Hindawi
Complexity
Volume 2020, Article ID 4793545, 13 pages
https://doi.org/10.1155/2020/4793545

*Research Article*

# Sorting Data via a Look-Up-Table Neural Network and Self-Regulating Index

**Ying Zhao**,[1,2] **Dongli Hu**,[1] **Dongxia Huang**,[1] **You Liu**,[1] **Zitong Yang**,[3] **Lei Mao**,[1] **Chao Liu**,[4] **and Fangfang Zhou**[1]

[1]*School of Computer Science and Engineering, Central South University, Changsha 410083, China*
[2]*Rail Data Research and Application Key Laboratory of Hunan Province, Changsha 410083, China*
[3]*School of Automation, Central South University, Changsha 410083, China*
[4]*Institute of Systems Engineering, Academy of Military Sciences, People's Liberation Army, Beijing 100000, China*

Correspondence should be addressed to Chao Liu; generaladolph@163.com and Fangfang Zhou; zff@csu.edu.cn

Received 16 February 2020; Revised 9 June 2020; Accepted 16 June 2020; Published 27 July 2020

Academic Editor: Quanmin Zhu

The so-called learned sorting, which was first proposed by Google, achieves data sorting by predicting the placement positions of unsorted data elements in a sorted sequence based on machine learning models. Learned sorting pioneers a new generation of sorting algorithms and shows a great potential because of a theoretical time complexity $O(N)$ and easy access to hardware-driven accelerating approaches. However, learned sorting has two problems: controlling the monotonicity and boundedness of the predicted placement positions and dealing with placement conflicts of repetitive elements. In this paper, a new learned sorting algorithm named LS is proposed. We integrate a back propagation neural network with the technique of look-up-table in LS to guarantee the monotonicity and boundedness of the predicted placement positions. We design a data structure called the self-regulating index in LS to tentatively store and duly update placement positions for eliminating potential placement conflicts. Results of three controlled experiments demonstrate that LS can effectively control the monotonicity and boundedness, achieve a better time consumption than quick sort and Google's learned sorting, and present an excellent stability when the data size or the number of repetitive elements increases.

## 1. Introduction

Sorting is a fundamental data operation that obtains a sorted data sequence from an unsorted dataset increasingly or decreasingly. It is a crucial part of various advanced algorithms and systems [1, 2]. Classical sorting algorithms, such as bubble sort, quick sort, and heap sort, accomplish sorting by comparing data elements with certain strategies; they can satisfy general sorting requirements in scientific studies and engineering projects [3, 4]. However, the efficiency of comparison sorting algorithms would be greatly challenged when faced with large-scale data. At present, two categories of approaches are proposed to meet this challenge. The first category is hardware-driven accelerating approaches that reduce sorting time by utilizing multicore CPU/GPUs or computer clusters to conduct parallel sorting [5–8]. Such approaches require massive computing resources and do not reduce the time complexity of sorting. Approaches of the second category, which is called non-comparison sorting, distribute input data in intermediate data structures to gather and place them in a sorted sequence [9–12]. The time complexity of non-comparison sorting, such as self-indexed sort and Qureshi sort, is slightly superior to that of quick sort, while their memory space consumptions are fairly large and suitable data types are limited [11, 12].

Most recently, Google proposed a new sorting algorithm (Google-sort) that sorts data by two steps. First, it predicts the placement position of each element of an unsorted dataset in a sorted sequence based on a machine learning model [13, 14]. In accordance with the predicted positions, unsorted elements are placed one by one into a sorting result array to obtain the final sorted sequence. This algorithm pioneers a new generation of sorting algorithms, that is, the so-called learned sorting. Learned sorting has three advantages: (1) the time complexity can reach $O(N)$ theoretically. (2) Machine learning models readily adapt to various data types. (3) Machine learning models can be easily combined with hardware-driven accelerating approaches.

However, we find that learned sorting has two problems: (1) constructing the cumulative distribution function (CDF) of an unsorted dataset by using a machine learning model for placement position prediction is crucial in the first step of learned sorting. However, most machine learning models cannot fully guarantee the monotonicity and boundedness of CDF. For instance, a fully-connected neural network guarantees neither the monotonicity nor boundedness (Section 5.1); recursive-model indexes used by Google-sort dissatisfy the monotonicity [13, 14]. Therefore, a small part of unsorted elements would be predicted to incorrect or non-existent placement positions. Additional data operations are needed for correction. (2) Some elements in an unsorted dataset inevitably have the same key value. We call such elements as repetitive elements. For example, in a dataset with 7 elements and key values [3, 4, 8, 7, 4, 8, 4], the three elements with a value of 4 and the two elements with a value of 8 are repetitive. Repetitive elements would be predicted to the same position, which would result in placement conflicts. That is, only one of the repetitive elements with the same value can be correctly placed in the sorting result array. Additional data operations are needed again. Google-sort locally adopts quick sort to deal with placement conflicts. This manner reduces the overall performance of Google-sort. Self-indexed sort and Qureshi sort use linear mapping and difference mapping, respectively, to eliminate conflicts. However, both of them present high space consumptions [11, 12].

This paper proposes a new learned sorting algorithm that sorts data via a look-up-table (LUT) neural network (LNN) and self-regulating index (SRI) data structure, herein referred to as LS. This algorithm can effectively solve the two aforementioned problems. For the first problem, we introduce the technique of LUT into back propagation (BP) neural network for placement position prediction. LUT allows using predefined monotonicity rules and output ranges to control the mapping of inputs to outputs. We propose to replace the weight matrix of BP neural network with LUT to generate an LNN for guaranteeing the monotonicity and boundedness of CDF. For the second problem, we design a new data structure called the self-regulating index (SRI). SRI uses an auxiliary array of the same size to tentatively store the predicted placement positions of unsorted elements. SRI provides a self-regulating operation that automatically updates the tentatively stored positions during placing unsorted elements into the sorting result array to eliminate placement conflicts. Moreover, SRI features low space consumption and a simple and efficient data operation.

Three controlled experiments are conducted to verify the effectiveness of LS. The first experiment demonstrates that LNN can fully guarantee the monotonicity and boundedness of CDF, but three referenced machine learning models fail. The second experiment shows that LS has an average margin of 27% faster execution than quick sort on the experimental datasets with different data sizes. The third experiment indicates that the performance of LS algorithm is more stable than that of Google-sort when the number of repetitive elements increases.

In summary, we propose a new learned sorting algorithm inspired by Google-sort. This algorithm introduces LUT and SRI to solve the two problems that may be encountered in learned sorting. In theory, the algorithm can reach the $O(N)$ time complexity. The algorithm is also stable regardless of the data size and the proportion of repetitive elements in unsorted datasets.

## 2. Related Work

Sorting is a basic and ubiquitous operation in computer science and has a long history. Many classical sorting algorithms, such as quick sort, merge sort, and heap sort, have been widely used [15]. Meanwhile, technical improvements are continuously evolving in these years toward the unremitting goal of being faster than before. The existing improvements can generally be categorized into two technical routes, namely, the hardware-driven accelerating approach and complexity reduction.

Hardware-driven accelerating approaches speed up sorting by utilizing high-performance hardware. Marszalek [5] proposed a method that divides the input data into smaller parts to ensure that each processor of a multiprocessor computer can perform sorting operations on the allocated memory. Empirically, the speed of the entire process is high when a large number of processors are involved. Pang et al. [6] presented a large-scale distributed sorting algorithm based on cloud computing; this algorithm applies a control terminal to manage a set of location-distributed working machines to complete a parallel sorting. The experimental results showed that the data transfer optimized by the control terminal can improve the overall sorting efficiency. Gebali et al. [7] presented a new structured algorithm for parallel sorting. The algorithm achieves parallelization by using processors to sort each dimension of multidimensional datasets, and there are no restrictions on the number of processors in each dimension. Faujdar and Ghrera [8] accelerated the bubble sort using GPU. All the aforementioned methods can significantly speed up the processing of data sorting. However, these hardware-driven accelerating methods commonly require a large amount of computing resources and do not change the time complexity of sorting; thus, their application ranges are limited [16].

Reducing the complexity of sorting algorithms features a common meaning. However, Thomas [17] proved that the average-case time complexity of sorting based on comparison operations, such as heap sort, merge sort, and quick sort, cannot be less than $O(N\log^{N})$ mathematically. Therefore, noncomparison sorting is the main method for complexity reduction. Several noncomparison sorting methods have been proposed. The most classical noncomparison algorithm is Radix_sort [9], which utilizes several buckets to preserve the original order of the keys and, then, maps them into a sorted sequence. Curiquintal et al. [10] presented an integer sorting algorithm called bit-index sort, which can achieve a time complexity close to $O(N)$ by utilizing a bit array to map the input integer elements to a sorted output sequence. However, this algorithm is unsuitable to other data types and has relatively slow speed in the case of high data repeatability. Wang [11] presented a noncomparison sorting method called self-indexed sort, which directly maps the element into a relative offset based on its value. However, this algorithm has difficulty obtaining a satisfactory optimization result on large-scale data due to the requirement of a large auxiliary space. Qureshi [12] proposed a method called Qureshi sort, which achieves the optimal time complexity $O(N)$ by using two additional arrays; however, its worst-case time complexity is $O(N^2)$, and it requires large auxiliary spaces. In general, noncomparison methods have two-common drawbacks: limited data types and large extra required spaces.

Machine learning has made remarkable achievements in various fields of computer science [18–22]. Ai et al. [23] proposed a dual learning algorithm for ranking; the algorithm jointly learns unbiased propensity models and ranking models from user clicking data without preprocessing. Google proposed a brilliant idea that speeds up indexes in databases by modeling the CDF of the input data using a neural network [13]. Subsequently, Google introduced a similar idea to speed up data sorting in databases and presented a new sorting algorithm (Google-sort) [14]. Google-sort initially predicts the position of each element of an unsorted dataset in a sorted sequence using a data distribution model built by CDF. Then, it places all unsorted elements into the corresponding positions in a result array. This algorithm has a significant performance benefit on large-scale data over comparison sorting algorithms. Placing elements in positions is the major time-consuming part, the theoretical time of which achieves complexity $O(N)$. Moreover, neural network can well adapt to various data types and can be easily combined with hardware-driven accelerating approaches. However, Google-sort has two problems that have been explained in Sections 1 and 3. This work is inspired by Google-sort. We try to design a new learned sorting algorithm by introducing the technique of LUT and a new data structure to solve the two problems mentioned above.

## 3. Design Problems

As mentioned in Section 1, two major difficult problems are encountered in learned sorting. This section introduces the basic workflow of learned sorting and provides a detailed explanation to the two problems.

*3.1. Learned Sorting Foundations.* The basic idea of learned sorting can be described by two steps: prediction and placement. The prediction step constructs a data distribution model of an unsorted dataset to predict the position of each unsorted element in a sorted sequence. The placement step places all unsorted elements into a result array according to the corresponding predicted placement positions. The prediction step can be further divided into two substeps. First, machine learning methods, such as neural network and linear regression (LR), are used to learn the distribution of an unsorted dataset and generate the corresponding distribution model, such as the CDF model, for predicting the CDF value of each element. Then, the CDF values are multiplied by the size of an unsorted dataset to obtain the placement positions of elements. For example, a dataset has 7 elements with key values [3, 4, 8, 7, 4, 8, 4]. The predicted CDF value of the first element with a value 3 is 0.138. We multiply 0.138 by 7 and round down the result to finally obtain the placement position 0 of the element in a sorted sequence.

*3.2. Data Distribution Model Construction.* The first problem is that the data distribution model for placement position prediction must satisfy the mathematical properties of CDF. Given an element of an unsorted dataset, its CDF value is defined as

$$v = P(X \leq \text{Key}), \quad (1)$$

where $v$ is the CDF value of the element, $X$ is any element in the array, Key is the value of the element, and $P$ is the likelihood to observe $X$ smaller or equal to Key [13].

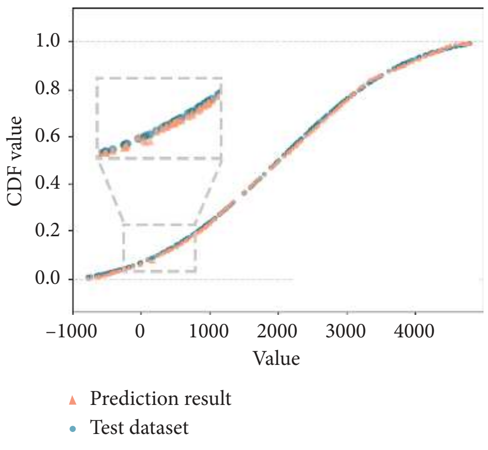
The data distribution model for placement position prediction can be defined as

$$p = F(\text{Key}) * N, \quad (2)$$

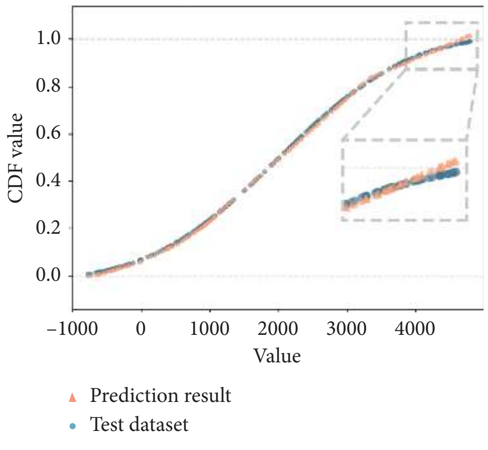
where $p$ is the position of an unsorted element, Key is the value of the element, $F(\text{Key})$ is the CDF value of the element with value Key, and $N$ is the total number of elements.

A straightforward method to obtain a CDF of an unsorted dataset is using a BP neural network, which is a common method to simulate functions with forms unknown due to the ability of fitting arbitrary functions and good anti-interference performance [24]. However, we find that such a CDF may violate two important mathematical properties of CDF: (1) monotonicity, that is, the CDF is monotonous and nondecreasing; (2) boundedness, that is, the range of CDF values belongs to [0, 1]. Figure 1 illustrates two typical unexpected examples. Figure 1(a) depicts the occurrence of one small local decline marked in the dashed box, which causes elements with large values to be incorrectly predicted as small ones. At the upper-right dashed box of Figure 1(b), a few pink triangle-shaped points exceed 1.0, which leads to the $F(\text{Key})$ greater than 1. As a result, the relevant elements will be predicted to nonexistent positions. Similar situations occur in the CDFs obtained from many machine learning models (Section 5.1). Accordingly, we need to find a new method to construct data distribution models that guarantee the monotonicity and boundedness of CDF.

*3.3. Placement Conflict.* The second problem is placement conflicts. A single element has a definite key value that determines an accurate placement position. However, some elements with the same value, which are called repetitive



(a)



(b)

Figure 1: Illustration of two examples violating the monotonicity (a) and boundedness (b) properties of CDF, respectively. In both plots, the $x$-axis presents the values of unsorted elements in the test dataset and the $y$-axis represents the values of CDF. In a plot, the pink triangle-shaped points depict the CDF values predicted by a BP neural network, the blue points present the true CDF values of unsorted elements in the test dataset, and the lower and upper gray lines are the lower and upper bounds of the CDF (i.e., 0 and 1), respectively.

elements, will be predicted to the same position. This condition causes placement conflicts. As shown in Figure 2, the unsorted dataset has three elements with the same value of 4. The three elements will be predicted to the second position from left to right in the sorted sequence. Surely, the second position can only be occupied by a single element.

Two common existing methods, namely, open hashing and open addressing, can solve the problem. Open hashing creates an empty linked list for a position to temporally store the elements that are assigned to the position [25]. This method is time and memory consuming due to massive memory request operations. Open addressing places elements assigned to the same position in closest empty positions [26]. The process of finding empty positions is also time consuming. Google-sort introduces another idea to solve placement conflicts. It processes repetitive elements by locally utilizing quick sort, which makes its time complexity approach to comparison sorting methods in the case the unsorted array has a high proportion of repetitive elements. Accordingly, the existing methods are all obsessed with large time or memory consumptions. Therefore, data placement in learned sorting is still challenging.

## 4. LS Algorithm

In this section, we detail our LS algorithm from three parts: a training data generation method, an LNN, and a self-regulation data structure.

*4.1. Training Data Generation Method.* Obtaining training data is an integral part of any machine learning algorithm. In many application scenarios, a portion of raw data is randomly selected as training data. However, such a training data generation method is unsuitable in our case. Training data randomly selected from raw data cannot accurately reflect the distribution of all unsorted elements; this incapability results in low accuracy of position prediction.

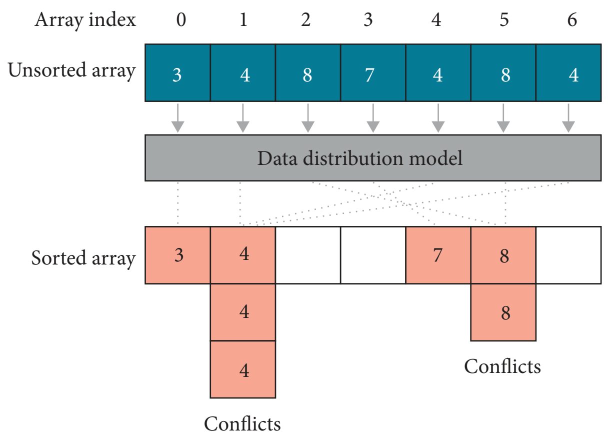


Figure 2: Illustration of placement conflicts.

We propose to generate the training data based on the frequency distribution and cumulative frequency distribution histograms of an unsorted array. First, we divide unsorted elements into equal intervals based on the range of values and calculate the proportion of elements in each interval to the total elements, that is, frequency of each interval, to obtain a frequency distribution histogram. Second, we obtain a cumulative frequency distribution histogram based on the frequency distribution histogram. Third, we take the rightmost value of each interval in the cumulative frequency distribution histogram and the corresponding cumulative frequency value to generate an input-output pair. In this way, we obtain a set of input-output pairs as our raw-labeled data, which has the same number of intervals and can accurately reflect the overall distribution of unsorted elements. Finally, we randomly select 25% of the raw-labeled data as training data. The rest 75% of the raw-labeled data is kept as test data to evaluate the accuracy of position prediction. Notably, inputs of these pairs may not be actual element values, and outputs of these pairs are in the range of [0, 1].

We take an unsorted array of 292 elements with values that range from −210 to 210 as an example (Figure 3). We divide the value range into 21 equal intervals and obtain the frequency distribution histogram (Figure 3(a)) and cumulative frequency distribution histogram (Figure 3(b)). A total of 21 input-output pairs, which are marked with red crosses, will be extracted from the cumulative frequency distribution histogram to generate the raw-labeled data. For instance, the first cross from the left is the first pair with an input of −190 and an output of 0.075, and the second cross from the left is the second pair with an input of −170 and an output of 0.127. Then, we randomly select 25% of the raw-labeled data as training data.

*4.2. Look-Up-Table Neural Network.* We use neural network to learn the training data for obtaining a proper CDF model and, then, predict CDF values for unsorted elements. A neural network generally consists of an input layer, several hidden layers, and an output layer. An input value will start from the input layer to hidden layers and finally end at the output layer to obtain the output value [27, 28]. Such a process is called a computing link. We need to guarantee that the relationship of input and output of each link in a neural network is monotonic and bounded. This issue is the first problem mentioned in Section 3; that is, constructing a data distribution model that satisfies the mathematical properties of CDF. Our idea is to introduce the technique of LUT into the traditional BP neural network [29, 30]. We call such a neural network as LNN. Figure 4 illustrates a BP neural network and an LNN. The BP neural network (Figure 4(a)) obtains the output value of a link by calculating the values of nodes using weights (e.g., $w_{ij}$ and $w_{jk}$) between neurons, while the LNN uses an LUT to replace the weights of a link to ensure that the process result is monotonic and bounded.

LUT is a control technique of data mapping. LUT allows using a predefined monotonicity and output range to rule the mapping of inputs to outputs [31]. We take an LUT with input values within [1, 10] and output values within [1, 10] as an example (Figure 5) to demonstrate how the LUT guarantees that the relationship between inputs and outputs is nondecreasing and bounded. Five LUT control points (dark-gray points) are shown from left to right in Figure 5. Five control points are connected by straight lines. Thus, a piecewise input-output relationship is formed. This LUT guarantees the mathematical properties of the relationship from three aspects. First, the up and low bounds of the relationship are limited between 0 and 1. Second, the output of any node is not less than that of its left node. Thus, the relationship is nondecreasing. Third, we can fit any complex input-output relationship by increasing the number of control points. The numbers of control points, input values, and output values of control points in this case are manually prespecified, while they are automatically determined by the training process of the neural network in actual experiments.

Next, we need to determine the number of LNN's hidden layers and the number of nodes contained in each layer. Considering the balance between time consumption and fitting accuracy, we decide to utilize an LNN containing one hidden layer with 20 nodes. A large number of hidden layers indicate precise fitting but large time consumption. Our sorting needs to model the distribution of unsorted data, which is a simple task for neural network. We conduct a large number of pilot experiments and find that an LNN of one hidden layer with 20 nodes can obtain an ideal result in most cases. Moreover, we set the input and output layers to have only one node because we assume that unsorted elements are a single-dimensional real number in this study.

The following part introduces how we use an LNN with single input and output nodes and one hidden layer of 20 nodes to build a model for predicting CDF values of unsorted elements. As shown in Figure 4(b), the value of an unsorted element, that is, $x_1$, enters the input layer and obtains the corresponding output value $z_j$ through $\mathrm{LUT}_{ij}$. Next, $z_j$ obtains $\alpha_j$ through an activation function, and then, $\alpha_j$ obtains the corresponding output value $y_1$ through $\mathrm{LUT}_{jk}$ [32]. Finally, $y_1$ obtains the output value through an activation function, that is, $x_1$'s CDF value $\beta_1$. The formulas are shown as follows:

$$z_j = \sum_{i=1} \mathrm{LUT}_{ij}(x_1), \tag{3}$$

$$\alpha_j = f(z_j), \tag{4}$$

$$y_1 = \sum_{j=1} \mathrm{LUT}_{jk}(\alpha_j), \tag{5}$$

$$\beta_1 = f(y_1). \tag{6}$$

$\mathrm{LUT}_{ij}$ in formula (3) is an LUT between input layer $i$ and hidden layer $j$. $f$ in formulas (4) and (6) is the commonly used activation function sigmoid for realizing nonlinear fitting of a neural network. $\mathrm{LUT}_{jk}$ in formula (5) is an LUT between hidden layer $j$ and output layer $k$. An LNN's computing link has two kinds of computations, namely, LUT and sigmoid computations. The monotonicity and boundedness of an LUT can be guaranteed by setting constraints before training, and the sigmoid function is nondecreasing with the range [0, 1]. Thus, all computing links of LNN satisfy the condition, which means the output of LNN is monotonic and bounded. Figure 6 shows the experiment result of an example using LNN. The CDF values in Figure 6 do not violate monotonicity and boundedness.

*4.3. Placement Method Based on the Self-Regulating Index.* After an LNN is prepared, we use the data distribution model built by the LNN to predict the placement positions of unsorted elements and, then, place them into a result array in an ascending order. We will encounter the second problem mentioned in Section 3, that is, placement conflicts. A data structure called the SRI is designed to solve the challenge. SRI is a data structure consisting of a memory space and a set of data operations. The memory space is an array of the same type and size as unsorted elements, that is, the index array. The data operations include memory space

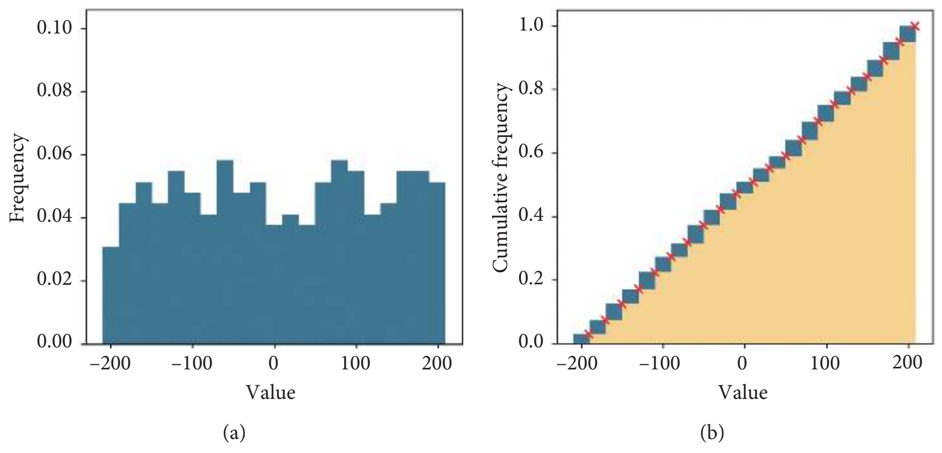


FIGURE 3: Illustration of training data generation. The $x$-axes of both histograms represent values in raw data. In the frequency distribution histogram (a), the $y$-axis represents the frequency of each interval. In the cumulative frequency distribution histogram (b), the $y$-axis represents the total frequency of the present interval (blue) and all intervals on the left (yellow).

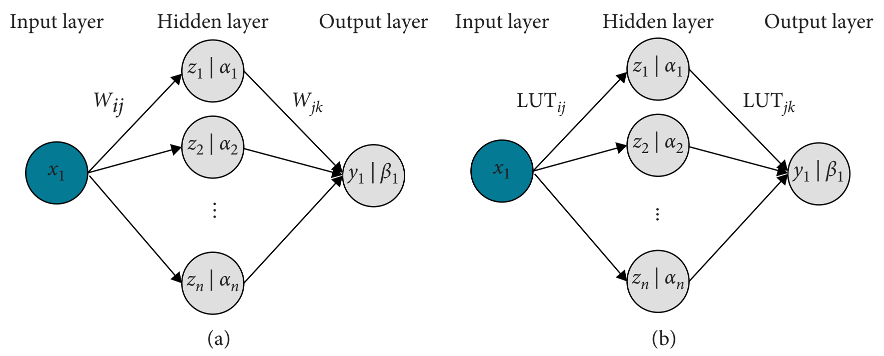


FIGURE 4: Illustration of the traditional BP neural network (a) and LNN (b). Both plots have only one input node $x_1$, one output node $y_1$, and one hidden layer in neural network. $z_j$ represents hidden layer nodes. $\text{LUT}_{ij}$ and $\text{LUT}_{jk}$ in Figure 4(b) are used to replace weights $w_{ij}$ and $w_{jk}$ in Figure 4(a).

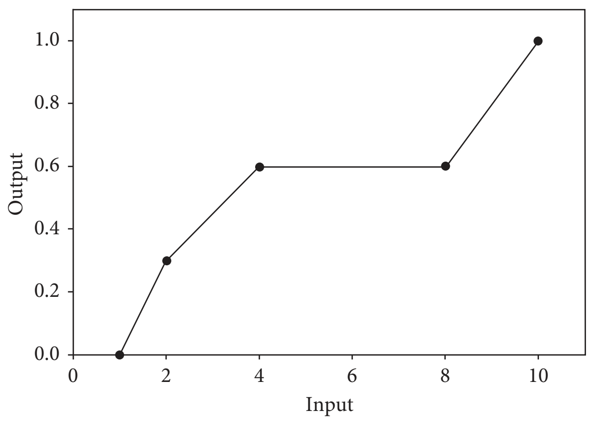


FIGURE 5: Illustration of an LUT with 5 control points. The $x$-axis presents the input value of the LUT, and the $y$-axis presents the output value of the LUT. The entire line presents the input-output relationship of the LUT, in which dark-gray points present control points.

initialization, index array assignment, and index array self-regulation. In the following, we use an example shown in Figure 7 to specialize the three operations.

*4.3.1. Memory Space Initialization.* This operation prepares three arrays with the same size of unsorted elements. As shown in Figure 7(a), an original array (OArray) stores raw unsorted elements. In this case, it has four elements with values [16, 16, 18, 11]. An index array (IArray) is a set with initial values [−1, −1, −1, −1]. An empty result array (RArray) is prepared for storing sorted elements. The indices of all arrays start from 0 upward. For example, OArray[0] represents the first element of the original array, and its key value is 16.

*4.3.2. Index Array Assignment.* For each element in the OArray, this operation obtains its placement position in the RArray predicted by the data distribution model and assigns the position as the value to the IArray element with an index equal to the position. As illustrated in Figure 7(b), the values of OArray[0] and OArray[1] are both 16, and their predicted placement positions are both 1. Thus, IArray[1] is set to 1. The value of OArray[2] is 18 and is predicted as 3. Accordingly, IArray[3] is set to 3. The value of OArray[3] is 11 and is predicted as 0. As a result, IArray[0] is set to 0.

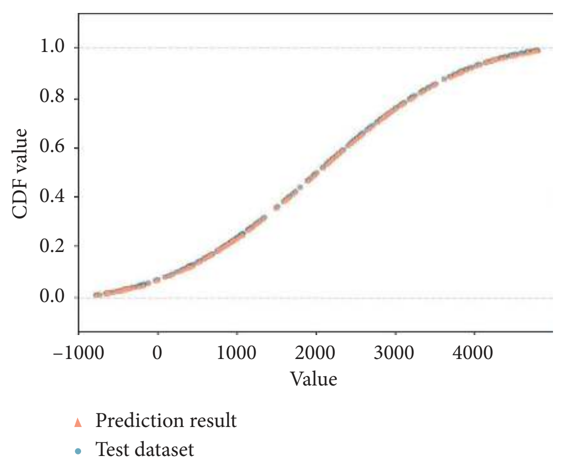


Figure 6: Illustration of a CDF example obtained by LNN. The $x$-axis represents values of unsorted elements in the test dataset, and the $y$-axis represents CDF values. In the plot, the pink triangle-shaped points depict the CDF values predicted by LNN, the blue points present the true CDF values of unsorted elements in the test dataset, and the lower and upper gray lines are the lower and upper bounds of the CDF (i.e., 0 and 1), respectively. The test dataset in this figure is the same as that in Figure 1.

*4.3.3. Index Array Self-Regulation.* This operation utilizes the predicted placement positions that are tentatively stored and self-regulated in the IArray to place all OArray elements into the RArray in an ascending order. For each element in the OArray, we take its placement position predicted as the target index of the IArray. Then, we take the element value of the IArray under the target index as the actual placement position of the OArray element in the RArray. After the OArray element is placed into the RArray in accordance with the actual placement position, the element value of the IArray is incremented by 1. The abovementioned process is called self-regulation. It ensures that repetitive OArray elements, which have the same predicted positions and, thus, result in placement conflicts, can be placed at adjacent empty positions in the RArray. After sequentially executing the self-regulation operation on all elements in the OArray, we eventually obtain the sorted RArray.

We take Figures 7(c)–7(h) as an example to illustrate the self-regulation operation. For OArray[0], its value is 16, its predicted position is 1, and IArray[1] is 1, which indicates that the tentative and actual placement positions of OArray [0] in the RArray are both 1. Then, the self-regulation operation assigns the value 16 of OArray[0] to RArray[1] (Figure 7(c)) and increments the value of IArray[1] by 1. Figure 7(d) shows that the current value of IArray[1] is 2. That is, if a repetitive OArray element with a value of 16 occurs, then it will be placed at RArray[2]. OArray[1] is a repetitive element of OArray[0] with a value of 16 and has a predicted position of 1 again. However, the value of IArray [1] is 2 now. Thus, the self-regulation operation assigns the second 16 to RArray[2] (Figure 7(e)) and increment IArray [1] by 1. Figure 7(f) shows that the current value of IArray[1] is 3. Thereafter, the value of OArray[2] is 18 predicted as 3 while IArray[3] is 3. As a result, 18 is assigned to RArray[3] (Figure 7(f)) and IArray[3] is incremented by 1, as shown in Figure 7(g). Finally, the last element OArray[3] with a value of 11 is predicted as 0 while IArray[0] is 0. Thus, it is placed at RArray[0] (Figure 7(g)), and IArray[0] becomes 1 in Figure 7(h). Now, all unsorted OArray elements are stored in the RArray in an ascending order, as shown in Figure 7(h).

The complexities of three main data operations of LS algorithm, namely, memory space initialization, index array assignment, and index array self-regulation, are all $O(N)$. LS algorithm only involves a simple memory allocation operation, that is, memory is initially allocated only once. With regard to the two problem-solving methods for placement conflicts introduced in Section 3, open hashing is memory consuming due to massive memory request operations, while open addressing is time consuming owing to the laborious calculation of finding empty positions.

## 5. Evaluation

We conduct three experiments to evaluate the effectiveness of the proposed LS algorithm. The first experiment is designed to test whether the LNN method can guarantee the properties of CDF. The second experiment evaluates the time consumption of LS by taking Google-sort and popular sorting algorithms as the references. The third experiment evaluates the stability of Google-sort and LS under different rates of repetitive elements. All the three experiments are conducted on a MacBook Air with a 1.8 GHz Intel Core i5 CPU and 8 GB 1600 MHz DDR3.

*5.1. Performance of the LNN in Different Distributions.* This experiment aims to test whether the proposed LNN method can effectively guarantee the mathematical properties of CDF, namely, monotonicity and boundedness. This experiment consists of three steps. (1) We generate 36 synthetic unsorted datasets that present 9 distributions on 4 scale levels, namely, 10000, 100000, 1 million, and 10 million, as raw datasets. Figure 8 shows examples of the 9 distributions with the scale of 10 million. Then, we obtain the corresponding 36 training datasets and 36 test datasets by using the training data generation method mentioned in Section 4.1. (2) We input each training dataset into the LNN and three widely used prediction methods to train predictive models. The three reference methods are adaptive boosting (AdaBoost), a fully-connected neural network (DNN), and LR. A total of 144 predictive models (36 training datasets × 4 prediction methods) are obtained. (3) We use each predictive model to predict the CDF values of the elements of the corresponding test dataset in an ascending order. Eventually, we obtain 144 predicted CDF value sets as experimental results.

We analyze the experimental results in three aspects. (1) We check the violation of monotonicity. That is, we examine the existence of prediction errors that the predicted CDF values of elements with larger values are less than those with smaller values. (2) We check the violation of boundedness. That is, the existence of the predicted CDF values outside the range [0, 1] is reviewed. (3) We conduct a statistical analysis grouped by the prediction method on the rate of violating

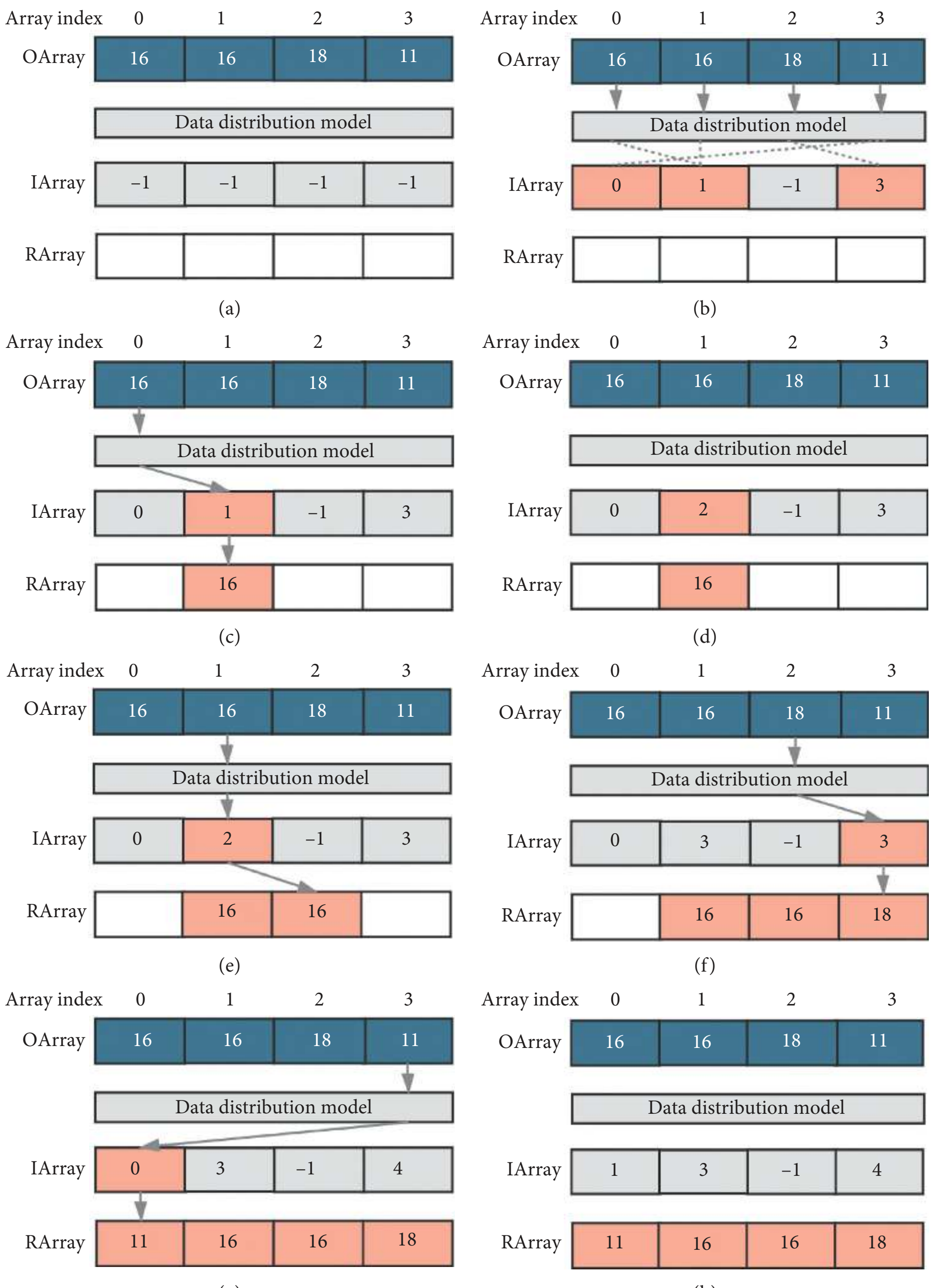


FIGURE 7: Illustration of the proposed placement solution. The array elements highlighted in pink show the changes during the placement.

monotonicity (RVM) and rate of violating boundedness (RVB). For instance, among the 36 predicted CDF value sets fitted by the DNN, if 2 sets break the monotonicity and 17 sets break the boundedness, then the RVM and RVB of DNN are 47.2% (17/36) and 5.5% (2/36), respectively. The RVM and RVB of the four methods are listed in Table 1.

The analysis results, as shown in Table 1 and Figure 9, demonstrate that the LNN obtains a 0% RVM and a 0% RVB. This result indicates that the LNN can fully guarantee the monotonicity and boundedness properties of CDF. A result example of LNN is shown in Figure 9(a). The LR derives an RVB up to 88.9% due to its linear limitation, while most raw datasets are nonlinearly distributed. As shown in Figure 9(b), the pink triangle-shaped points at the lower-left and upper-right corners are invalid CDF values out of the range of [0, 1] on the same dataset in Figure 9(a). The AdaBoost obtains an RVB and an RVM of 0%, but we find that the AdaBoost model possibly causes a staircase phenomenon. As shown in Figure 9(c), the pink triangle-shaped points form two local staircases marked in the dashed boxes. Based on regression tree methods, the AdaBoost outputs the mean instead of the continuous predicted CDF values, which

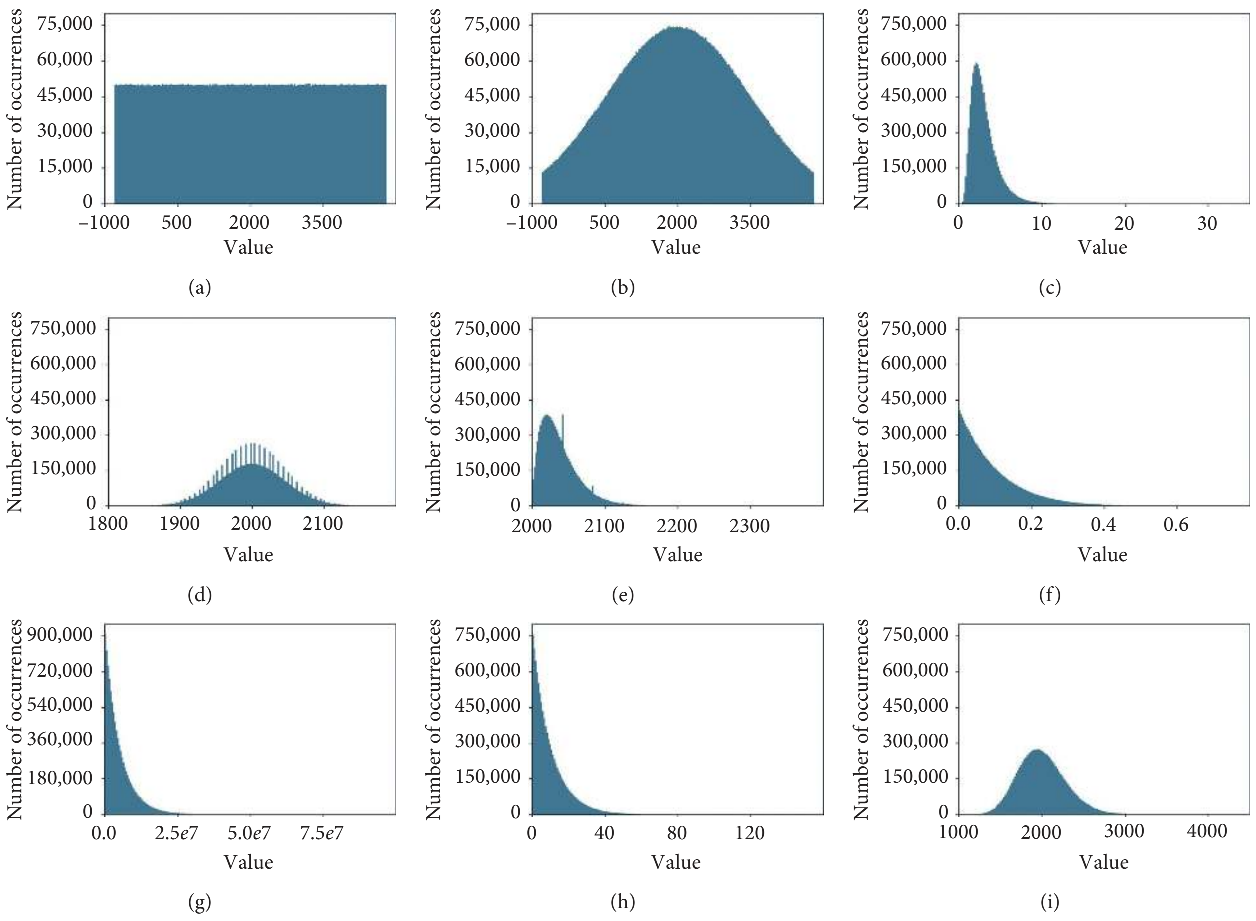


Figure 8: Examples of 9 distribution histograms of raw experimental datasets with the scale of 10 million: (a) a uniform distribution in the range of $[-800, 4800]$, (b) a normal distribution with $\mu = 2000$, $\sigma = 1500$, (c) a log normal distribution with $\mu = 2000$, $\sigma = 1500$, (d) a Poisson distribution with $\lambda = 2000$, (e) a negative binomial distribution with $r = 2.17$, $p = 0.055$, (f) a beta distribution with $\alpha = 1$, $\beta = 10$, (g) an exoinential distribution with $\lambda = 2000$, (h) a gamma distribution with $\alpha = 1$, $\beta = 10$, and (i) a Wald distribution with $\mu = 2000$.

Table 1: Rates of violating monotonicity (RVM) and violating boundedness (RVB) of the four prediction methods.

| Prediction methods | RVM (%) | RVB (%) |
|---|---|---|
| LNN | 0 | 0 |
| LR | 0 | 88.9 |
| AdaBoost | 0 | 0 |
| DNN | 5.5 | 47.2 |

causes the elements with adjacent values to be predicted to the same CDF value [33, 34]. The RVB and RVM of DNN reach 47.2% and 5.5%, respectively. This result implies that the DNN dissatisfies the monotonicity and boundedness. Figure 9(d) depicts the occurrence of one small local decline marked in the dashed box. In summary, the prediction sets of all test datasets made by the LNN effectively satisfy the two mathematical properties of CDF, but the three reference methods fail.

*5.2. Performance in the Average Case.* The second experiment is designed to evaluate the sorting time of two learned sorting algorithms (i.e., LS and Google-sort algorithms) and three popular sort algorithms (i.e., heap sort, quick sort, and merge sort). In theory, the two learned sorting algorithms have the $O(N)$ time complexity and the three popular sort algorithms have the $O(N \log N)$ time complexity. We select five data sizes from 10 million to 50 million referring to the experiment settings in the study of Google-sort; we also randomly generate 10 raw datasets for each data size (50 raw datasets in total) [14]. For the three popular sort algorithms, we use each of them to sort each raw dataset in an ascending order and calculate the average time consumptions by algorithm and data size. For each of the two learned sorting algorithms, we perform two steps on each raw dataset. The first step is data preparation. A training dataset and a predictive model are obtained using the methods mentioned in Sections 4.1 and 4.2, respectively. The second step is to sort the raw dataset in an ascending order. We record the time consumption of each sorting and calculate the average time consumptions by algorithm and data size. As a result, 25 average time consumptions (5 data sizes $\times$ 5 sorting algorithms) are obtained. The result is given in Table 2 and Figure 10.

The results in Table 2 show that the quick sort outperforms the merge sort and heap sort. The quick sort is 26% faster than merge sort and 174% faster than heap sort

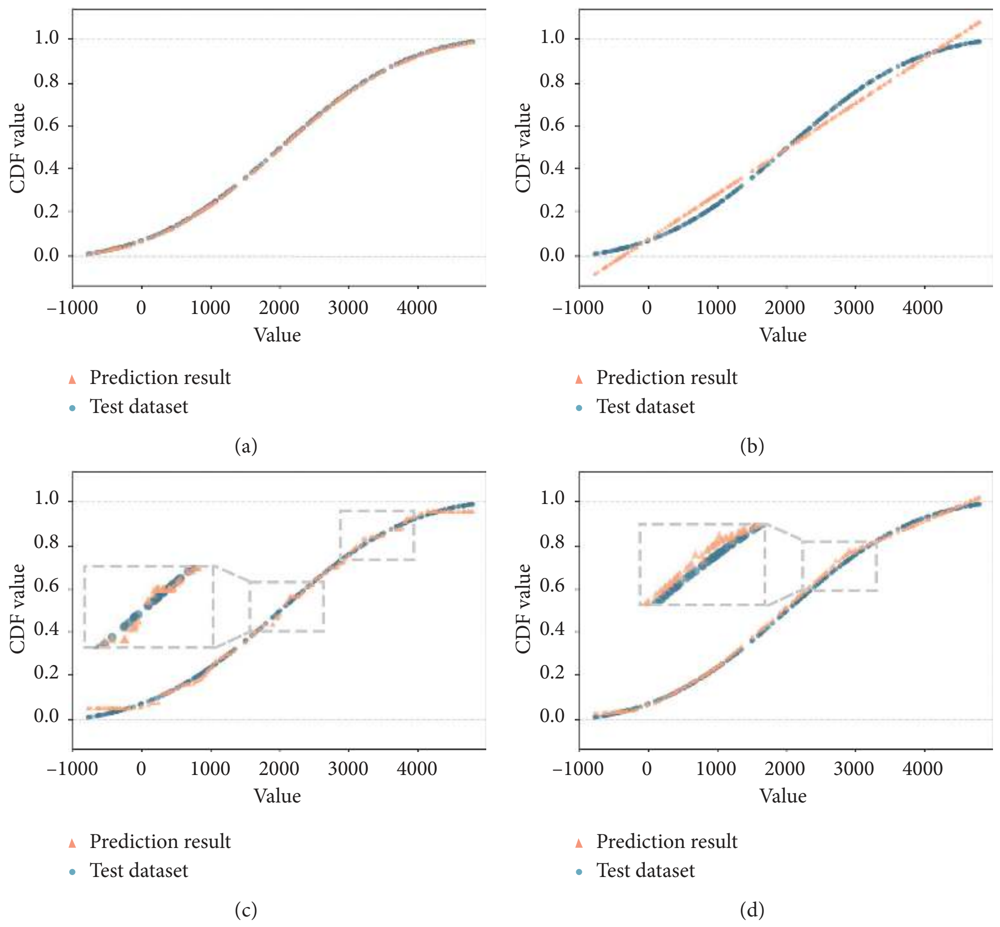


Figure 9: Illustration of some experimental results. In all charts, the $x$-axis represents the values of unsorted elements in the test dataset and the $y$-axis represents their CDF values. All charts are drawn based on the same dataset in normal distribution with a 1 million scale. The pink triangle-shaped points depict the CDF values predicted by predictive models, the blue points present the true CDF values of unsorted elements in test datasets, and the lower and upper gray lines are the lower and upper bounds of the CDF (i.e., 0 and 1), respectively. (a) LNN model, (b) LR model, (c) Ada Boost model, and (d) DNN model.

Table 2: Average time consumptions of five sorting algorithms on five experimental data sizes (in seconds).

| Data size | Heap sort (s) | Merge sort (s) | Quick sort (s) | Google-sort | LS |
|---|---|---|---|---|---|
| 10 million | 4.12 | 2.17 | 1.66 | 0.91 | 1.18 |
| 20 million | 9.25 | 4.32 | 3.52 | 2.45 | 2.62 |
| 30 million | 14.71 | 6.82 | 5.34 | 4.32 | 4.54 |
| 40 million | 21.34 | 9.27 | 7.30 | 6.63 | 6.08 |
| 50 million | 26.88 | 11.29 | 9.19 | 8.56 | 7.61 |

on average. Figure 10 depicts that the time consumptions of the three sort algorithms grow steadily as the data size increases. Among them, the growth trend of time consumption of quick sort is the slowest. Compared with the quick sort, the results in Table 2 show that our LS has a significant performance benefit and an average margin of 27% faster execution than the quick sort. Compared with the time consumption of Google-sort, the time consumption of LS is very close to that of Google-sort when the data size is lower than 30 million. However, the LS outperforms the Google-sort when the data size is greater than 30 million. Figure 10 shows that the growth trend of time consumption of Google-sort suddenly speeds up at the data size of 30 million, and the time consumption of Google-sort is close to that of quick sort at the data size of 50 million. We speculate that this phenomenon is caused by the manner of processing repetitive elements of Google-sort. The Google-sort utilizes the quick sort to process repetitive elements. When the proportion of repetitive elements increases with the data size, the sorting time of Google-sort would significantly increase. In summary, this experiment demonstrates that the proposed LS algorithm can achieve a satisfied sorting efficiency on large-scale datasets.

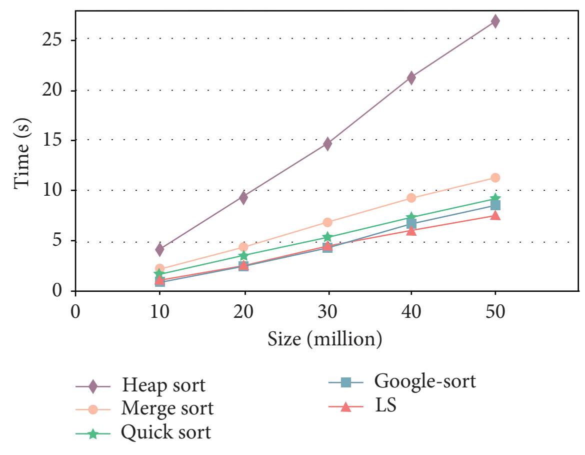


Figure 10: Trends of average time consumptions of five sorting algorithms. The $x$-axis represents the size of experimental datasets (in millions), and the $y$-axis represents the sorting time in the average case (in seconds).

*5.3. Performance in Different Repetitive Element Rates.* The last experiment investigates whether our LS algorithm can effectively solve placement conflicts. A specific indicator, named the repetitive element rate (RER), is defined for this experiment. It represents the ratio of repetitive elements to all elements in a dataset. In this experiment, we select five RERs (i.e., 0%, 20%, 40%, 60%, and 80%) and randomly generate 10 raw datasets for each RER on the fixed data size of 20 million (50 raw datasets in total) as the experimental datasets. By taking a raw dataset with 20% RER and 20 million data size as an example, we generate the raw dataset in the following steps: (1) 16 million unsorted elements with different element key values are randomly generated. (2) One element is randomly selected from the 16 million elements, and this step is randomly repeated 0–10 times. (3) Step 2 is repeated until we gather 4 million repetitive elements. We ensure that the selected element from the 16 million elements is different for each time. (4) The 4 million elements are randomly inserted into the 16 million elements. We select Google-sort as the reference algorithm. Using the same experimental process of the second experiment, we obtain 10 average time consumptions (2 sorting algorithms × 5 RERs) as the results. The results are shown in Table 3 and Figure 11.

The results show that the time consumptions of LS only fluctuate 1.9% as the RER increases from 0% to 80%. By contrast, the Google-sort consumes more time as the RER increases. At an RER of 60%, the time consumption of Google-sort reaches 3.65 s, which is even slower than that of quick sort recorded in Table 2 (3.52 s with a scale of 20 million). This experiment confirms that the RER greatly negatively affects the sorting time of Google-sort. When faced with high RERs, the time complexity of Google-sort would approach $O(N\lg N)$ because of using quick sort to solve placement conflicts. In summary, our algorithm can achieve a stable performance when the proportion of repetitive elements in a dataset increases.

Table 3: Average time consumptions of LS and Google-sort algorithms on five RERs (in seconds).

| RER (%) | LS (s) | Google-sort (s) |
|---|---|---|
| 0 | 2.62 | 2.45 |
| 20 | 2.62 | 2.98 |
| 40 | 2.65 | 3.32 |
| 60 | 2.62 | 3.65 |
| 80 | 2.60 | 3.92 |

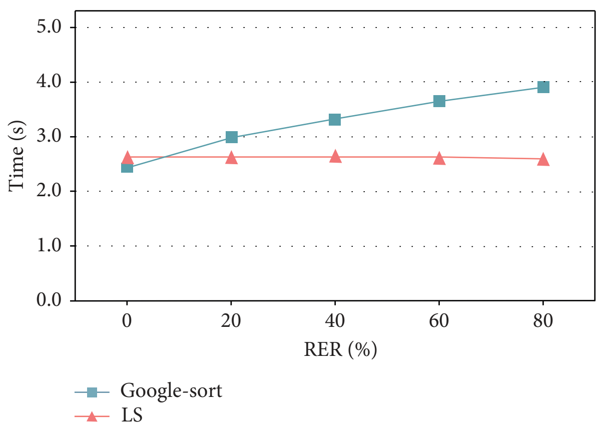


Figure 11: Trends of average time consumptions of LS and Google-sort algorithms with five RERs. The $x$-axis represents the RER of experimental datasets (%), and the $y$-axis represents the sorting time in the average case (in seconds).

## 6. Discussion

In this section, we discuss the limitations of the proposed LS algorithm and give suggestions for further work.

We select to use an LNN with only one hidden layer of 20 nodes, which is determined based on our experience gained from pilot experiments. Notably, this simple neural network may be challenged in complex sorting scenarios, such as complex data distribution and multidimensional sorting. However, increasing the number of hidden layers and nodes will consume much time for model training. Similarly, we sample 1000 points as the raw-labeled dataset when generating training data. This empirical setting may not be the absolute best number of sampling points balancing time and accuracy. Moreover, trained models require generalizability to new data. In our experiments, we train a predictive model to learn the distribution of each raw dataset, where overfitting is a good thing. Maybe we can explore inner relationships between raw datasets, which can be exploited to speed up the training process of predictive models.

Placement conflicts are mainly caused by repetitive elements. In practice, some elements with different values may be predicted to the same position, which results in placement conflicts. This situation occurs under two conditions: (1) a dataset has elements from a very large domain. (2) Most elements of the dataset are concentrated in a small area of the domain space. Raw predicted placement positions are double type. Raw predicted placement positions of elements concentrated in such a small area would be extremely close to each other. When rounding raw positions for placement, a few elements with adjacent values would be placed to the same position of the sorted array. This situation is rarely observed in our experiments but is worth considering. To deal with the problem, we can sort the raw dataset in accordance with the process of LS. Then, we can use insertion sort, which works very fast with almost sort array, to adjust the sorted array. This problem is worthy of optimization in the future.

When generating experimental datasets with repetitive elements in Section 5.3, we make the values of repetitive elements as evenly distributed as possible. This case is the worst for learned sorting because sorting a dataset with all the repetitive elements having only one key value is easy. In the experiments, we do not test the case that sorts a backward sequence to a forward sequence because the order of a dataset does not affect the performance of learned sorting. Moreover, our experiments are conducted on a normal CPU to compare the performances of traditional comparison sorting algorithms and learned sorting algorithms under common circumstances. Notably, the data preparation time of LS and Google-sort is excluded to ease the establishment of the same experimental conditions. We believe that our LS algorithm can achieve a better result with high-performance hardware. Of course, we should consider the balance between time consumption and economic consumption.

This study limits the scope to single-dimensional sorting. However, application scenarios that need multidimensional sorting often occur. Two potential methods can be used to improve our LS algorithm to sort multidimensional data. (1) Sort by dimension: this method sorts values of each dimension until all dimensions are sorted. (2) Building the distribution model of multidimensional data to enable LS to predict the placement positions of multidimensional data and place them without placement conflicts. Our experiments only include real number sorting. Many other types of data sorting, such as imaginary number sorting, are available. When dealing with such type of data sorting, we can think of the dataset as a two-dimensional dataset. One dimension stores the values of the real part, and the other dimension stores the values of the imaginary part. Thus, we can use the two aforementioned methods to sort the two-dimensional dataset.

## 7. Conclusions

In this study, we comprehensively analyzed two potential problems, namely, CDF property violations and placement conflicts, in applying machine learning models for data sorting. We proposed to integrate a BP neural network with the technique of LUT to avoid CDF property violations. We designed an SRI data structure to eliminate placement conflicts. We proposed a new learned sorting algorithm named LS based on the two manners. The results of three controlled experiments demonstrate that LS can completely solve the two problems and can achieve a satisfied and stable performance on large-scale data. This study moves the idea of learned sorting forward after it was first raised by Google. This study can inspire researchers and engineers that learned approaches have great potentials in improving fundamental methods in computer sciences.

## Data Availability

Data can be obtained from the corresponding author upon request.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work was supported in part by the National Key Research and Development Program of China (No. 2018YFB1700403), the National Natural Science and Technology Fundamental Resources Investigation Program of China (No. 2018FY10090002), and the National Natural Science Foundation of China (No. 61872388).

## References

[1] R. Ding, Q. Wang, Y. Dang, Q. Fu, H. Zhang, and D. Zhang, "YADING: fast clustering of large-scale time series data," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 473–484, 2015.

[2] S. Geuens, K. Coussement, and K. W. De Bock, "A framework for configuring collaborative filtering-based recommendations derived from purchase data," *European Journal of Operational Research*, vol. 265, no. 1, pp. 208–218, 2018.

[3] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Proceedings of 2010 IEEE International Parallel and Distributed Processing Symposium*, IEEE, Atlanta, GA, USA, pp. 1–10, April 2010.

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: a unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[5] Z. Marszalek, "Parallelization of modified merge sort algorithm," *Symmetry*, vol. 9, no. 9, pp. 176–194, 2017.

[6] N. Pang, D. Zhu, Z. Fan, W. Rong, and W. Feng, "A large-scale distributed sorting algorithm based on cloud computing," *Communications in Computer and Information Science*, vol. 557, pp. 226–237, 2015.

[7] F. Gebali, M. Taher, A. M. Zaki, M. W. El-kharashi, and A. Tawfik, "Parallel multidimensional lookahead sorting algorithm," *IEEE Access*, vol. 7, pp. 75446–75463, 2019.

[8] N. Faujdar and S. P. Ghrera, "A practical approach of GPU bubble sort with CUDA hardware," in *Proceedings of International Conference on Cloud Computing*, pp. 7–12, Noida, India, January 2017.

[9] Radix_sort, https://en.wikipedia.org/wiki/Radix_sort, 2020.

[10] L. F. Curiquintal, J. O. Cadenas, and G. M. Megson, "Bit-index sort: a fast non-comparison integer sorting algorithm for permutations," in *Proceedings of 2013 the International Conference on Technological Advances in Electrical Electronics and Computer Engineering*, IEEE, Konya, Turkey, pp. 83–87, May 2013.

[11] S. Y. Wang, "A new sort algorithm: self-indexed sort," *ACM Sigplan Notices*, vol. 31, no. 3, pp. 28–36, 1996.

[12] M. A. Qureshi, "Qureshi sort: a new sorting algorithm," in *Proceedings of 2009 2nd International Conference on*

*Computer, Control and Communication*, IEEE, Karachi, Pakistan, February 2009.

[13] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of International Conference on Management of Data*, pp. 489–504, Houston, TX, USA, June 2018.

[14] T. Kraska, M. Alizadeh, A. Beutel et al., "SageDB: a learned database system," in *Proceedings of Conference on Innovative Data Systems Research*, Asilomar, California, January 2019.

[15] M. B. Ludwig, H. M. Chehreghani, and J. M. Buhmann, "The information content in sorting algorithms," in *Proceedings of 2012 IEEE International Symposium on Information Theory*, IEEE, Cambridge, MA, USA, July 2012.

[16] E. Strubell, A. Ganesh, and A. Mccallum, "Energy and policy considerations for deep learning in NLP," 2019, https://arxiv.org/abs/1906.02243.

[17] H. C. Thomas, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[18] Y. Zhao, X. Luo, X. Lin et al., "Visual analytics for electromagnetic situation awareness in radio monitoring and management," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 590–600, 2020.

[19] F. Zhou, X. Lin, C. Liu et al., "A survey of visualization for smart manufacturing," *Journal of Visualization*, vol. 22, no. 2, pp. 419–435, 2019.

[20] O. Abedinia and N. Amjady, "Net demand prediction for power systems by a new neural network-based forecasting engine," *Complexity*, vol. 21, no. S2, pp. 296–308, 2016.

[21] H. Bao, J. H. Park, and J. Cao, "Synchronization of fractional-order delayed neural networks with hybrid coupling," *Complexity*, vol. 21, no. S1, pp. 106–112, 2015.

[22] R. Li, J. Cao, A. Alsaedi, B. Ahmad, F. E. Alsaadi, and T. Hayat, "Nonlinear measure approach for the robust exponential stability analysis of interval inertial Cohen-Grossberg neural networks," *Complexity*, vol. 21, no. S2, pp. 459–469, 2016.

[23] Q. Ai, K. Bi, C. Luo, J. Guo, and W. B. Croft, "Unbiased learning to rank with unbiased propensity estimation," The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, July 2018.

[24] X. Li, H. Xie, R. Wang et al., "Empirical analysis: stock market prediction via extreme learning machine," *Neural Computing and Applications*, vol. 27, no. 1, pp. 67–78, 2016.

[25] N. Sun and R. Nakamura, "An alternative analysis of the open hashing algorithm," in *Proceedings of International Conference on Applied Mathematics*, pp. 1–6, Chalkis, Greece, September 2004.

[26] J. I. Munro and P. Celis, "Techniques for collision resolution in hash tables with open addressing," in *Proceedings of ACM Fall Joint Computer Conference*, IEEE Computer Society Press, Los alamitos, CA, USA, pp. 601–610, November 1986.

[27] A. Abels, D. M. Roijers, T. Lenaerts, A. Nowé, and D. Steckelmacher, "Dynamic weights in multi-objective deep reinforcement learning," in *Proceeding of the 36th International Conference On Machine Learning*, pp. 11–20, Long Beach, CA, USA, May 2018.

[28] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[29] M. M. Fard, K. R. Canini, A. Cotter, J. Pfeifer, and M. R. Gupta, "Fast and flexible monotonic functions with ensembles of lattices," in *Proceedings of 29th Conference on Neural Information Processing Systems*, pp. 2919–2927, Barcelona, Spain, December 2016.

[30] S. You, D. Ding, K. R. Canini, J. Pfeifer, and M. R. Gupta, "Deep lattice networks and partial monotonic functions," in *Proceedings of 31st Conference on Neural Information Processing Systems*, pp. 2985–2993, Mountain View, CA, USA, May 2017.

[31] M. A. Hasan, "Look-up table-based large finite field multiplication in memory constrained cryptosystems," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 749–758, 2000.

[32] A. K. Jain, J. Jianchang Mao, and K. M. Mohiuddin, "Artificial neural networks: a tutorial," *Computational Science and Engineering*, vol. 29, no. 3, pp. 31–44, 1996.

[33] G. Rätsch, T. Onoda, and K.-R. Müller, "Soft margins for AdaBoost," *Machine Learning*, vol. 42, no. 3, pp. 287–320, 2001.

[34] V. Kuznetsov, M. Mohri, and U. Syed, "Multi-class deep boosting," in *Proceeding of 28th Conference on Neural Information Processing Systems*, Montreal, Canada, December 2014.