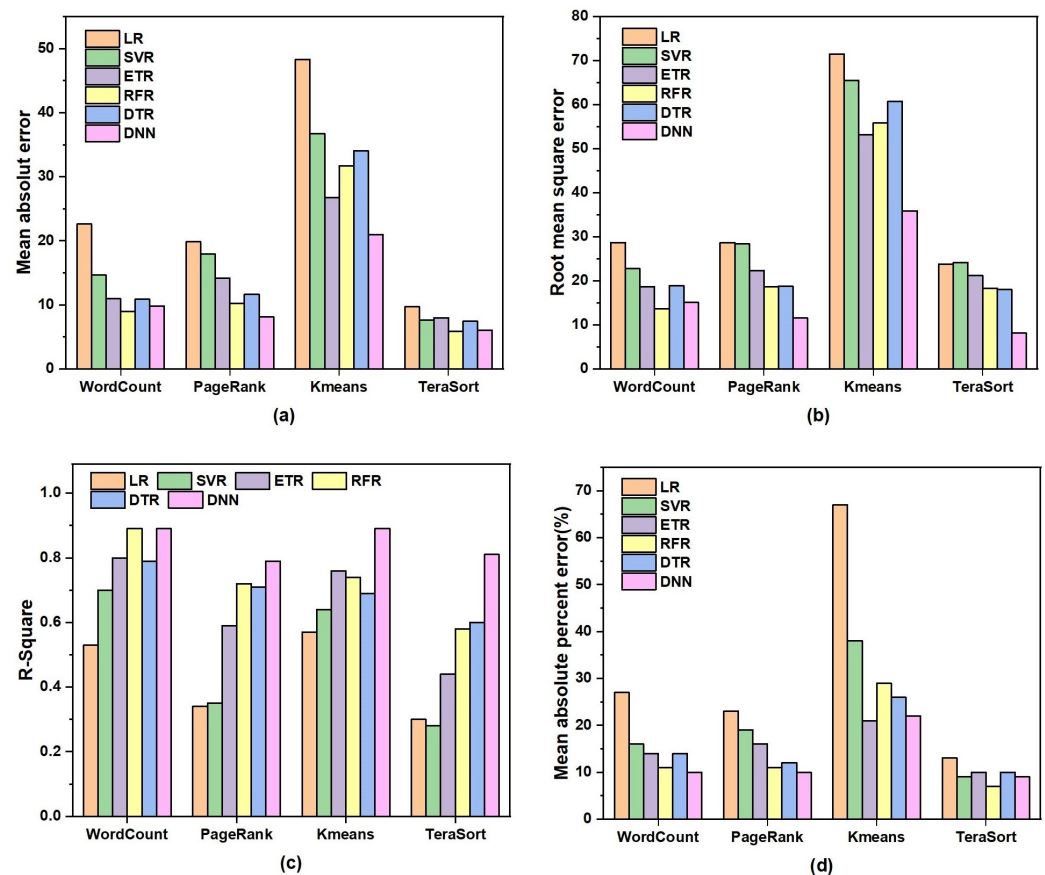
**Relationship between the performance prediction model and the number of training data:** As with most machine-learning tasks, using more training data leads to better predictive models. However, in practice, since each training datum is collected by the Spark application execution in a real cluster environment, collecting too many training data increases the time cost. Therefore, we need to find a balance between various model evaluation indicators and the number of training data, that is, reduce the number of training data as much as possible under certain conditions. Figure 3 shows the changes in the evaluation indicators of different Spark application performance prediction models from 100 to 1000 training data. Figure 3a shows that, with the continuous increase in the number of training data, the MAE value of the prediction model of each Spark application gradually decreased, of which Kmeans varied most significantly. When the number of training samples reached 700, the MAE value was no longer significantly decreased. Figure 3b,d shows that the changing trends of RMSE and MAPE were similar to those in Figure 3a. Figure 3c shows that, with the continuous increase in the number of training data, the value of  $R^2$  also rapidly increased, that is, the fitting degree of our model was improved. After the number of training data had reached 800, the value of  $R^2$  tended to be stable. The above experimental analysis shows that it is better to build a performance prediction model for a Spark application when the number of training data is greater than 800.





**Figure 3.** The number of training data is from 100 to 1000, and the evaluation indicators of different Spark application performance prediction models changed: (a) Mean absolute error. (b) Root mean square error. (c) R-squared. (d) Mean absolute percentage error.

**Performance comparison of deep neural network models and other regression models:** We compared Spark performance prediction models built with linear regression (LR), support vector machine regression (SVR), external tree regression (ETR), random forest regression (RFR), decision trees (DTR), and deep neural networks (DNNs). To fairly compare these methods, we used the same training environment, training dataset, and testing set as those of our proposed DNN method. Figure 4 shows the comparison of various evaluation metrics between different performance prediction models on four Spark applications. The DNN model showed significant improvement over the other models in all evaluation metrics for all Spark applications. For the Kmeans application in Figure 4b, the RMSE of the DNN model was reduced by 49.7%, 45.1%, 32.4%, 35.6%, and 40.8% compared to the LR, SVR, ETR, RFR, and DTR regression models, respectively. This is because Spark not only has a huge configuration parameter space, but the inter-relationship between configuration parameters is also very complex. Spark configuration parameter performance prediction models are definitely not linear, so LR performed the worst. For such high-dimensional and feature-related problems, shallow machine-learning methods such as SVR, ETR, and DTR are also not very effective. Deep learning can gradually learn through multiple networks, extract complex and effective features, and has higher prediction accuracy and generalization ability.



**Figure 4.** Performance comparison of deep neural-network models with other regression models on different Spark applications: (a) Mean absolute error. (b) Root mean square error. (c) R-squared. (d) Mean absolute percentage error.

**Performance comparison between the improved Q-learning algorithm and other parameter search algorithms:** Table 2 shows the optimal values and search times for random search, simulated annealing, and the variant of the Q-learning algorithm on four Spark applications. In terms of optimal values, the random search algorithm performed the worst, with simulated annealing and improved Q-learning yielding very similar values. In terms of search time, the simulated annealing algorithm had the longest parameter search time, and the improved Q-learning algorithm had the shortest parameter. In the TeraSort application, the parameter search time of the improved Q-learning algorithm was reduced by 60.4% compared to the simulated annealing algorithm. The improved Q-learning algorithm could greatly reduce the time cost of parameter searching while obtaining better results, and achieved better time performance. This is because the agent of the improved Q-learning algorithm started from the best state at the current moment in each epoch of learning. If the next state was not better, the learning of this epoch was ended immediately, so as to prevent the agent from performing invalid searches among parameter sets with poor performance.

**Time performance overhead evaluation:** Our scheme consists of a performance prediction model and an improved Q-learning search algorithm. We applied the scheme proposed in this paper to four different types of Spark applications. Each type of application was conducted 9 times to search for the better configuration parameters and calculate the exact time cost, which is shown in Table 3. The time overhead of Spark configuration parameter optimization is about 61 s, which is within an acceptable range.

**Table 2.** Performance comparison of the improved Q-learning algorithm and other search algorithms on different Spark applications.

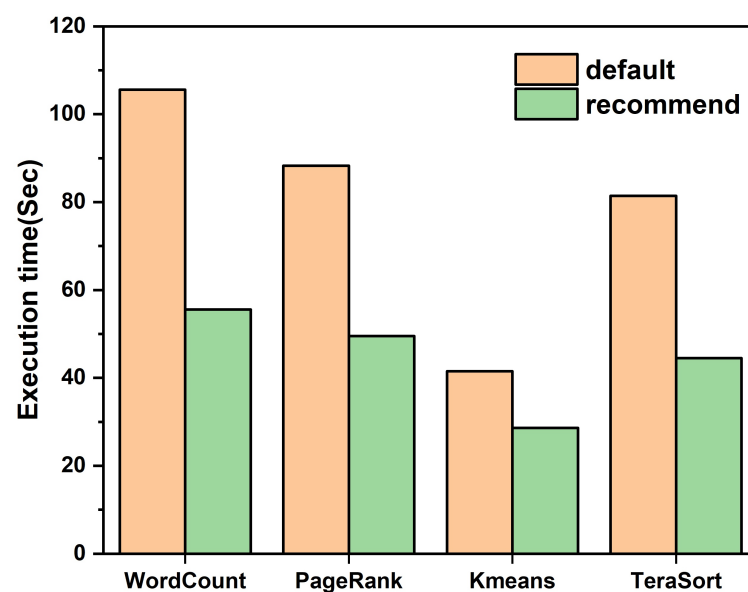
Algorithms	WordCount		PageRank		Kmeans		TeraSort	
	Value (s) <sup>1</sup>	Time (ms) <sup>2</sup>	Value (s) <sup>1</sup>	Time (ms) <sup>2</sup>	Value (s) <sup>1</sup>	Time (ms) <sup>2</sup>	Value (s) <sup>1</sup>	Time (ms) <sup>2</sup>
Rand	51.9	653.0	50.5	463.8	26.2	589.4	42.6	818.6
Simulated Annealing	50.9	830.6	49.4	835.8	25.8	854.0	41.8	905.4
Q-learning	50.2	463.8	49.6	435.0	26.1	296.4	42.5	358.0

1. The optimal value searched by the algorithm; 2. time indicates the time spent by the algorithm to search for the optimal value.

**Table 3.** The time overhead of better configuration parameter solving for different types of Spark applications.

Spark Application	Time (s)
WordCount	61.571
PageRank	61.402
Kmeans	61.543
TeraSort	61.345

**Performance comparison between recommended configuration parameters and default configuration parameters:** By using our method, the recommended configuration parameters for four different types of Spark applications were obtained, as shown in Table 4. In order to avoid the contingency of the experiment, we submitted four types of Spark applications to the Spark cluster 9 times with the default and recommended configuration parameters, and obtained the median of execution time of Spark applications under the two configurations. The experimental results are shown in Figure 5. Compared with the default configuration, the recommended configuration for WordCount, PageRank, Kmeans, and TeraSort achieved performance improvement of 47%, 43%, 31%, and 45%, respectively, which shows that our Spark configuration parameter optimizer is effective in improving the performance of different types of Spark applications.

**Figure 5.** Execution time comparison between recommended configuration parameters and default configuration parameters on different Spark applications.

**Table 4.** Default parameter values and recommended parameter values for different Spark applications.

Parameter	Default	Recommend			
		WordCount	PageRank	Kmeans	TeraSort
spark.executor.cores	1	8	6	7	5
spark.executor.memory	1 g	5 g	4 g	4 g	7 g
spark.executor.instances	2	4	5	5	6
spark.driver.cores	1	2	3	3	3
spark.driver.memory	1 g	3 g	1 g	2 g	3 g
spark.reducer.maxSizeInFlight	48 m	48 m	72 m	56 m	36 m
spark.shuffle.compress	true	true	true	false	true
spark.shuffle.spill.compress	true	true	false	true	false
spark.shuffle.file.buffer	32 k	48 k	96 k	48 k	112 k
spark.broadcast.blockSize	4 m	4 m	10 m	6 m	6 m
spark.broadcast.compress	true	true	false	true	false
spark.memory.fraction	0.6	0.4	0.7	0.5	0.4
spark.memory.storageFraction	0.5	0.6	0.5	0.5	0.4
spark.rpc.message.maxSize	128 m	256 m	128 m	192 m	160 m
spark.rdd.compress	false	true	false	false	true
spark.io.compression.code	lz4	lz4	snappy	lz4	snappy

## 5. Conclusions

For Spark's configuration parameter optimization problem, we employed a deep neural network to build a Spark performance prediction model, and designed an improved Q-learning algorithm as our Spark configuration parameter optimizer to search for better configuration parameters. The experimental results show that the Spark performance prediction model based on deep neural network is more accurate, and the improved Q-learning algorithm could greatly reduce the time cost of parameter searching while obtaining better results compared with the default configuration parameters for four different types of Spark applications.

Not only can the configuration parameters of the Spark application affect the performance, but the state of the Spark cluster (such as memory utilization, CPU utilization, and network bandwidth) can also influence the performance of the application. When CPU utilization is too high, it is better to allocate a small number of CPU cores to the application to reduce waiting time.

Our current work can only solve the optimizer problem for a single Spark application on a given Spark cluster. In future work, we aim to simultaneously optimize multiple Spark applications with the resource constraints of a Spark cluster.

**Author Contributions:** Conceptualization, H.Z. and X.H.; methodology, X.H.; software, X.H.; validation, H.Z., X.Z. and X.H.; formal analysis, H.Z.; writing—original draft preparation, X.H.; and writing—review and editing, H.Z., X.Z. and X.H.; visualization, H.Z., X.Z. and X.H.; supervision, H.Z.; project administration, H.Z.; funding acquisition, H.Z. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported in part by the Science and Technology Research Project of Hebei Higher Education Institutions (No. QN2020133) and in part by the Natural Science Foundation of Hebei Province of China (Nos. F2019201361, F2020201018).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Dataset link: <https://github.com/Intel-bigdata/HiBench> (accessed on 26 June 2022).

**Acknowledgments:** The authors would like to thank all of the cited authors, and the anonymous reviewers in this article for their helpful suggestions.

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

## References

1. 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. [\[CrossRef\]](#)
2. 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 Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA, USA, 25–27 April 2012; pp. 15–28.
3. Cheng, G.; Ying, S.; Wang, B.; Li, Y. Efficient performance prediction for apache spark. *J. Parallel Distrib. Comput.* **2021**, *149*, 40–51. [\[CrossRef\]](#)
4. Esmaeilzadeh, A.; Heidari, M.; Abdolazimi, R.; Hajibabae, P.; Malekzadeh, M. Efficient large scale nlp feature engineering with apache spark. In Proceedings of the 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 26–29 January 2022; pp. 0274–0280.
5. Haggag, M.; Tantawy, M.M.; El-Soudani, M.M. Implementing a deep learning model for intrusion detection on apache spark platform. *IEEE Access* **2020**, *8*, 163660–163672. [\[CrossRef\]](#)
6. Roghani, H.; Bouyer, A.; Nourani, E. PLDLS: A novel parallel label diffusion and label Selection-based community detection algorithm based on Spark in social networks. *Expert Syst. Appl.* **2021**, *183*, 115377. [\[CrossRef\]](#)
7. Yu, Z.; Bei, Z.; Qian, X. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA, 24–28 March 2018; pp. 564–577.
8. Gao, Z.; Wang, T.; Wang, Q.; Yang, Y. Execution Time Prediction for Apache Spark. In Proceedings of the 2018 International Conference on Computing and Big Data, Charleston, SC, USA, 8–10 September 2018; pp. 47–51.
9. Rahman, M.A.; Hossen, J.; Sultana, A.; Al Mamun, A.; Aziz, N.A.A. A smart method for spark using neural network for big data. *Int. J. Electr. Comput. Eng.* **2021**, *11*, 2525. [\[CrossRef\]](#)
10. Feng, Q.; Chen, L.; Chen, C.P.; Guo, L. Deep fuzzy clustering—A representation learning approach. *IEEE Trans. Fuzzy Syst.* **2020**, *28*, 1420–1433. [\[CrossRef\]](#)
11. Baker, B.; Gupta, O.; Naik, N.; Raskar, R. Designing neural network architectures using reinforcement learning. *arXiv* **2016**, arXiv:1611.02167.
12. Zhong, Z.; Yang, Z.; Deng, B.; Yan, J.; Wu, W.; Shao, J.; Liu, C.L. Blockqnn: Efficient block-wise neural network architecture generation. *IEEE Trans. Pattern Anal. Mach. Intell.* **2020**, *43*, 2314–2328. [\[CrossRef\]](#) [\[PubMed\]](#)
13. Herodotou, H.; Chen, Y.; Lu, J. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–37. [\[CrossRef\]](#)
14. Gounaris, A.; Torres, J. A methodology for spark parameter tuning. *Big Data Res.* **2018**, *11*, 22–32. [\[CrossRef\]](#)
15. Singhal, R.; Singh, P. Performance assurance model for applications on SPARK platform. In Proceedings of the Technology Conference on Performance Evaluation and Benchmarking, Munich, Germany, 28 August 2017; pp. 131–146.
16. Huang, S.; Xu, J.; Liu, R.; Liao, H. A novel compression algorithm decision method for spark shuffle process. In Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 11–14 December 2017; pp. 2931–2940.
17. Wang, G.; Xu, J.; He, B. A novel method for tuning configuration parameters of spark based on machine learning. In Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, Australia, 12–14 December 2016; pp. 586–593.
18. Islam, M.T.; Karunasekera, S.; Buyya, R. dSpark: Deadline-based resource allocation for big data applications in Apache Spark. In Proceedings of the 2017 IEEE 13th International Conference on E-Science (e-Science), Auckland, New Zealand, 24–27 October 2017; pp. 89–98.
19. Chao, Z.; Shi, S.; Gao, H.; Luo, J.; Wang, H. A gray-box performance model for apache spark. *Future Gener. Comput. Syst.* **2018**, *89*, 58–67. [\[CrossRef\]](#)
20. Shah, S.; Amannejad, Y.; Krishnamurthy, D.; Wang, M. PERIDOT: Modeling Execution Time of Spark Applications. *IEEE Open J. Comput. Soc.* **2021**, *2*, 346–359. [\[CrossRef\]](#)
21. Gu, J.; Li, Y.; Tang, H.; Wu, Z. Auto-tuning spark configurations based on neural network. In Proceedings of the 2018 IEEE International Conference on Communications (ICC), Kansas City, MO, USA, 20–24 May 2018; pp. 1–6.
22. Tooley, R. Auto-Tuning Spark with Bayesian Optimisation. Available online: [https://www.cl.cam.ac.uk/~ey204/pubs/MPHIL\\_P3/2021\\_Ross.pdf](https://www.cl.cam.ac.uk/~ey204/pubs/MPHIL_P3/2021_Ross.pdf) (accessed on 26 June 2022).
23. Patanshetti, T.; Pawar, A.A.; Patel, D.; Thakare, S. Auto Tuning of Hadoop and Spark parameters. *arXiv* **2021**, arXiv:2111.02604.

24. Karau, H.; Warren, R. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2017.
25. Apache. Spark Configuration. 2022. Available online: <https://spark.apache.org/docs/latest/configuration.html> (accessed on 26 June 2022).
26. Petridis, P.; Gounaris, A.; Torres, J. Spark parameter tuning via trial-and-error. In Proceedings of the INNS Conference on Big Data, Thessaloniki, Greece, 23–25 October 2016; pp. 226–237.
27. Huang, S.; Huang, J.; Dai, J.; Xie, T.; Huang, B. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In Proceedings of the 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), Long Beach, CA, USA, 1–6 March 2010; pp. 41–51.
28. Herodotou, H.; Lim, H.; Luo, G.; Borisov, N.; Dong, L.; Cetin, F.B.; Babu, S. Starfish: A Self-tuning System for Big Data Analytics. *Cidr* **2011**, *11*, 261–272.
29. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.