Deep Learning Service for Efficient Data Distribution Aware Sorting

Xiaoke Zhu[†]
Beihang University
China
zhuxk@buaa.edu.cn

Qi Zhang
Meta Platforms
USA
qizhang@meta.com

Wei Zhou*
Yunnan University
China
zwei@ynu.edu.cn

Ling Liu
Georgia Institute of Technology
USA
ling.liu@cc.gatech.edu

***Abstract*—In this paper, we present a neural network-enabled data distribution aware sorting method, coined as NN-sort. Our approach explores the potential of developing deep learning techniques to speed up large-scale sort operations, enabling data distribution aware sorting as a deep learning service. Compared to traditional pairwise comparison-based sorting algorithms, which sort data elements by performing pairwise operations, NN-sort leverages the neural network model to learn the data distribution and uses it to map large-scale data elements into ordered ones. Our experiments demonstrate the significant advantage of using NN-sort. Measurements on both synthetic and real-world datasets show that NN-sort yields 2.18× to 10× performance improvement over traditional sorting algorithms.**

I. Introduction

Sorting is a fundamental operation in the realm of big data, powering everything from database systems [1] to bigdata analysis [2]. As the scale of data continues to grow, traditional sorting algorithms face increasing limitations in performance. While methods such as Quick Sort family [3], Merge Sort family [4], and Radix Sort family [5], [6] have long been relied upon, their comparison-based and non-comparison-based optimizations appear to be reaching its bottleneck.

Recent research [7]–[10] has extensively explored how deep learning models can enhance the performance of traditional big data systems and algorithms. For instance, Tim Kraska et al. introduced a learned index [9] that leverages an empirical cumulative distribution function (CDF) to improve the performance of traditional data structures. They also proposed a learned approach [10], [11] to improve sorting performance.

Their method, known as SageDB Sort, utilizes a learned model to generate a roughly ordered state of elements by predicting (mapping) the positions of elements, and then refined it by a traditional sorting algorithm like Quick Sort. However, this approach has limitations. Conflicts often arise when converting the learned model's output, such as multiple elements being mapped to the same position, leading to performance bottlenecks (see Section IV). Resolving these conflicts (especially when numerous) can be time-consuming, making it less efficient than traditional algorithms.

The question of how to design an effective deep learning-based sorting algorithm remains unanswered. Specifically, key issues include determining which type of neural network performs best for sorting, understanding the complexity of neural network-based sorting, dealing with conflicts, and minimizing operations such as data comparison and movement during the sorting process.

To address these issues, this work presents NN-sort, a neural network-based sorting algorithm designed to move beyond traditional sorting paradigms. Instead of relying solely on comparisons or inherent data characteristics, NN-sort harnesses the power of neural networks to create a data distribution-aware sorting method. By training a model on historical data to predict the sorted positions of new data, NN-sort offers a novel approach that achieves efficient and scalable sorting while incorporating an effective conflict-handling mechanism.

The architecture of NN-sort consists of three distinct phases: the input phase, where data is transformed into neural network-compatible vectors; the sorting phase, where a learned neural network iteratively refines the data order; and the polish phase, where traditional methods finalize the sorting (*i.e.,* ensuring the output is correct). This layered approach enables NN-sort to handle large datasets efficiently, minimizing the computational overhead from conflicts—a primary performance bottleneck in SageDB Sort. Moreover, we systematically explored the potential of NN-sort, discussed its complexity, performance, and advantages over traditional algorithms. Through rigorous experiments on both synthetic and real-world datasets, we justify the effective of NN-sort.

The contributions of this paper are summarized as follows: (a) we investigate the opportunities and challenges of enhancing traditional sorting processes by leveraging neural network-based learning approaches; (b) we develop NN-sort, a novel neural network-based sorting method that utilizes historical data to train a data distribution-aware model. This trained model performs high-performance sorting on incoming data iteratively, with an additional touch-up process to ensure the correctness of the final result. In contrast to state-of-the-art learned sorting methods, *e.g.,*SageDB Sort, NN-sort scales effectively by reducing conflicts during CDF mapping; (c) we provide a formal analysis of NN-sort's complexity using a cost model that clarifies the intrinsic relationship between model accuracy and sorting performance. (d) we evaluate NN-sort's performance on both synthetic and real-world datasets. Experimental results demonstrate that NN-sort achieves up to an order of magnitude speed-up in sorting time compared to

[†]work done while author was with Yunnan University
*corresponding author

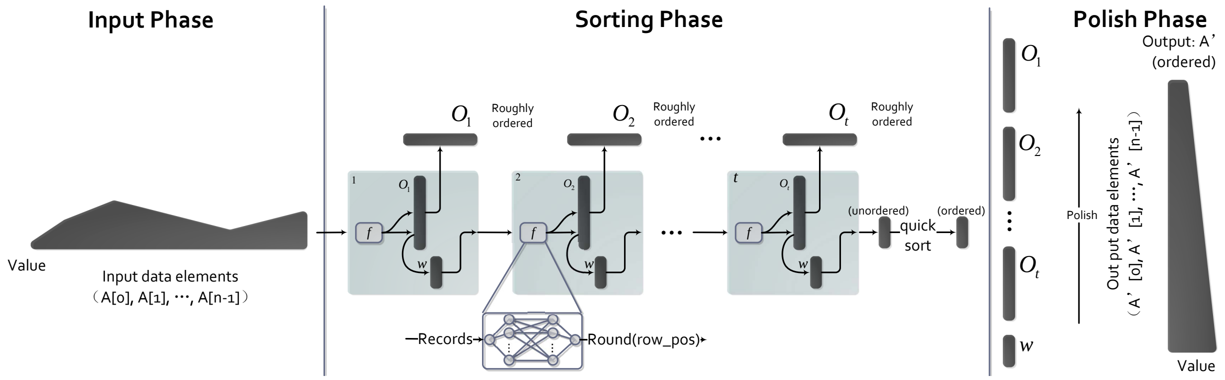

Fig. 1: NN-sort architecture

state-of-the-art sorting algorithms.

The rest of the paper is organized as follows: the design of NN-sort is presented in Section II. The complexity of NN-sort is discussed in Section III. The experimental results are presented and analyzed in Section IV. Related work is reviewed in Section V, and the paper is concluded in Section VI.

II. Neural Network Based Sort

In this section, we discuss the design of NN-sort, including how to use a neural network model for effective sorting, as well as how such a neural network model can be trained.

Challenges. Sorting involves mapping elements from an unsorted state to a sorted state. Rather than relying on traditional comparison-based methods (*e.g.,* Quick Sort), this mapping can be achieved through a data distribution-aware model that takes each element as input and returns its expected position within the sorted array. Given an ideal function f and distinct elements $x, y \in A$ such that $x < y$, the function would ideally satisfy $f(x) < f(y)$. While such a ”magic” function may not always hold perfectly, it can still accelerate sorting, with greater accuracy leading to more effective acceleration. Before discussing the methods, several challenges must be addressed to ensure both effectiveness and accuracy. (1) first, for correctness, the model must precisely reflect the order among input data elements, producing results consistent with those of traditional sorting algorithms. However, it is impossible to have that function f, especially if we train the model just based on samples from the input data; (2) second, for effectiveness, the model ideally should sort large volumes of data in a single pass. Achieving this requires a model to be both complex and accurate enough to capture the exact order of all elements, which can result in significant training cost and inference cost. Thus, a balance between model accuracy and sorting performance is crucial. (3) third, conflicts arise when multiple input elements are mapped to the same output position, *i.e.,* $f(a) = f(b)$ where $a, b \in A$ and $a \neq b$. Effectively managing these conflicts is crucial for both the correctness and efficiency of the learned sorting approach. SageDB Sort addresses these collisions using traditional sorting algorithms, such as Quick Sort, which incurs computational overhead when collisions is too large. That is to said, there is still room for improvement.

```
Procedure NN-sort
Input: $A$: the array of data elements
$f$: the learned model
$m$: the relaxation factor
$\epsilon$ the predefined iteration limit
$\tau$ the predefined threshold
Output: $A'$: the array of data elements after sorted
$w \leftarrow$ translate($A$)
init $O \leftarrow \emptyset$, $i \leftarrow 0$
while $0 < i < \epsilon$ and count($w$) > $\tau$ do
    init $o_i \leftarrow \emptyset$, $c \leftarrow \emptyset$
    $w\_pos \leftarrow w.map(f)$
    for $idx$ in count($w$) do
        $pos \leftarrow$ round($w\_pos[idx] * m$)
        if $o_i$[$pos$] is empty do
            $o_i[pos] \leftarrow w[idx]$
        else
            $c$.append($w[idx]$)
    $O$.append($o_i$)
    $w \leftarrow c$
    ++i
QuickSort($w$)
$A' \leftarrow$ polish($O, w$)
Return $A'$
```

Fig. 2: Algorithm NN-sort

NN-sort Design. In response to these challenges, we designed an iterative neural network-based sorting method. Unlike SageDB Sort, which employs a complex model to sort data in a single round, our approach utilizes a simpler model to sort over multiple rounds. Each round generates a roughly sorted array, with conflicts carried forward to the next iteration. This process continues until the conflicts in that iteration fall below a predefined threshold or a number of iterations is reached. The final small conflict array is then sorted using a traditional method like Quick Sort and merged with the roughly sorted arrays. After a final traversal to ensure correctness, a fully

```
Procedure polish($O$)
Input: $O = \{o_1, o_2, ...\}$: array of roughly sorted arrays.
$w$: strictly sorted array.
Output: $A'$: the array of data elements after sorted
1. $A' \leftarrow w$
2. for $o_i \in O$ do
3.     $A' \leftarrow$ InsertSort($A'$, $o_i$)
4. Return $A'$
```

Fig. 3: Algorithm polish

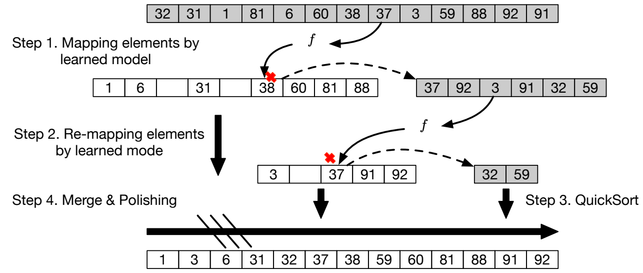

Fig. 4: Example

sorted result is obtained. The benefits are two folds: (1) using a simpler model reduces both inference and training costs; (2) the learned model can be applied repeatedly, avoiding direct sorting of conflicting elements with traditional methods.

Figure 1 illustrates this approach, where the input array A is sorted into A'. The process is divided into three phases: *Input*, *Sorting*, and *Polish*. Figure 2 details the workflow of NN-sort, with Line 1 addressing the input phase, Lines 2-15 covering the sorting phase, and Line 16 corresponding to the polish phase.

Input Phase. The input phase prepares the data for the neural network by encoding it appropriately, ensuring compatibility for processing. For example, string-type data is converted into ASCII values. This encoding step is crucial, as it standardizes the data format and enables the neural network to interpret and process a wide variety of input types, such as integers, floating-point numbers, or categorical data, in a structured and efficient manner. For simplicity, we denoted such operations as $w \leftarrow \mathsf{translate}(A)$ (line 1, Figure 2).

Sorting Phase. In the sorting phase, a learned model f iteratively organizes unordered data into approximately sorted arrays. First, the learned model f maps each element to its expected position (line 5, Figure 2). Then, o_i stores elements of w based on their value in w_pos, where i denotes the iteration number. If a collision occurs in o_i—where multiple elements map to the same position—only the first element is stored in o_i, while subsequent elements are placed in a temporary conflict array c (line 8-11, Figure 2). In the following iteration, elements in c are reprocessed by learned model f (line 13, Figure 3). This process continues until either a predefined maximum number of iterations ϵ is reached or the size of c drops below a threshold τ, at which point it is sorted using a traditional algorithm.

It is worth mentioning that, each element in w_pos represents the expected position of corresponding elements of w within the final sorted array. Instead of using $w\_pos[idx]$, which is the direct output of f, we use $round(w\_pos[idx]{*}m)$, a rounded value, to represent the position of $w[idx]$. The reasons are two folds: (1) the outputs of f are decimals while the positions need to be integers. (2) with relaxation factor m the input data elements can be mapped into a larger space, thereby reducing the number of conflicts. In addition, all conflicting data elements are stored in a conflict array c and used as input to f for the next iteration. If the model f does not perform effectively, *i.e.,* the conflicting array may never shrink below τ or may decrease too slowly, potentially resulting in higher overhead than traditional sorting algorithms. To prevent this, a threshold ϵ limits the maximum number of iterations. As we will show in the experimental section, $\epsilon = 2$ or $\epsilon = 3$ are good enough for accelerating sorting. There is a clearly decreased edge effect on the number of iterations.

Polish Phase. The polish phase refines the roughly sorted arrays $O = \{o_1, o_2, ...\}$ to ensure the correctness of the output. Figure 3 outlines this process, where the arrays in O are polished and merged with the strictly ordered array A'. The algorithm iterates over each array in O, merging them with A' one by one. Elements in o_i are either appended or inserted into the result, depending on their order relative to A' (*i.e.,* Insert Sort).

Remark. Since o_i is only roughly ordered, out-of-order elements are inserted into their correct positions in A', ensuring NN-sort's reliability despite potential errors from the learned model. Though insertion is costlier than appending, it is limited to out-of-order elements. As model accuracy improves, the polish phase incurs acceptable overhead. Section III discusses NN-sort's complexity, with experimental results showing few out-of-order elements, yielding nearly linear performance.

Example 1: Figure 4 illustrates how NN-sort sorting.

Given thresholds $\tau = 2$, $\epsilon = 2$ and an unordered array $A = \{32, 60, 31, 1, 81, 6, 88, 38, 3, 59, 37, 92, 91\}$, NN-sort first checks if A's size is below τ; if so, a traditional sorting method is applied. Otherwise, learned sorting begins.

Here, NN-sort processes A in two rounds with a learned model. A conflict arises between elements 37 and 38, as f maps them to the same position, placing 37 in a conflict array c. At the end of the first iteration, elements 92, 3, 91, 32, and 59 are also in the conflict array.

After the first iteration, since w's size exceed τ and the iteration count is below ϵ, all elements in $c = \{37, 92, 3, 91, 32, 59\}$ are reprocessed by f in a second iteration, yielding a new sorted array $o_2 = \{3, 37, 91, 92\}$ and a smaller conflict array $c = \{32, 59\}$. Then a traditional sorting algorithm (*e.g.,* Quick Sort) is applied to c, and finally, o_1, o_2, and sorted c are merged in the polish phase, producing a fully ordered result.

$\square$

TABLE I: Notations

symbols	notations
n	the amount of data elements to be sorted
σ_i	collision rate per iteration
e_i	the number of data elements that were out-of-order in the i-th iteration
ϵ	the predefined limit of iterations
t	the number of completed iterations
θ	The operations required for data to pass through f

Training. While training time is not the focus, all our models—whether shallow neural networks or simple linear/multivariate regression models—train quickly and perform well, as perfect position mapping (i.e., no conflicts or out-of-order elements) is unnecessary. The training and test data can differ; any learned order relationships help the model understand the sorting task.

III. Model Analysis

This section establishes the time complexity of NN-sort by analyzing key operations—moving, mapping, and comparing data elements. A cost model is introduced to highlight relationships among factors like conflict rate, model scale, iteration count, out-of-order rate, data volume, and required operations.

The total operations of NN-sort is expressed as a $T(n, e, \sigma, t, \theta)$, where: n is the number of data elements to be sorted, $e = \{e_1, ..., e_t\}$ is the set of probabilities, with e_i denoting the proportion of out-of-order elements in the i-th iteration, $\sigma = \{\sigma_1, ..., \sigma_t\}$ is the set of conflict rates, where σ_i represents the conflict rate in the i-th iteration, t is the number of iterations completed, θ denotes the number of operations required for each data element to pass through the neural network. These basic notations are summarized in Table I.

As shown in Eq 1, the number of operations for NN-sort to sort n ($n > 1$) data elements is $C_1 n^2 + C_2 nlogn + C_3 n$.

$$T(n, e, \sigma, t, \theta) = \begin{cases} 1, & if\ n = 1 \\ C_1 n^2 + C_2 nlogn + C_3 n, & if\ n > 1 \end{cases} \tag{1}$$

$$C_1 = [\frac{1}{2}\sum_{i=1}^{t} e_i(1-\sigma_i)(\prod_{j=1}^{i-1}\sigma_j)^2]$$

$$C_2 = \prod_{j=1}^{t}\sigma_j$$

$$C_3 = \sum_{i=1}^{t}[\theta\sum_{j=1}^{i}\sigma_j + (1-e_i)(1-\alpha_i)\prod_{j=1}^{i-1}\sigma_j + \prod_{j=1}^{i}\sigma_j] + (\prod_{j=1}^{t}\sigma_j)log(\prod_{j=1}^{t}\sigma_j)$$

In NN-sort, the majority of the cost is spent in the Sorting and Polish phases. Let $s(n)$ represent the time spent in the Sorting phase and $p(n)$ represent the time spent in the Polish phase, we now formally analyze the complexity of NN-sort.

$s(n)$ consists of two kinds of operations: iteratively feeding the data elements into a learned model f and sorting the array w at the last iterations using traditional sorting algorithms (*e.g.,* QuickSort), the time complexity of which is $nlogn$. If $n > 1$, then $\theta\sum_{j=1}^{i}\sigma_j n$ operations are required to feed data into model f in the i-th iteration. An additional $(\prod_{j=1}^{t}\sigma_j)nlog(\prod_{j=1}^{t}\sigma_j)n$ operations are required to keep w order, since the size of conflicting array w updated in the last iteration is $(\prod_{j=1}^{t}\sigma_j)n$. Therefore, at the end of the algorithm, the total operations of $s(n)$ is $(\prod_{j=1}^{t}\sigma_j)nlog(\prod_{j=1}^{t}\sigma_j)n + \theta\sum_{i=1}^{t}\sum_{j=1}^{i}\sigma_j n$.

$$\begin{aligned}
T(n) &= s(n) + p(n)\ ,\ (n > 1) \\
&=(\prod_{j=1}^{t}\sigma_j)nlog(\prod_{j=1}^{t}\sigma_j)n + \theta\sum_{i=1}^{t}\sum_{j=1}^{i}\sigma_j n + p(n) \\
&=(\prod_{j=1}^{t}\sigma_j)nlog(\prod_{j=1}^{t}\sigma_j)n + \theta\sum_{i=1}^{t}\sum_{j=1}^{i}\sigma_j n \\
&\quad + \sum_{i=1}^{t}[e_i(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n \times \frac{\prod_{j=1}^{i-1}\sigma_j n}{2} \\
&\quad + (1-e_i)(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n + \prod_{j=1}^{i}\sigma_j n] \\
&=[\frac{1}{2}\sum_{i=1}^{t} e_i(1-\sigma_i)(\prod_{j=1}^{i-1}\sigma_j)^2]n^2 + \prod_{j=1}^{t}\sigma_j nlogn \\
&\quad + \{\sum_{i=1}^{t}[\theta\sum_{n=1}^{i}\sigma_j + (1-e_i)(1-\alpha_i)\prod_{j=1}^{i-1}\sigma_j \\
&\quad + \prod_{j=1}^{i}\sigma_j] + (\prod_{j=1}^{t}\sigma_j)log(\prod_{j=1}^{t}\sigma_j)\}n
\end{aligned}$$

$p(n)$ involves two tasks: correcting any out-of-order elements and merging the intermediate arrays (*i.e.,* $o_1, ..., o_t$ and w). If no elements are out of order in o_i, NN-sort only needs to traverse the data once to merge them. However, in practice, out-of-order elements are almost inevitable, as the model f is unlikely to be 100% accurate.

For the ordered elements in o_i, NN-sort only requires appending it, with a time complexity of time complexity of $O(1)$. Therefore, in the i-th iteration, at most $\prod_{j=1}^{i-1}\sigma_j n$ operations are required to complete the insertion, and at least one operation is needed to insert out-of-order elements. While, for an out-of-order element in the i-th merge iteration, $\frac{\prod_{j=1}^{i-1}\sigma_j n}{2}$ operations are required to insert it into the final ordered result. Theoretically, assume that there are $e_i(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n$ out-of-order elements in the i-th iteration. It takes a total of $e_i(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n \times \frac{\prod_{j=1}^{i-1}\sigma_j n}{2}$ operations to process these elements. Correspondingly, $(1-e_i)(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n$ elements in o_i and $\prod_{j=1}^{i}\sigma_j n$ elements in w remain ordered. Thus in the i-th merge iteration, a total of $\prod_{j=1}^{i}\sigma_j n + (1-e_i)(1-$

$\sigma_i)\prod_{j=1}^{i-1}\sigma_j n$ operations are required to append the ordered elements to the final result. Overall, NN-sort requires a total of $\sum_{i=1}^{t}[e_i(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n \times \frac{\prod_{j=1}^{i-1}\sigma_j n}{2} + (1-e_i)(1-\sigma_i)\prod_{j=1}^{i-1}\sigma_j n + \prod_{j=1}^{i}\sigma_j n]$ to sort n data elements (We show the detail in Equation 1).

IV. Experimental Study

Using real and synthetic data, we conducted five experiments to evaluate (1) overall sorting performance, (2) iteration impact, and (3) effects of changing data distribution.

A. Experimental setup

Datasets. The datasets used in this section are generated from the most commonly observed distributions in the real world, such as uniform distribution, normal distribution, and lognormal distribution. The models used in the experiments are trained over a subset of the testing data. The sizes of the testing dataset vary from 200MB to 500MB and each data element is 64 bits wide floating number. To verify the performance of the NN-sort under the real-world dataset. We use the QuickDraw game dataset from Google Creative Lab [12], which consists of $50,426,265$ records of schema $\{$'key-id', 'word', 'country code', 'timestamp', 'recognized', 'drawing'$\}$. Sorting is perform on 'key-id'.

Baselines. We compared with five baselines (1) **Quick Sort** [13]: This algorithm divides the input dataset into two independent partitions, such that all the data elements in the first partition are smaller than those in the second partition. Then, the dataset in each partition is sorted recursively. The time complexity of **Quick Sort** can achieve $\mathcal{O}(nlogn)$ in the best case while $\mathcal{O}(n^2)$ in the worst case. (2) **std::sort** [14]: std::sort is one of the most widely used sorting algorithms from c++ standard library, and its time complexity is $\mathcal{O}(nlogn)$ (3) **std::heap sort** [14]: **std::heap sort** is another sorting algorithm from c++ standard library, and it guarantees to perform at $\mathcal{O}(nlogn)$ time complexity. (4) **Redis Sort** [15]: Redis Sort is a sorting method based on a data structure named $sortSet$. To sort M data elements in a $sortSet$ of size N, the efficiency of **Redis Sort** is $\mathcal{O}(N + Mlog(M))$. In addition, we also compared NN-sort with (5) **SageDB Sort** [10], [11], leading performance DNN-based sorting method.The relaxation factor m is set to 1.25 for learned sorting methods to reduce conflicts.

Measurements. We used sorting time and sorting rate of Equation 2 to evaluate the overall performance.

$$\text{sorting rate} = \frac{\text{elements}}{\text{time to finish sorting}} \tag{2}$$

We also used traditional sorting rate to evaluate learned-based sorting methods which is described in Equation 3. This rate indicates how many data elements still require traditional sorting due to model inaccuracy in the learning-based approach. Ideally, a lower traditional sorting rate signifies the better performance of learning-based sorting.

TABLE II: Evaluation under real-world data

Algorithm name	Time (sec.)	Sorting Rate (elements/sec.)	The traditional sorting rate (%)
Quick Sort	10.86	4666.14	-
std::heap sort	13.46	3746.44	-
std::sort	23.71	2127.19	-
Redis Sort	63.14	798.6320	-
SageDB Sort	10.53	4790.125	9.16
NN-sort	8.47	5950.186	0.4

$$\text{Traditional sorting rate} = \frac{\text{size(last conflicting array } w\text{)}}{\text{size of the original array A}} \tag{3}$$

Environment. Experiments were conducted on a machine with 64GB RAM, a 2.6GHz Intel i7 processor, and a GTX1080Ti GPU with 16GB memory, running RedHat Enterprise Server 6.3. Each result reported is the median of ten runs.

Training details. We employed a regression model with three hidden layers, containing 2, 6, and 1 neurons, respectively. A rounding function is used to determine each element's final position. Adam [16] was the chosen optimizer. The training was conducted using a GPU and is performed offline, so training time is excluded from the runtime.

$$loss_\delta = \begin{cases} \frac{1}{2}(f(x_i) - label_i)^2, & if\ |f(x_i) - label_i| \leq \delta, \\ \delta|f(x_i) - label_i| - \frac{1}{2}\delta^2, & otherwise \end{cases} \tag{4}$$

To avoid the impact of outliers during training, the model used in experiments is trained according to the Huber loss [17] as shown in Equation 4. The batch size for training is set to 128. For all environments, we use the Adam optimizer with a learning rate of 0.001.

B. Experimental results.

Exp-1: Overall Sorting Performance. Figure 5 presents a performance comparison of NN-sort against traditional sorting algorithms across various datasets with increasing sizes. Figures 5(a)–(c) show the total sorting time, while Figures 5(d)–(f) illustrate the sorting rates. Figures 5(g)–(i) highlight the traditional sorting rate comparison between NN-sort and SageDB sort, as defined in Equation 3. we observe the following:

NN-sort exhibits notable advantages over traditional sorting algorithms. As shown in Figure 5 (d), its sort rate for a lognormal distribution dataset reaches nearly 8,300 elements per second, outperforming std::heap sort by 2.8×, Redis Sort by 10.9×, std::sort by 4.78×, and Quick Sort by 218%. It also exceeds SageDB Sort by 15%. The dataset's value range—defined by its maximum and minimum values—affects NN-sort 's performance. As shown in Figure 5 (h), a slight decline in sorting rate occurs with highly concentrated values, which create more conflicts and reduce efficiency. In contrast, fewer records within the same range enhance sorting perfor-

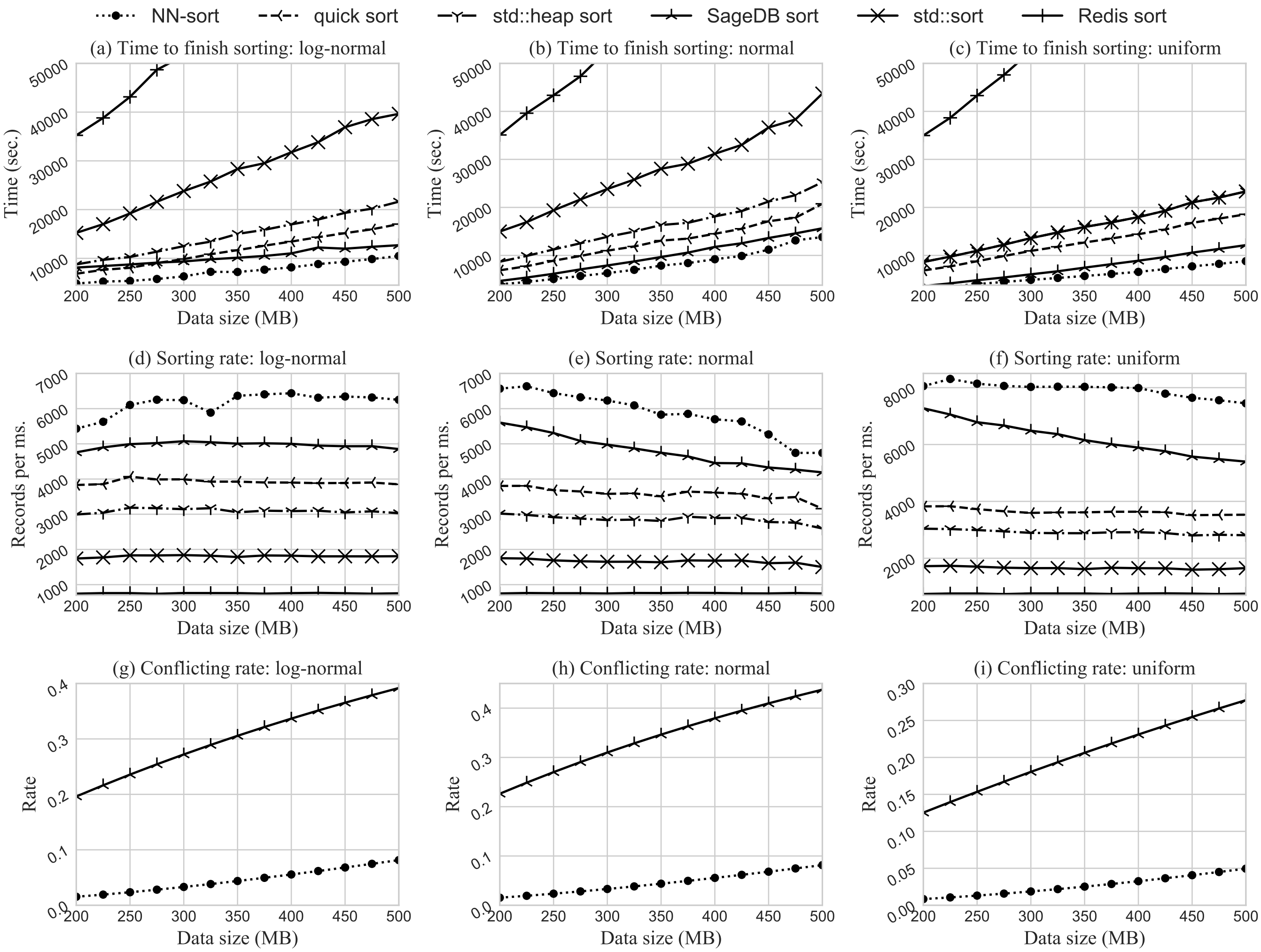

Fig. 5: Overall performance evaluation

mance.

NN-sort achieves optimal performance with uniformly distributed data, reaching a sorting rate of approximately 8,000 records per second—about 1.3× higher than with a normal distribution—due to fewer conflicts in uniformly distributed records.

Compared to SageDB Sort, NN-sort consistently reduces reliance on traditional sorting. A larger proportion of elements are accurately sorted by NN-sort's neural model, minimizing the need for the more time-consuming traditional sorting and contributing to NN-sort 's superior performance over SageDB Sort.

Exp-2: Evaluation on real-word Dataset. Using the model trained in previous sections on uniformly distributed data, we evaluated NN-sort 's performance on the real-world dataset QuickDraw. As shown in Table II, NN-sort delivers significant performance gains over traditional sorting algorithms on real-world data. With a sorting rate of 5,950 elements per second, NN-sort outperforms std::sort by 2.72× and Redis Sort by 7.34×, and is also 58% faster than std::heap sort. Additionally, NN-sort surpasses SageDB Sort in both traditional sorting rate and overall sorting rate.

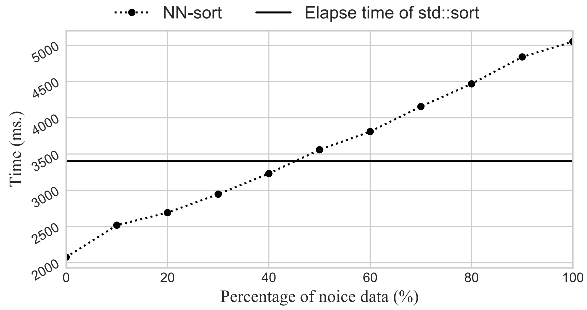

Fig. 6: The impact of data distribution on NN-sort performance

Exp-3: Impact of the Changing Data Distribution. As

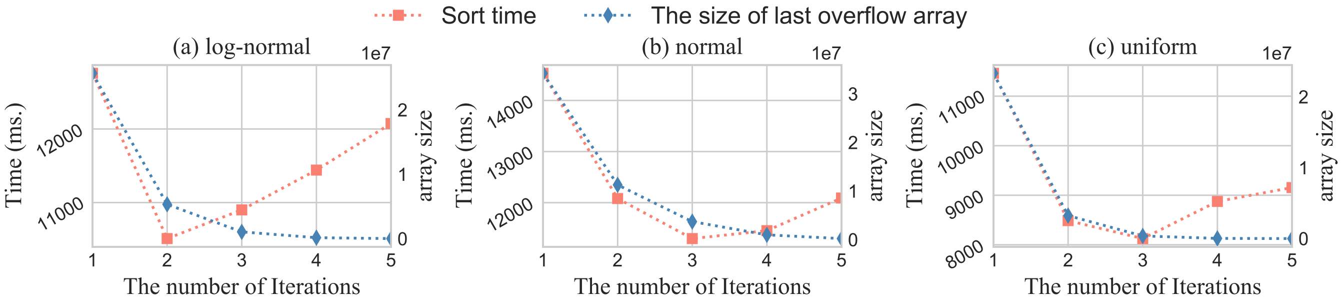

Fig. 7: Impact of Iterations

shown in previous experiments, NN-sort performs optimally when the distribution of the sorting data resembles that of the training data. But how does it perform when faced with a different data distribution? To explore this, we trained a model using a 100MB uniformly distributed dataset, then applied it to sort datasets with varying distributions. Specifically, the test dataset combined uniformly and normally distributed data, with the latter considered ”noisy.” Sorting time was measured to assess NN-sort 's effectiveness, as shown in Figure 6. Results indicate that as noise in the dataset increases, NN-sort's effectiveness declines due to a growing number of conflicts elements generated in each iteration when the test data distribution diverges from the training data. These elements must then be handled by traditional algorithms during the polish phase. Nonetheless, NN-sort shows resilience to distributional changes, outperforming the widely-used std::sort algorithm even with up to 45

Exp-4: Impact of Iterations. NN-sort 's sorting performance is influenced by both the size of the final conflicting array and the number of iterations. Increasing the number of iterations reduces the size of the remaining conflicting array that requires traditional sorting, yet also extends model processing time. Conversely, fewer iterations leave a larger conflicting array, increasing the time required for traditional sorting. In this set of experiments, we quantify these factors to guide users in optimizing NN-sort for improved performance.

In Figure 7, the rhombus-dotted line represents the size of the final conflicting array, while the round-dotted line indicates total sorting time. Results show that while additional iterations reduce the size of the final conflicting array, they don't necessarily improve performance, as each iteration requires the model to process all input data elements. Our experiments suggest that 2-3 iterations provide an optimal balance between conflicting array size and sorting time.

Exp-5: Evaluation of sorting accuracy. A more complex neural network generally enhances model expressibility, resulting in lower conflict rates, and fewer out-of-order elements, but higher inference times. Thus, understanding the impact of these factors on NN-sort's overall time complexity is essential. In this section, we use a cost curve to illustrate how model quality—represented by the conflict rate σ_i and out-of-order rate e_i in each sorting iteration—affects NN-sort 's performance.

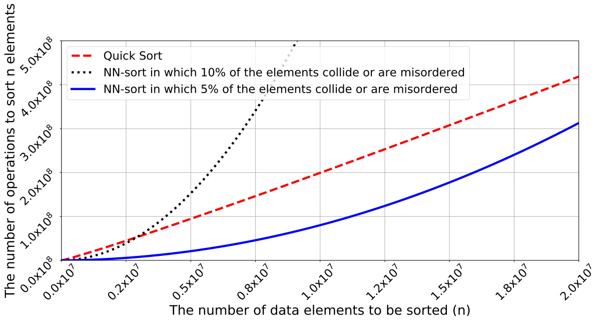

Fig. 8: Comparison of operations between traditional sorting algorithm and NN-sort with different model qualities.

Figure 8 compares the number of operations required by NN-sort to sort n elements against Quick Sort's baseline complexity ($\mathcal{O}(n \log n)$). To illustrate performance variations, we adjust NN-sort 's model quality. This analysis assumes NN-sort performs up to five iterations ($t = 5$), with a model scale θ of 32 neurons, and equal conflict and out-of-order rates ($\sigma_i = e_i$) that remain constant across iterations. Results show that NN-sort substantially outperforms Quick Sort when conflict and out-of-order rates are at 10%, with even greater performance gains as these rates drop to 5

In summary, fewer conflicts and misordered elements result in more efficient sorting with NN-sort. A well-trained model with a misorder rate of 10% or lower can outperform traditional sorting algorithms in terms of computational efficiency.

V. Related Work

Sorting is one of the most fundamental algorithms in computing. We identify two key research areas: methods to reduce sorting time complexity and neural network-based data structures.

Methods for reducing the sorting time complexity. Many researchers have focused on accelerating sorting by reducing its time complexity. Traditional comparison-based sorting algorithms like Quick Sort, Merge Sort, and Heap Sort require at least $logn! \approx nlogn - 1.44n$ operations to sort n elements [18]. Among these, Quick Sort achieves $\mathcal{O}(nlogn)$ average complexity but degrades to $\mathcal{O}(n^2)$ in the worst case. Merge Sort, while guaranteeing a worst-case of $nlogn-0.91n$, requires additional linear space relative to the number of elements [18]. To mitigate the drawbacks of these algorithms and further reduce sorting time, researchers have combined different sorting techniques to leverage their strengths. Tim Sort [19], the default sorting algorithm in Java and Python, combines Merge Sort and Insertion Sort [13] to achieve fewer than $nlogn$ comparisons on partially sorted arrays. Stefan Edelkamp et al. proposed Quickx Sort [20], which uses at most $nlogn - 0.8358n + \mathcal{O}(logn)$ operations for in-place sorting. They also introduced a median-of-medians variant of Quick Merge Sort [18], which employs the median-of-medians algorithm for pivot selection, reducing the operation count to $nlogn + 1.59n + \mathcal{O}(n^{0.8})$.

Redis Sort [15] is a build-in sorting method of the Redis database based on the sortSet data structure. It sorts M elements in a sortSet of size N with an efficiency of $\mathcal{O}(N + Mlog(M))$.

Unlike previous work, this approach uses a learned model complexity to map an unordered array to a roughly ordered state, reducing overall operations. In the worst case, NN-sort has complexity $\mathcal{O}(n^2)$ if all elements map to the same position, though practical operations remain lower than traditional sorting, This is validated by our experiment in Figure 8.

Learned data structures and algorithms. This thread of research is to explore the potential of using the neural network-based learned data structures to improve the performance of systems. Tim Kraska [9], [10] discussed the benefits of learned data structures and suggested that R-tree can be optimized by learned data structures. Xiang et al. [8] proposed an LSTM-based inverted index structure. By learning the empirical distribution function, their learned inverted index structure led to fewer average look-ups when compared with traditional inverted index structures. Alex Galakatos et al. [21] presented a data-aware index structure called FITing-Tree, which can approximate an index using piece-wise linear functions with a bounded error specified at construction time. Michael Mitzenmacher [22] proposed a learned sandwiching bloom filter structure, while the learned model is sensitive to data distributions.

Unlike the research mentioned above, our approach integrates sorting with learning by training a model to enhance sorting performance. Additionally, we employ an iteration-based mechanism to further optimize performance by minimizing conflicts. We also provide a formal analysis of the time complexity of our approach and present a cost model to balance model accuracy with sorting performance. A closely related work is SageDB Sort [11], [21], which leverages deep neural networks for sorting. Our approach improves upon SageDB Sort by offering a more efficient solution for handling position conflicts generated by the learned model.

VI. Conclusion

Sorting is fundamental in big data processing. We introduce NN-sort, a neural network-based sorting method that uses historical data to sort new data, iteratively reducing sorting conflicts—a key bottleneck in learned sorting. Our analysis includes complexity, a cost model, and the balance between model accuracy and performance. Experiments show NN-sort outperforms traditional algorithms. Future work includes enhancing NN-sort's adaptability to changing data distributions.

References

[1] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, p. 10, 2006.

[2] R. Hilker, C. Sickinger, C. N. Pedersen, and J. Stoye, "UniMoG—a unifying framework for genomic distance calculation and sorting based on DCJ," *Bioinformatics*, vol. 28, no. 19, pp. 2509–2511, 2012.

[3] D. Cederman and P. Tsigas, "A practical quicksort algorithm for graphics processors," in *Algorithms - ESA 2008*, D. Halperin and K. Mehlhorn, Eds., 2008, pp. 246–258.

[4] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, "Sorting in linear time?" *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 74–93, 1998.

[5] S. Bandyopadhyay and S. Sahni, "GRS - GPU radix sort for multifield records," in *HiPC*, 2010, pp. 1–10.

[6] J. Tang and X. Zhou, "Cardinality sorting and its bit-based operation-based optimization (in chinese)," *JOURNAL OF NANJINGUNIVERSITY OF TECHNOLOGY*, vol. 20, 2006.

[7] X. Zhu, Q. Zhang, T. Cheng, L. Liu, W. Zhou, and J. He, "Dlb: deep learning based load balancing," in *CLOUD*, 2021.

[8] W. Xiang, H. Zhang, R. Cui, X. Chu, K. Li, and W. Zhou, "Pavo: A rnn-based learned inverted index, supervised or unsupervised?" *IEEE Access*, vol. 7, pp. 293–303, 2019.

[9] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.

[10] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," in *CIDR*, 2019.

[11] J. Ding, R. Marcus, A. Kipf, V. Nathan, A. Nrusimha, K. Vaidya, A. van Renen, and T. Kraska, "Sagedb: An instance-optimized data analytics system," *PVLDB*, vol. 15, no. 13, 2022.

[12] Google, "Google creative lab," Available: https://github.com/googlecreativelab, google Creative Lab [Online].

[13] T. H. Cormen, *Introduction to Algorithms, 3rd Edition*. Press.

[14] "C++ resources network," http://www.cplusplus.com/, general information about the C++ programming language, including non-technical documents and descriptions.

[15] "Redis," https://redis.io/, redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR* , 2015.

[17] P. J. Huber, "Robust estimation of a location parameter," *Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964.

[18] S. Edelkamp and A. Weiß, "Worst-case efficient sorting with quickmergesort," in *ALENEX*, 2019, pp. 1–14.

[19] "Python resources network," https://www.python.org/, general information about the Python programming language, including non-technical documents and descriptions.

[20] S. Edelkamp and A. Weiß, "Quickxsort: Efficient sorting with n logn - 1.399n + o(n) comparisons on average," in *International Computer Science Symposium in Russia*, 2014, pp. 139–152.

[21] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *SIGMOD*, 2019.

[22] M. Mitzenmacher, "A model for learned bloom filters, and optimizing by sandwiching," *CoRR*, vol. abs/1901.00902, 2019. [Online]. Available: http://arxiv.org/abs/1901.00902