Neural Computing and Applications (XXXX) 35:21399–21414
https://doi.org/10.1007/s00521-023-08841-1

S.I.: TECHNOLOGIES OF THE 4TH INDUSTRIAL REVOLUTION WITH APPLICATIONS

Neural networks as building blocks for the design of efficient learned indexes

Domenico Amato[1] · Giosué Lo Bosco[1] · Raffaele Giancarlo[1]

Received: 31 January 2023 / Accepted: 28 June 2023 / Published online: 21 July 2023

Abstract
The new area of *Learned Data Structures* consists of mixing Machine Learning techniques with those specific to Data Structures, with the purpose to achieve time/space gains in the performance of those latter. The perceived paradigm shift in computer architectures, that would favor the employment of graphics/tensor units over traditional central processing units, is one of the driving forces behind this new area. The advent of the corresponding branch-free programming paradigm would then favor the adoption of Neural Networks as the fundamental units of Classic Data Structures. This is the case of Learned Bloom Filters. The equally important field of Learned Indexes does not appear to make use of Neural Networks at all. In this paper, we offer a comparative experimental investigation regarding the potential uses of Neural Networks as a fundamental building block of Learned Indexes. Our results provide a solid and much-needed evaluation of the role Neural Networks can play in Learned Indexing. Based on our findings, we highlight the need for the creation of highly specialised Neural Networks customised to Learned Indexes. Because of the methodological significance of our findings and application of Learned Indexes in strategic domains, such as Computer Networks and Databases, care has been taken to make the presentation of our results accessible to the general audience of scientists and engineers working in Neural Networks and with no background about Learned Indexing.

Keywords Information retrieval · Machine learning · Neural networks · Learned indexes

1 Introduction

Learned Data Structures are a new area of research that combines Machine Learning (ML) techniques with those inherent to Data Structures, with the goal of enhancing the speed and space efficiency of traditional Data Structures. It was started by Kraska et al. [27], has expanded quickly [16], and has recently been extended to include Learned Algorithms [35]. Its impact on the design of modern Information Systems is expected to be quite significant [28].

1.1 The motivation for learned data structures: computer architectures

Due to their learning ability, Neural Networks (NNs) [38] are without a doubt the ML models perceived to have the greatest potential in the field of Learned Data Structures [27]. However, they demand excessive computational power. This computational bottleneck has essentially been eliminated for their successful use in many application areas [29], thanks to the adoption of highly engineered development platforms such as TensorFlow [1], and the introduction of graphics processing units (GPU) and tensor

An extended abstract related to this paper has been presented at the 23rd International Conference EAAAI/EANN. The Conference proceedings are the Series Communications in Computer and Information Science by Springer 2022, Vol. 1600. The full reference is [5].

✉ Giosué Lo Bosco
giosue.lobosco@unipa.it

Domenico Amato
domenico.amato01@unipa.it

Raffaele Giancarlo
raffaele.giancarlo@unipa.it

[1] Dipartimento di Matematica e Informatica, Universitá degli Studi di Palermo, Via Archirafi 34, 90123 Palermo, Italy

processing units (TPU) architectures in commercial computers [40, 43]. The main advantage of these novel architectures is their excellent ability to parallelize algebraic tasks performed by NNs. Further gains are foreseen since current studies argue that the power of the GPU can increase by 1000x in the next few years. On the other hand, those performance gains are not anticipated for traditional Central Processing Units (CPUs), due to Moore's Law limitations [36]. Therefore, a programming paradigm that would promote straight-line mathematical operations, which can be easily pipelined on GPUs, appears much more promising than the one that uses branches of the if-then-else type. Due to these factors and to the fact that classic Data Structures implementations use branch instructions, the use of ML models, like NNs, to reduce the branchy part of code in Data Structures implementations may lead to versions of those latter that can take full advantage of the new cutting-edge architectures. Such an achievement would have major effects on many strategic domains, e.g., Computer Networks and Databases. Unfortunately, even though the reason for creating Learned Data Structures based on NNs is strong, how to achieve such an advantage and a quantification of how much would that be are at an early stage of an investigation, or not addressed at all, as we discuss in more detail in the following.

1.2 The role of NNs: from motivation to design and implementation of learned data structures

- **Learned Bloom Filters.** Since the very beginning of the field of Learned Data Structure, NNs have been actively employed in the design and implementation of Learned Bloom Filters [27]. Note that a Bloom Filter[9] is a Data Structure to solve the so-called *Approximate Membership Problem* for a given set $A \subset U$, where U is a universe of elements. This means deciding whether a query element $x \in U$ also belongs to A, with a certain False Positive Rate ϵ and zero False Negatives. Given a False Positive Rate, query time and space occupancy are crucial factors in determining how well a Bloom Filter performs. These variables are closely related to each other, as clearly stated in [11]. Convolutional and Recurrent NNs have been adopted in the most recent versions of Learned Bloom Filters [14, 27, 34, 42]. The reader who is interested in this particular Learned Data Structure can find an experimental comparative study in [19].
- **Learned Indexes.** They have been developed to address the so-called *Predecessor Search Problem*. Letting now A be a sorted table and given a query element $x \in U$, the Predecessor Search Problem entails locating the $A[j]$ such that $A[j] \leq x < A[j+1]$. In Sect. 2, a primer of the Learned Indexes methodology is provided, designed for a general audience due to the potential impact that this area may have on Machine Learning and Data Structures. As stated earlier, although the use of NNs to take advantage of novel computer architectures is one of the main motivations for this new area, unexpectedly and to the best of our knowledge, none of the Learned Indexes proposed so far, e.g., [4, 6, 7, 16, 17, 27, 31] to mention a few, use NNs. Our assessment is in agreement with what was reported in [30]. Even worse, no evaluation is available regarding the real role that NNs can have in the design and implementation of Learned Indexes.

  Although Dynamic Learned Indexes have been successfully designed, e.g., ALEX [15], PGM-Index [17] and SageDb [26], here we concentrate on the static version of the Predecessor Search Problem. Indeed, as evident in what follows, our finding in this specific context role out the use of NNs in the dynamic case.
- **Additional Learned Data Structures.** Both Learned Rank and Select Data Structures [10] and Learned Hash Functions [27] do not employ NNs. Once more, no analysis is available that justifies such a choice.

1.3 Our findings: the atomic case of neural networks in the design of learned indexes

Our original contribution to the development of Learned Data Structures is the first evaluation of the suitability of NNs as core elements of Learned Indexes, keeping in view the State of the Art described in Sect. 1.2. Atomic Learned Indexes (see Sect. 3.1) are taken into consideration in order to provide a clear comparative assessment of the potential usefulness of NNs in the aforementioned domain. They are the most basic models that come to mind. The justification for their choice is that in the case NNs do not provide any significant advancement with respect to very simple Learned Indexes, taking into consideration the results in [4, 7], even with the benefit of GPU processing, NNs have very little to offer to Learned Indexing. Basically, in this paper, we provide the following contributions.

- The first Learned Index design solely based on NN models. Because they provide an excellent balance between time effectiveness, space occupancy, and learning capacity [8], we opt for Feed Forward NNs. We refer to this Learned Index as an Atomic Learned Index because it lacks any additional ML subcomponents.
- A thorough experimental investigation on the performance of the Atomic Learned Index for both CPU and GPU processing.

- This Atomic Learned Index is thoroughly compared to analogous Atomic Learned Indexes that merely use linear regression[18] as the learned component. These models have been analysed and utilised in [4, 7]. More complicated models include them as building blocks, e.g, [4, 7, 16, 17, 27, 31].
- For completeness, and in order to highlight how delicate is to choose a Learned Index to effectively process a given table, we also compare the Atomic Learned Indexes with the ones that are considered State of the Art.

Our findings unequivocally show the need to design NNs specifically tailored for Learned Indexing, as opposed to Learned Bloom Filters. A work like this, which paves the road for the creation of NNs more tailored for Learned Indexing, is significant from a methodological perspective. In order to make this area accessible to researchers and engineers working on NNs, care is taken in providing an intuitive and easy-to-follow introduction to this area, that then "moves on" to account in full for the State of the Art. The software used for the experiments carried out in this paper can be found at [21].

1.4 Structure of the paper

The structure of this article is as follows. A high level presentation of the main ideas sustaining Learned Indexing is provided in Sect. 2. Atomic Learned Indexes (see Sect. 3.1) and the primary Hierarchical Learned Indexes (see Sect. 3.2) are both covered in Sect. 3, which provides an overview of Learned Indexes that have been actually proposed in the Literature. We describe the adopted experimental methodology in Sect. 4. Experiments and results are reported in Sect. 5. We present conclusions and future study directions in Sect. 6.

2 Classic and learned solutions for searching and indexing in a sorted set

Consider a sorted table A with n keys drawn from a universe U. As already stated in the Introduction, the Predecessor Search Problem, also known as Sorted Table Search, states that for a given query element x, return the $A[j]$ such that $A[j] \leq x < A[j+1]$. In the sections that follow, we detail both the traditional solutions to this problem and the method suggested by Kraska et al. [27] for turning it into a learning-prediction problem.

2.1 Traditional options: sorted table search techniques

Binary and Interpolation Searches are well-known algorithmic solutions to the Predecessor Search Problem. The first is optimal in a worst-case time setting under various computational models [2, 13, 25, 39], while the second has an excellent average case time performance for tables drawn uniformly and at random from the Universe U [33, 39]. Moreover, they both feature loops with a very small number of instructions, which makes them extremely quick, even in practice. For the purposes of this paper, we use two C++ implementations of Binary Search: a textbook one, which we refer to as Standard (abbreviated as **BS**), and an implementation of the Uniform variant (abbreviated as **US**), which was developed as part of a study by Khoung and Morin [22], following an approach suggested by Knuth [25]. Additionally, the Algorithms 1 and 2 provide the pseudo code for those two routines, respectively. Recalling that we are interested in Learned Sorted Table Search, as it will be evident in the remainder of this paper, we could have considered also Interpolation Search but, due to its poor performance in the Learned setting [3], we have excluded it for the experiments conducted in this paper.

Algorithm 1 Standard Binary Search in C++.

```
1: int StandardBinarySearch(int *A, int x, int left, int right){
2:     while (left < right) {
3:         int m = (left + right) / 2
4:         if(x < A[m]) right = m;
5:         else if( x > A[m]) left = m+1;
6:             else return m;
7:     }
8:     return right;
9: }
```

Algorithm 2 Uniform Binary Search in C++. The source code is taken from [22] (see also [25, 41]).

```
1:  int UniformBinarySearch(int *A, int x, int left, int right){
2:    const int *base = A;
3:    int n = right;
4:    while (n > 1) {
5:      const int half = n / 2;
6:      base = (base[half] < x) ? &base[half] : base;
7:      n -= half;
8:    }
9:    return (*base < x) + base - A;
10: }
```

2.2 Learned indexing: the essential ingredients

As already indicated, Kraska et al. [27] developed a novel strategy that converts the Predecessor Search Problem into a learning-prediction one. Referring to Fig. 1, the potential location of a query element in the table, in terms of an interval to search into, is returned by the model that was learned from the data. Binary Search is then used in that possibly smaller interval with respect to the entire table, in order to finalize the search. In other words, the model is used as an oracle to predict the query element position: the more accurate the oracle, the smaller the interval it returns. In addition to that, the oracle must return its prediction very quickly, possibly in constant time.

We now describe the simplest method for creating a model for A using Linear Regression and Mean Square Error Minimization [18], with the help of an example. In light of Fig. 2 and Table A in the caption, a Learned Index can be trained in the way described below.

- **Computation of the Cumulative Distribution Function (*CDF* for short) of a Sorted Table**. Regarding Fig. 2a, we may plot the elements of A on a graph, where the ordinates correspond to the ranks of the elements, and the abscissa displays the corresponding values of the elements. The outcome of this plot resembles a Cumulative Distribution Function *CDF* that underlines the table, as pointed out by Marcus et al [31]. The specific procedure illustrated here can be applied to any sorted table to obtain the mentioned discrete curve, which in the Literature is referred to as *CDF*.
- **Select a Model for the *CDF***. The discrete *CDF* must now be converted into a continuous curve. Fitting a straight line with the equation $F(x) = ax + b$ to the *CDF* is the simplest way to accomplish this. To get a and b in this case, we apply Linear Regression with Mean Square Error Minimization. This process is depicted in Fig. 2b, where the resulting a and b are, respectively, 0.01 and 0.85 when the array A specified in the figure legend is used as input.
- **Model Error Correction**. Since F is a rough estimate of the ranks of the elements in the table, using it to predict the rank of an element may result in an error e. For instance, in Fig. 2c, we apply the model to the element 398 and get a predicted rank of 4.68 instead of the actual rank of 7. This means that the error of the model $F(x) = 0.01 * x + 0.85$ on this element is $e = |7 - \lfloor 4.68 \rfloor| = 3$. We must thus correct this error before we can use the equation F to predict where an element x lies in the table. It is natural to correct such an approximate estimate by taking into account the maximum error: the largest gap ϵ between the actual rank of each element in the table and its rank as predicted by the model. Then, the search interval of an element x is specified to be $[F(x) - \epsilon, F(x) + \epsilon]$. In the example discussed in Fig. 2c, the resulting ϵ is equal to 3.

3 Learned indexes

In Sect. 2.2, we presented the essential ingredients of a Learned Index, also providing an example. However, those Data Structures are quite successful and key to the future development of Information Retrieval [28]. It comes as no surprise that many proposals for Learned Indexes have appeared in the Literature (see again [28]). In agreement with the State of the Art and in order to best appreciate the results of this paper, we provide synoptic details about the definition and functionality of a Learned Index. In particular, we can distinguish two types of indexes:

- **Atomic Learned Indexes**. They are the simplest type of Learned Indexes, i.e., as described in Sect. 3.1, they

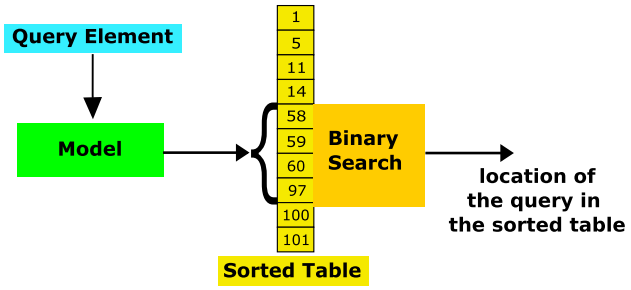

Fig. 1 A General Paradigm of Learned Searching in a Sorted Set [31]. The model is trained on the data in the table. Then, given a query element, it is used to predict an interval in the table of a reduced size where to search for the query (included in brackets in the figure). A binary search in the reduced interval is then applied to find the location of the query element in the table

consist of a single machine learning model capable of learning particular features of the data.

- **Hierarchical Indexes**. As described in Sect. 3.2, they use Atomic Learned Indexes as nodes of a tree-like structure to achieve better performance.

3.1 Atomic learned indexes

A model for Learned Indexes can be a linear function that models the CDF of the data, as described in Sect. 2.2. We identify a model as *Atomic* if it consists of a closed-form formula or a straightforward algorithm that estimates the CDF on a specific point (see Fig. 1). That is, it lacks any learned sub-component from the data.

3.1.1 Atomic learned indexes characterized by analytic solutions to regression problems

Regression is a technique to estimate a given function $G : \mathbb{R}^m \rightarrow \mathbb{R}$ using a particular function model $\tilde{G}$. Predictors and outcomes, respectively, are the terms used to describe the independent variables $x \in \mathbb{R}^m$ and the dependent variable $y \in \mathbb{R}$. By minimising a given error function calculated using a sample set of predictor-outcome measurements, the parameters of $\tilde{G}$ can be estimated. The Mean Square Error is the error function that is most frequently adopted, and there are various approaches to carry out the minimization task. Here, we adhere to the procedure described in [20], which specifically provides polynomial closed-form equations to solve the minimization of mean square error. When a geometric linear form is taken as a model for $\tilde{G}$, it is referred to as Linear Regression. Simple Linear Regression (SLR) is used when $m = 1$ and Multiple Linear Regression (MLR) in all other circumstances. The objective is to characterise the linear function model $\tilde{G}(\mathbf{x}) = \hat{\mathbf{w}}\mathbf{x}^T + \hat{b}$ by estimating the parameters $\hat{\mathbf{w}} \in \mathbb{R}^m$ and $\hat{b} \in \mathbb{R}$ using a given training set of n predictor-outcome couples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$. The design matrix $\mathbf{Z}$ can be defined as a matrix of size $n \times (m+1)$, where $\mathbf{Z}_i$ is the i-th row of $\mathbf{Z}$ and $\mathbf{Z}_i = [\mathbf{x}_i, 1]$. Additionally, $\mathbf{y}$ denotes a vector of size n, whose j-th component is indicated as y_j. The estimation is carried out by a Mean Square Error Minimization as follows:

$$\mathbf{MSE}(\mathbf{w}, b) = \frac{1}{n} \left\| [\mathbf{w}, b]\mathbf{Z}^T - \mathbf{y} \right\|_2^2 \quad (1)$$

Taking into consideration that **MSE** is a convex quadratic function on $[\mathbf{w}, b]$, it has a unique minimum that can be obtained by setting to zero its gradient $\nabla_{\mathbf{w},\mathbf{b}}$, whose value is

$$[\hat{\mathbf{w}}, \hat{b}] = \mathbf{y}\mathbf{Z}(\mathbf{Z}^T\mathbf{Z})^{-1} \quad (2)$$

The Simple Linear Regression case is characterized by a polynomial with degree $g = 1$. The general case of Polynomial Regression, which adopts a polynomial with degree $g > 1$, is a special case of Multiple Linear Regression, where

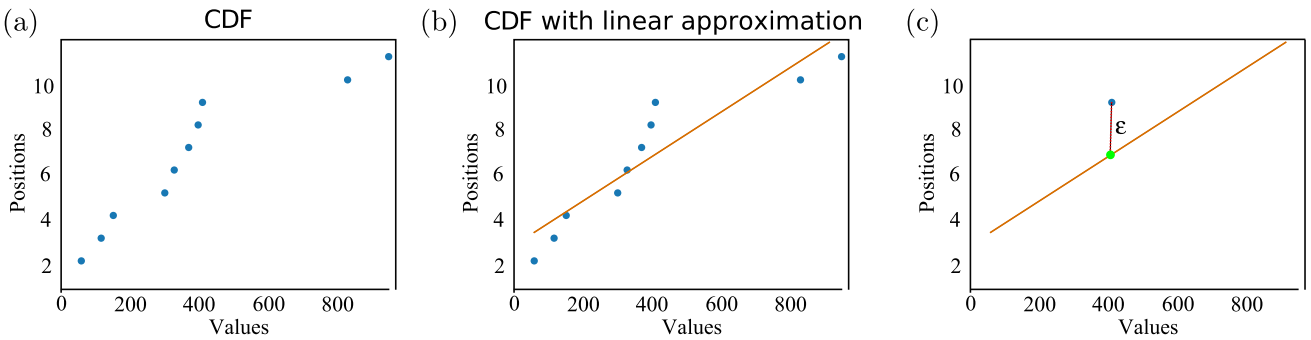

Fig. 2 The Linear Regression Method for Learning a Simple Model. Assume that A is [47, 105, 140, 289, 316, 358, 386, 398, 819, 939]. **a** The CDF of A is shown. In the diagram, the rank and value of each element in the table are indicated by their y and x coordinates, respectively. **b** By using linear regression, the values a and b of the equation $F(x) = ax + b$ are determined. **c** The highest error *epsilon* one can find using F is $epsilon = 3$, i.e. the largest difference between a rank of a value in the table and its predicted rank from F after rounding. For the provided query element x, the interval to search inside is given by $I = [F(x) - epsilon, F(x) + epsilon]$

$$\tilde{G}(\mathbf{z}) = \mathbf{w}\mathbf{z}^T + b \quad (3)$$

w is of size g and $\mathbf{z} = [x, .., x^{g-1}, x^g] \in \mathbb{R}^g$ is the predictor vector for **MLR**. In this study, we employ linear, quadratic and cubic regression models to estimate the function F provided by the underlying *CDF*. The relevant models are specifically prefixed with **L, Q,** or **C**, respectively.

3.1.2 Atomic learned indexes based on neural networks

The function $G : \mathbb{R}^m \rightarrow \mathbb{R}$ can be also learned by a NN. We concentrate on Multi-Layer Perceptrons (MLPs) Networks following the paper by Kraska et al [27]. MLPs are feed-forward networks and their general strategy entails an iterative training phase where the $\tilde{G}$ approximation is improved at each iteration. After starting with an initial approximation $\tilde{G}_0$, at each iteration i, an effort is made to reduce an error function E so that $E(\tilde{G}_{i-1}) \geq E(\tilde{G}_i)$. A training set T of examples is used for the minimization of E. The process can continue for a fixed number of steps or it can be terminated when, given a tolerance δ, $| E(\tilde{G}_{i-1}) - E(\tilde{G}_i) | \leq \delta$. We list the fundamental components that define the employed type of NN in the following.

1. **TOPOLOGY OF THE ARCHITECTURE**.
   (a) The standard Perceptron [8] with *relu* activation function represents the atomic element of our NN.
   (b) H, the number of Hidden Layers.
   (c) n_{h_i}, the Number of Perceptrons for each hidden layer h_i.
   (d) Fully Connected NN, i.e., each Perceptron of layer h_i is connected with each Perceptron of the next layer h_{i+1}.
2. **THE LEARNING ALGORITHM**.
   (a) E, the error function that measures how close is $\tilde{G}$ to G.
   (b) Starting from $\tilde{G}_0$, a gradient descent iterative process that at each step, approximates better and better G by reducing E. The parameters of each layer are changed via a backwards and forward pass. A learning rate characterizes the model, i.e., a multiplicative constant of the gradient error.
3. **THE TRAINING SCHEME**.
   (a) B, the size of a batch, i.e., the subset of elements to extract from the training set T. At each extraction of the batch, the model parameters are updated.
   (b) The number of epochs ne, i.e., the number of times the training set T is presented to the NN for the minimization of E.

A suitable training set is used by the learning algorithm of a NN, to carry out the iterative gradient descent process. The training data for the case of indexing, which is our goal, are in the form of scalar integers. It is necessary to represent a scalar integer x with a vector representation $\overrightarrow{x}$ to do a regression using a NN. As proposed by Kraska et al. for the same indexing problem [27], $\overrightarrow{x}$ is a string holding the 64-bit binary representation of x.

3.2 Learned indexes using hierarchical structures

The Atomic Learned Indexes have been utilised as the basis for more complex Learned Indexes with a hierarchical structure. Among the many proposals available, for this research, we consider the three indexes that appear to perform better than the others in the Literature [24, 31]. It is to be pointed out that, although all of the Atomic Indexes mentioned in Sect. 3.1 can be used as building boxes for the more complex indexes described next, to the best of our knowledge, and as already pointed out in the Introduction, none uses NNs.

3.2.1 The recursive model index

Historically, the first Learned Index proposal is the Recursive Model Index, or the **RMI** [27]. It is a Hierarchical Index with a tree-like structure, as seen in Fig. 4a. Its nodes are all Atomic Learned Indexes, just like those mentioned in Sect. 3.1. A prediction at each level indicates the model of the following level to be used for the next prediction, given a key to search for. The last level is achieved after continuing this process from the root. Finally, leaves provide a small interval for searching, which is given in input to the Binary Search. In order to obtain an **RMI** for a given Sorted Table, many hyperparameters must be specified, such as the number of levels, the number of nodes for each level, and the model to employ for each node. Which hyperparameters setting is best depends on the real context in which the specific index is employed, and its identification via efficient algorithms is an open problem. A partial solution to this is provided by the software platform **CDF-Shop** [32]: for a given dataset, it returns up to ten **RMI**s, that are identified via Combinatorial Optimization heuristics.

3.2.2 The piece-wise geometric model

The Piece-wise Geometric Model Index (abbreviated as **PGM**) [17] is a Learned Index that uses a piece-wise linear approximation algorithm [12] to estimate the *CDF* of the data, combined with a bottom-up procedure. The prediction error at each stage of its hierarchy is controlled by a user-defined approximation parameter. An example of the **PGM** is depicted in Fig. 4b and it is obtained as follows. The piece-wise linear approximation of the *CDF* provides three segments. The *CDF* of each segment is approximated by a linear model. By choosing the lowest values in each of the three segments, a new table is created. This new table is again divided into segments as in the first stage. The process is iterated until the resulting table consists of only one segment. The search for an element starts at the root of the **PGM** where the next model to query is selected until one of the segments is reached at the bottom. Then, Binary Search is used to finalize the query step.

3.2.3 The radix spline index

Another example of a bottom-up method to Learned Indexing is the Radix Spline Index (**RS** for short) [23], which, unlike the **PGM**, estimates the distribution through a spline curve [37]. As seen in Fig. 4c, a spline is constructed to roughly represent the *CDF* of the table, and the radix table is then utilised to determine which spline points should be used to narrow the search interval. A maximum approximation error is ensured for the **RS** using the user-defined value.

3.3 Prediction accuracy of a model

The approximation error is crucial in minimizing the size of the interval to search into, as it is clearly shown in Fig. 2. The part of the table where the last search must be done gets smaller while the error decreases. The percentage of the table that is no longer taken into account for searching after a prediction is the *reduction factor* (**RF**), which we use in this study to describe the accuracy in the prediction of a model. Because of the variability across the models to establish the search interval, and in order to place all models on par, we estimate empirically the **RF** of a model. In particular, we utilize a batch of queries and the model to decide how long the interval (I in Fig. 2) should be for each query. This allows us to simply calculate the reduction factor for that query. The reduction factor of the model for the specified table is then determined by averaging these reduction factors across the full set of queries. Note that the machine learning problem related to a learned index prediction does not involve any generalization error, since the problem is formulated in such a way that the predictions are always computed on the same dataset used for the training. As a consequence, the **RF** is always intended for the training.

4 Experimental procedure

4.1 Hardware

A workstation with an Intel Core i7-8700 3.2GHz CPU and an Nvidia Titan V GPU has been used for the experiments. A total of 32 Gbytes of DDR4 serve as the system memory. Additionally, the GPU is equipped with 12 Gbytes of DDR5 RAM on its own, and it uses the CUDA parallel computing framework. A PCIe 3 bus with a 32Gbyte/s bandwidth connects the CPU and GPU. Ubuntu LTS 20.04 is used as the operating system.

4.2 Datasets

We need to recall from Sect. 3.1.2 that an NN used in this research on a given integer dataset, requires that the 64 bits representation of each integer is transformed into a binary vector with 64 entries. Therefore, after training, the NN size is considerably larger than the original dataset used for training, having for this latter the possibility of packing each element in one memory word. Now, there exist large and real datasets that are *de facto* standards in terms of benchmarking of Learned Indexes [31]. However, they have a dimension in the Gigabytes and therefore, they are too large to be used in conjunction with NNs as Atomic Learned Indexes. This brings to light a limitation: NNs can be used only on relatively small datasets, i.e., size in the Mbs.

Given all of the above, in order to benchmark NNs for this study, we have resorted to a choice of datasets that have also appeared in the Learned Indexing Literature at the early stages of its development, e.g., [17, 27]. In particular, as detailed next, we use real and artificial datasets. The first type has a size that NNs can work with, while the size of the second type can be determined so that NNs have no space problem.

1. **Uni** collects data sampled from a Uniform distribution, defined as

$$U(x,a,b) = \begin{cases} \dfrac{1}{b-a} & \text{if } x \in [a,b] \\ 0 & otherwise \end{cases} \tag{4}$$

where $a = 1$ e $b = 2^{r-1} - 1$, where r is the CPU integer precision. It contains 1.05e+06 integers and has a size of 1.10e+04 Kb.

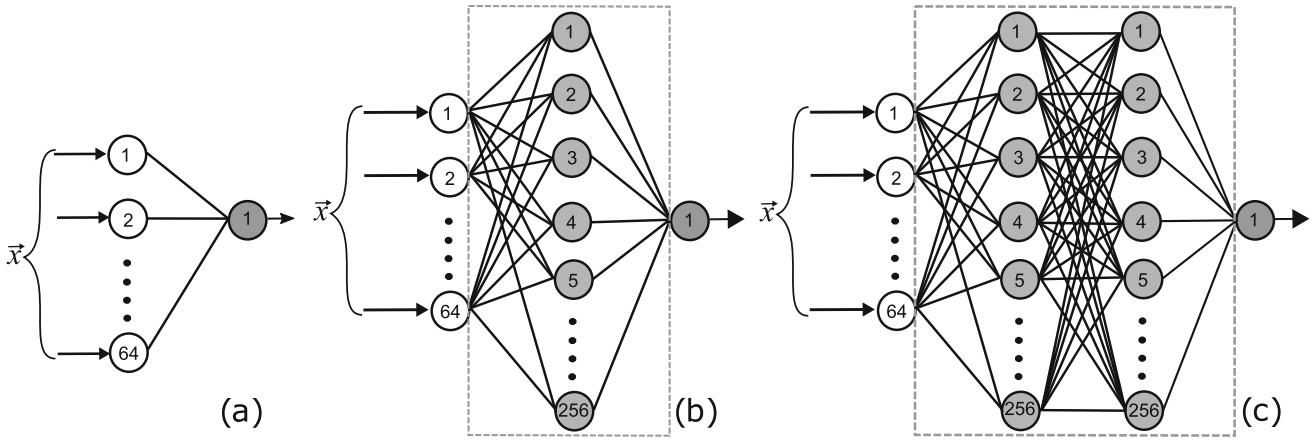

Fig. 3 The neural network architectures used in this research. We use the notation: **a NN0** for no hidden layers; **b NN1** for one hidden layer; and **c NN2** for two hidden layers. The number of input neurons is 64, each layer has 256 units

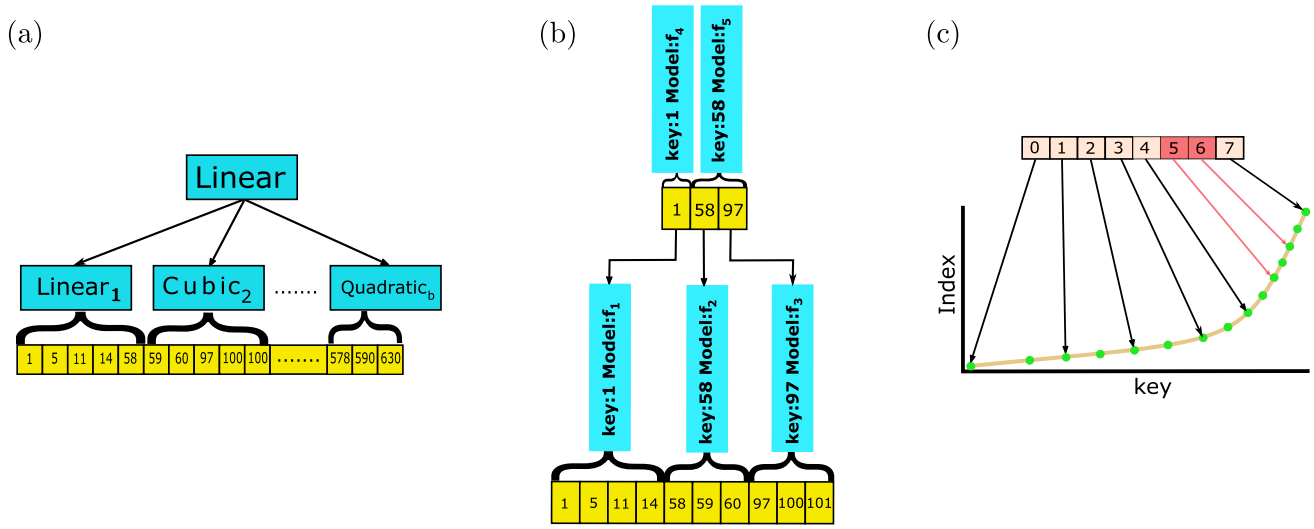

Fig. 4 Examples of different hierarchical learned indexes. **a** A **RMI** example with two layers and a branching factor b. The top box indicates that a linear function is used to choose the lower models. Regarding the leaf boxes, each one identifies which of the Atomic Learned Indexes is applied to the relevant part of the table where making a prediction. **b** An illustration of a **PGM** index. The table is broken into three segments at the bottom. In this manner, a new table is built, and the procedure is repeated. **c** A **RS** illustration. The bins at the top are where the elements fall according to their top three digits. A linear spline with appropriately selected spline points that approximates the CDF of the data is shown at the bottom. Each bucket points to a spline point, and as a consequence when a query element falls into a bucket (let's say six), the search interval is restricted by the spline points pointed by that bucket and the one before it (five in our example)

2. **Logn** collects data sampled from a Log-normal distribution, defined as

$$L(x, \mu, \sigma) = \frac{e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}}{x\sqrt{2\pi}\sigma} \tag{5}$$

where $\mu = 0$ is the means and $\sigma = 1$ is the variance of the distribution. It contains 1.05e+06 integers and has a size of 1.05e+04 Kb.

1. **Real-wl** collects timestamps of ∼715 M requests performed by a web server. It contains 3.16e+07 integers and has a size of 3.48e+05 Kb.
2. **Real-iot** collects timestamps of ∼26 M events recorded by IoT sensors. It contains 1.52e+07 integers and has a size of 1.67e+05 Kb.

The CDFs of each of those datasets are reported in Fig. 5. As evident from that figure, the **logn** dataset has a CDF that may be challenging to learn, as stated in [27], while the other datasets follow a uniform distribution and their CDF can be considered easy to learn.

For each of the aforementioned tables, the query dataset is equally divided across elements that are present in the table and those that are absent. Its size is equal to 50% of the reference dataset. The query datasets for all experiments are not sorted.

4.3 Atomic learned indexes and binary search

We use the standard Binary Search strategy, and in addition to it, Uniform Binary Search [22, 25] for the last search step. They are the routines **BS** and **US** of Sect. 2.1, respectively. In terms of Atomic Learned Indexes, we employ **L, Q,** and **C**. As for NNs, and as mentioned already, we follow the initial proposal by Kraska et al. [27], and use only Multi-layer Perceptron Networks with an increasing complexity of architecture in terms of hidden layers, as detailed below. Specifically, **NN0** denotes an NN with no hidden layers (see Fig. 3a), **NN1** one with one hidden layer (Fig. 3b), and **NN2** one with two hidden layers (Fig. 3c). Each layer has 256 units. For the sake of clarity, we anticipate that, as is evident from the results discussed in Sect. 5.2.2, no further studies are needed regarding the investigation of network hyperparameters or more complex architectures such as the Convolutional (CNN), as even the simplest networks considered here turn out not to be competitive in terms of query time with the other models investigated. Two Atomic Learned Indexes are provided by each of the models mentioned above, one for each Binary Search routine utilized in the final search stage. However, **US** may be streamlined to prevent "branchy" instructions in its implementation, according to Khuong and Morin [22], yielding better performance with respect to **BS**. For this reason, it performs better than **BS** in our experiments and, for conciseness, we report only the experiments regarding **US**. Given an input table, all Atomic Indexes are built according to the procedures outlined in Sect. 3.1.

4.4 Hierarchical learned indexes

As for Hierarchical Indexes, we consider **RMI**, **PGM** and **RS**. We use the *Search on Sorted Data* (**SOSD** for short) [31] platform for model training of the **PGM** and the **RS**, for a given table. However, as already mentioned, for the training of the **RMI**, we use **CDF-Shop**. As for query processing, for each Hierarchical Index, each batch of queries is executed within **SOSD**. It should be noted that the main version of the **SOSD** platform only offers a Uniform Binary Search implementation as the final stage of the model that utilises the standard C++ **lower_bound** procedure. A version of this platform suitably modified for this research is described in [6].

Table 1 Training time and reduction factor for atomic learned indexes

	Uni		Logn	
	TT (s)	RF (%)	TT (s)	RF (%)
NN0	2.55e−04	94.08	1.39e−04	54.40
NN1	4.18e−04	99.89	3.79e−04	94.21
NN2	4.49e−04	99.87	8.60e−04	97.14
L	8.20e−08	99.94	5.61e−08	77.10
Q	1.27e−07	99.98	1.02e−07	90.69
C	1.84e−07	99.97	1.74e−07	95.76
	Real-wl		Real-iot	
	TT (s)	RF (%)	TT (s)	RF (%)
NN0	2.50e−04	99.99	1.28e−04	89.90
NN1	2.31e−04	99.88%	4.20e−04	98.54
NN2	2.33e−04	99.80	3.57e−04	97.31
L	5.82e−08	99.99	7.70e−08	96.48
Q	1.14e−07	99.99	1.25e−07	99.10

The training time for each element, indicated in seconds (column **TT (s)**), and the percentage of table reduction (column **RF (%)**), as defined in Sect. 3.3, are displayed for each dataset, indicated in the first row of the tables, and each model, indicated in the first column. **L**, **Q** and **C** are the linear, quadratic and cubic atomic models while **NN0, NN1, NN2** indicate NN models with 0, 1 and 2 hidden layers respectively

5 Experiments and findings

The datasets outlined in Sect. 4.1 have been considered. To use them as input for the NNs, both training and query datasets are modified as described at the end of Sect. 3.1.2. In regard to both training and query times, we report the average per element. That is, for query processing, we take the total time to process a batch with a given Learned Index and then divide that time by the number of items in the batch. An analogous procedure is followed for training on an input dataset. This method of collecting timing results is in agreement with the Literature since it assures a reliable measure of time performance [31]. We report results regarding both training and querying as follows.

- **Training**. A comparison between GPU and CPU training is performed and reported in Sect. 5.1. In particular, we use the highly engineered Tensorflow platform with GPU support to train NNs Atomic Indexes. As far as **L**, **Q** and **C** are concerned, we use a CPU implementation for their training.
- **Batch Query Processing**. We perform four different kinds of experiments in order to study the competitiveness of NNs Atomic Indexes with respect to the other

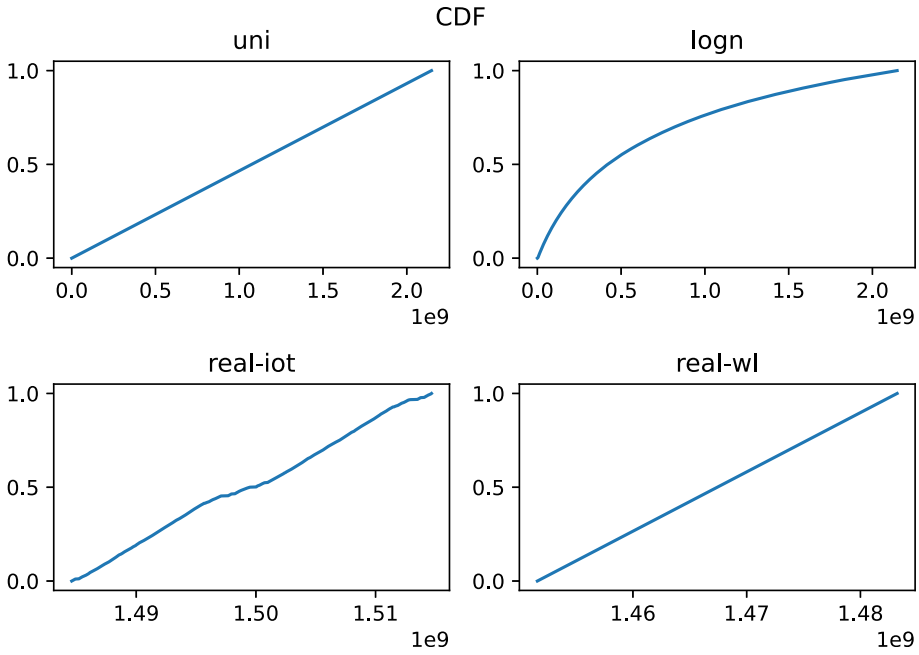

Fig. 5 Datasets CDF. For each dataset used in this paper, we report in the *x*-axis the value of each element in the dataset and in the *y*-axis, its cumulative probability computed as in [27]

Learned Indexing models considered in this research, and also in regard to the various hardware and software solutions that we have available.

- **TensorFlow**. In order to do query searches with the Learned Indexes based on NNs, we have carried out query experiments using Tensorflow with GPU support. Because of the overhead of uploading Tensorflow to the GPU, results are quite poor and hence not reported. This is consistent with the outcomes mentioned in [27]. For this reason, Tensorflow was exclusively utilized to train NNs.
- **GPU and NNs vs Parallel Binary Search**. We employ two C++ CUDA implementations, one of the Learned Index corresponding to **NN0**, and the other relative to a parallel version of **BS**. Section 5.2.1 shows a report of the findings highlighting that using the GPU is not advantageous, also in comparison with the baseline **BS** implemented in CUDA. As a result, no additional GPU experiments have been carried out. We point out that here we use **BS** due to the technical difficulties a C++ CUDA implementation of **US** imposes, *de facto* making it analogous to **BS**.
- **CPU only**.

  Atomic Learned Indexes. We have carried out all of the experiments using the Atomic Learned Indexes considered for this research. For the sake of clarity and as anticipated, because the results on **BS** would contribute very little to the discussion, we only provide the results using **US**. Section 5.2.2 reports and discusses them.

Table 2 Training time and reduction factor of neural networks for different sizes of the logn training set

Sampling percentage	NN0		NN1		NN2	
	TT (s)	RF (%)	TT (s)	RF (%)	TT (s)	RF (%)
25%	4.77e−04	15,76%	4.07e−03	63,42	4.44e−03	83,71
50%	2.6e−04	17,09%	2.43e−03	64,23	3,57e-03	83,94
75%	3.13e−04	16,52%	2.42e−03	63.90	5,06e-03	82.61
100%	1.39e−04	54.40%	3.79e−04	94.21	8.60e−04	97.14

The first column indicates the sample size, while the remaining part of the table is organized as Table 1

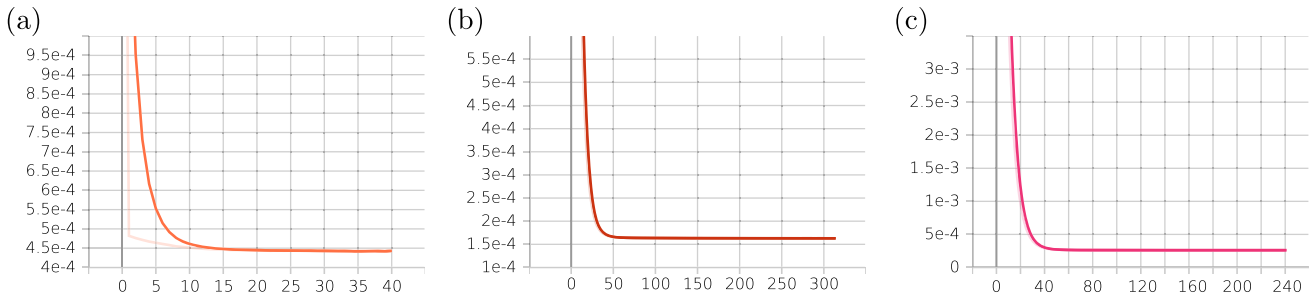

Fig. 6 Training Curve on a Sample of Size 25% of the logn dataset. Each figure reports the **MSE** (ordinate) over the epochs (abscissa) for the given training sample. **NN0**, **NN1** and **NN2** are reported from left to right

Table 3 A comparison of prediction accuracy on the logn dataset: atomic learned indexes versus hierarchical ones

Indexes	Logn (%)
NN2	97.14
C	95.76
RMI	99.99
PGM	99.99
RS	99.99

For conciseness, only the most complex Atomic Learned Indexes are reported in the first two rows of the table (**NN2, C**). The hierarchical models are **RMI, PGM, RS**. The values in the table are the **RF** of each of the considered Indexes on the **logn** dataset

Table 4 Query time on GPUs

Methods	Copy (s)	Op. (s)	Search (s)	Query (s)
NN0-BS	3.27e−08	4.20e−09	1.84e−09	3.27e−08
BS	2.55e−09	-	1.89e−09	4.44e−09

Binary Search using **NN0** as the prediction step is referred to as **NN0-BS**, whereas GPU-based parallel Binary Search without a prior prediction is indicated as **BS**. We report various time results, in seconds and per element (averaged over an entire batch of queries), as follows. The time for CPU-GPU copy operations and vice versa (column **Copy (s)**); the time for math operations (column **Op. (s)**), the time to search into the interval with **BS** (column **Search (s)**), and the overall time to finish the query process (column **Query (s)**)

Insights into the Learning the Complex CDF of a Sorted Table. The Hierarchical Learned Indexes, i.e., **RMI**, **PGM** and **RS**, are the most competitive in terms of query time. Therefore, it is natural to ask what is the gap that NNs have in performance with respect to the most advanced Learned Indexes. As a complement to this assessment, we also show that the choice of a Learned Index for a given Sorted Table depends critically on the complexity of the CDF that needs to be learned. Section 5.2.3 reports and discusses the findings of this set of experiments.

5.1 Training: GPU versus CPU

The training times for the experiments outlined at the beginning of this section are reported in Table 1, along with the corresponding **RF**, which was calculated as explained in Sect. 3.3. The training time for Atomic Learned Indexes **L**, **Q** and **C** is equal to the time required to solve Eq. (2). The stochastic gradient descent learning approach is employed for NN models, with a momentum parameter of 0.9 and a learning rate of 0.1. A number of epochs equal to 2000 are used with a batch size of 64. All those values have been set via trial and error, observing that increasing batch size and number of epochs does not lead to improvements. As shown by the findings listed in Table 1, the size of the employed NNs affects the reduction factor and training time. In particular, the more layers, the better the reduction factor and the higher the training time. Moreover, NNs are not competitive with the **L**, **Q** and **C** Atomic Learned Indexes, both in training time and **RF**.

In regard to training time, this lack of performance holds even with GPU support and the use of the highly-engineered Tensorflow platform. In fact, for each dataset, the training time for NNs is four orders of magnitude longer than that for non-NN Atomic Learned Indexes with a similar **RF**, although those latter use a CPU and therefore they do not benefit from parallelism. Since the training of NN models is performed via Tensorflow, it is not possible, to the best of our knowledge, to profile the CPU-GPU I/O time, i.e., the time to input the data and get the trained NN, and compare it with the training phase that is performed in parallel within the GPU. Despite such a shortcoming and in

Table 5 Query time of NN atomic learned indexes on CPU for each of the considered datasets

Dataset	US	NN0-US	NN1	NN2
Uni	2.81e−07	1.31e−07	1.56e−06	5.16e−06
Logn	2.08e−07	1.92e−07	1.69e−06	5.24e−06
Real-wl	3.38e−07	4.59e−07	Space Error	Space Error
Real-iot	3.07e−07	4.76e−07	1.90e−06	1.94e−05

The datasets are indicated in the first column. The time taken by Uniform Binary Search alone is indicated in the column named **US**, while its version using **NN0** as the prediction step is indicated by the column **NN0-US**. The other columns refer to the time taken by **NN1** and **NN2** only for the interval prediction. The time is in seconds and is per query element (averaged over an entire batch of queries). The label *Space Error* indicates the case when the queries are too big to fit in the main memory

Table 6 Query time of non-NN atomic learned indexes on CPU for each of the considered datasets

Dataset	US	L-US	Q-US	C-US
Uni	2.81e−07	9.42e−08	8.11e−08	9.39e−08
Logn	2.08e−07	1.60e−07	1.59e−07	1.54e−07
Real-wl	3.38e−07	5e05e-08	2.12e−7	1.80e−7
Real-iot	3.07e−07	8.32e−08	1.99e−7	2.57e−7

Results with Linear, Quadratic and Cubic models are reported and indicated in columns **L-US**, **Q-US**, **C-US** respectively. The first two columns are the same as in Table 5. Every time in the Table is represented in seconds and is per query element (averaged over an entire batch of queries)

Table 7 Query time of hierarchical indexes on CPU for each of the considered datasets

Dataset	US	RMI	PGM	RS
Uni	2.81e−07	1.5e−07	1.62e−07	1.66e−07
Logn	2.08e−07	1.45e−07	1.59e−07	1.66e−07
Real-wl	3.38e−07	1.11e−07	1.26e−07	1.59e−7
Real-iot	3.07e−07	1.36e−07	1.45e−07	1.68e−7

The columns **RMI**, **PGM** and **RS** report the results with the considered Hierarchical Learned Indexes. The first two columns are the same as in Table 5. Every time in the Table is represented in seconds and is per query element (averaged over an entire batch of queries)

view of the results reported in Sect. 5.2.1, it is reasonable to justify such a result with the fact that the I/O time dominated the GPU training time, to the point of making the use of parallelism detrimental. Given the order of magnitude difference in training time between NN and other Atomic Models training, such a picture is unlikely to change even if the I/O bandwidth CPU/GPU increases by constant factors.

Regarding the **RF** performance, the support for NN use was motivated by Kraska et al. with their ability to learn "complex patterns" in the data that could not be captured by Atomic Models based on regression. This may be the case, as indicated by comparing the **RF** performance of the NNs on the **logn** dataset with that of the other Atomic Learned Indexes.

As far as NNs are concerned, it is of interest to assess how **TT** and **RF** vary with respect to the size of the training set. We have experimented only with the "most difficult" of the datasets, i.e., **logn**. Specifically, for each of **NNO**, **NN1** and **NN2**, different sizes of the training set have been considered, ranging from 25% to 100%. The results are reported in Table 2. It is clear that the *RFs* increase as the size of the training set and the complexity of the networks increase, with a corresponding increase in **TT**. It is worth recalling from Sect. 3.3 that the machine learning problem related to the learned index prediction does not involve any generalization error, so the *RF* on validation or test are not provided. For completeness, Figs. 6-9 depict the **MSE** of the three models with respect to the learning epochs, on different training set sizes.

Given all of the above regarding NNs, it seems to be more profitable, in terms of prediction accuracy, to resort to the Hierarchical Learned Indexes that divide the CDF to be learned into pieces, possibly obtaining a set of "simpler" curves to learn. Such a fact is illustrated in Table 3 for the **logn** dataset, where the Hierarchical Learned Indexes have a nearly perfect prediction. Note that, in this table, we report only the reduction factor of the NN2 model because it has shown the best results in terms of training time and reduction factor in regard to the experiments conducted on all the datasets, reported in Table 1.

5.2 Query

5.2.1 GPU and NNs versus parallel binary search

Assuming as a baseline a simple parallel implementation of Binary Search, i.e., **BS**, in order to determine whether there is actually a benefit to using the GPU for queries in conjunction with NNs, we conduct an experiment by comparing the former classic and simple parallel routine with **NN0** on the simplest of the datasets to learn, i.e., **uni**. Such a choice provides an advantage to the NN. As already mentioned, we could not use the Uniform version of Binary Search, i.e., **US**, on the GPU because it was difficult to import within CUDA. The results are reported in Table 4. They clearly show that on GPUs, the usage of NNs on this architecture is superfluous because a traditional parallel

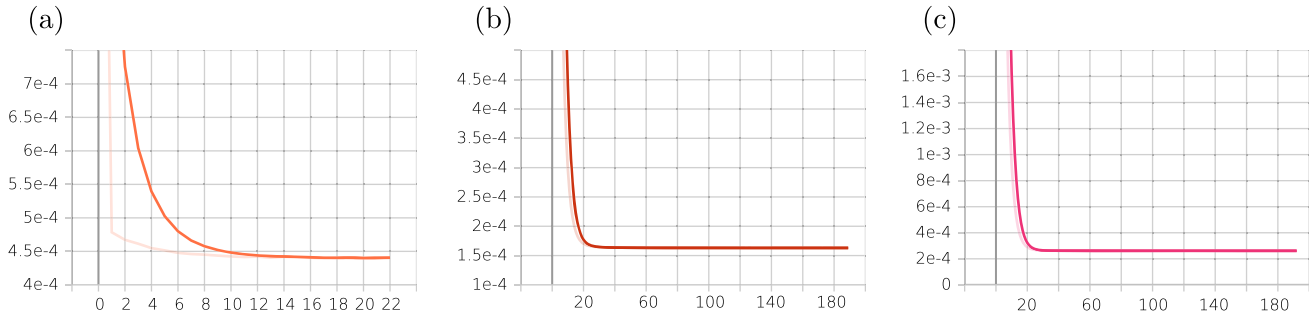

Fig. 7 Training curve on a sample of size 50% of logn dataset. The legend is as in Fig. 6

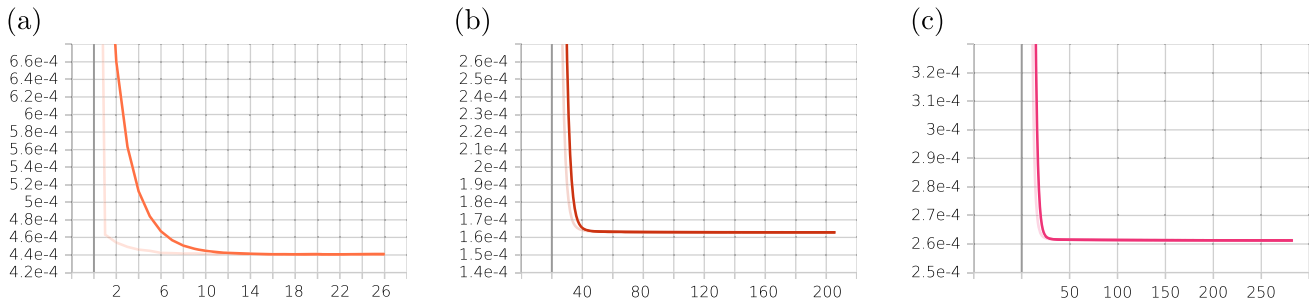

Fig. 8 Training curve on a sample of size 75% of the logn dataset. The legend is as in Fig. 6

Binary Search on the GPU is by itself faster than its Learned counterparts. It is also of interest to point out that copy operations from CPU to GPU and vice versa cancel out the one order of magnitude speedup of maths operations. The lectures to be learned from this experiment are rather subtle. Indeed, it is the case that GPU code execution favours straight-line math operations (and therefore NNs) with respect to if-then-else constructs (Binary Search). However, transfer from the Main Memory to the GPU may play a key role in cancelling eventual time gains. Moreover, recalling that the joint usage of GPU and NNs was once again one of the motivations to resort to Learned Indexing approaches, we find that when one is willing to use the GPU, a simple parallel implementation of standard Binary Search is enough. That is, the Learned Indexing framework is certainly of success, but not with the components initially envisaged. In particular, GPU usage.

5.2.2 CPU only: atomic learned indexes

We take Uniform Binary Search **US** as a baseline to compare against the Atomic Learned Indexes. Such a choice is motivated by the fact that results reported in [22] indicate that **US** is usually faster than **BS** on modern computer architectures. It results convenient to report the experiments regarding NNs separately (Table 5) with respect to the ones regarding the other Atomic Indexes (Table 6). From those results, we have that only **NN0** is competitive on the artificial datasets with respect to the baseline. As for the other two NNs, either they are not competitive even for prediction time only, or they run out of space. This latter fact is due to the coding of the query data that must be used for its use in conjunction with the NNs and that causes an expansion of the query set size, (see Sect. 3.1.2). Therefore, NNs more complex than **NN0** are slow in time and costly in space. As for the remaining Atomic Indexes, they all report a gain in time with respect to the baseline. In conclusion, NNs are not very competitive as Atomic Learned Indexes.

5.2.3 CPU only: insights into the complexity of learning the CDF of a sorted table

Table 7 contains the findings of the query experiments performed with the use of the Hierarchical Learned Indexes on the datasets considered for the Atomic Indexes. A comparison with Table 5 shows that NNs are not competitive with Hierarchical Learned Indexes unless the CDF of the input Table is particularly easy to learn (see the CDF of **uni** in Fig. 5). In this case, only **NN0** is worth consideration. It is also of interest to compare the Hierarchical Indexes with the remaining Atomic ones, whose results are reported in Table 6. With the exclusion of the somewhat difficult-to-learn table **logn**, the Atomic **L** Index is much

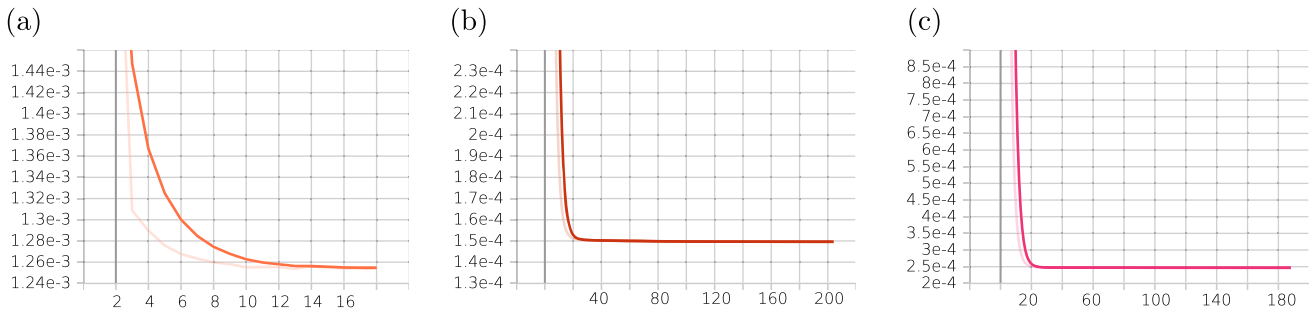

Fig. 9 Training curve on the full logn dataset. The legend is as in Fig. 6

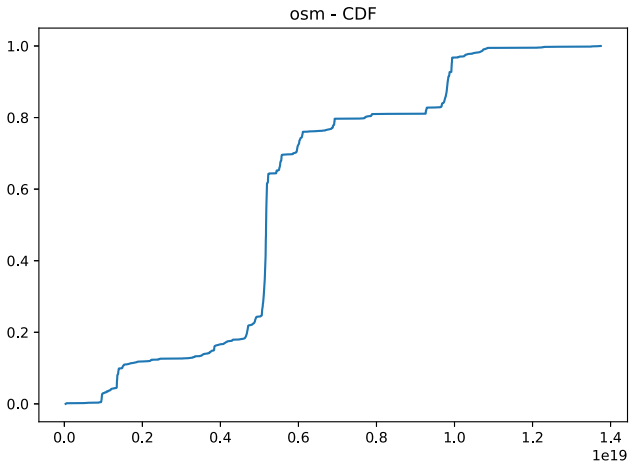

Fig. 10 OSM Dataset CDF. The legend is as in Fig. 5

better than the Hierarchical Learned Indexes. This is due to the regularity of the CDFs to be learned, not requiring complicated models.

In order to better illustrate the above point, we have performed an additional experiment with the use of the Open Street Map (**osm** for short) dataset. It is the most difficult in terms of the CDF, among the real benchmark datasets used in [31] and that, as already pointed out, are too large for the NNs. This dataset comprises 200 M 64-bits integers representing the cell IDs of embedded locations in Open Street Map. Its CDF is depicted in Fig. 10. The query time results of the Hierarchical Learned Indexes and of the Regression-based ones are reported in Table 8. The **Q** Index has been excluded for brevity since it performs as the other two Atomic ones. As evident from those results, the Hierarchical Learned Indexes are now competitive with respect to the Atomic ones.

The lesson to be drawn here is that NNs are far from being competitive with Hierarchical Indexes which, in turn, are competitive with respect to Atomic Learned Indexes only on Sorted Tables with a complex CDF curve to learn.

6 Conclusions

A perceived paradigm shift is one of the motivations for the introduction of the Learned Indexes. Despite that, the use of a GPU architecture for Learned Indexes based on NNs seems not to be appropriate when we use generic NNs as we have done here. It is to be pointed out that certainly, the use of a GPU accelerates the performance of math operations, but the data transfer between CPU and GPU is a bottleneck in the case of NNs: not only data but also the size of the model matters. When we consider CPU only, NN models are not competitive with very simple models based on Linear Regression. This research clearly points to the need to design NN architectures specialized for Learned Indexing, as opposed to what happens for Bloom Filters where generic NN models guarantee good performance to their Learned versions. In particular, those new NN models must be competitive with the Atomic Learned Indexes based on Linear Regression, which are widely used

Table 8 Query time of hierarchical indexes on difficult datasets and a comparison with atomic indexes

Dataset	US	L	C	RMI	PGM	RS
OSM	6.85e−07	6.67e−07	5.49e−07	2.75e−07	1.72e−07	1.71e−07
Logn	2.08e−07	1.60e−07	1.54e−07	1.45e−07	1.59e−07	1.66e−07

The Table reports query times of the search alone (column **US**), of Atomic Indexes (columns **L** and **C**, respectively), and of Hierarchical Learned Indexes (columns **RMI**, **PGM** and **RS**, respectively) on two "difficult" datasets, one real (**osm**) and one synthetic (**logn**). Every time in the Table is represented in seconds and is per query element (averaged over an entire batch of queries)

as building blocks of more complex models [4, 7, 31]. It is to be remarked that we have considered the static case only, i.e., no insertions or deletions are allowed in the table. The dynamic case has also been considered in the literature, i.e., [15, 17]. However, for that setting, no NN solution is available. In conclusion, this study provides solid grounds and valuable indications for the future development of Learned Data Structures, which would include a pervasive presence of NNs.

Acknowledgements This research is funded in part by the MIUR Project of National Relevance 2017WR7SHH "Multicriteria Data Structures and Algorithms: from compressed to learned indexes, and beyond". We also acknowledge an NVIDIA Higher Education and Research Grant (donation of a Titan V GPU). Raffaele Giancarlo is also partially supported by INDAM- GNCS Project 2023 "Approcci computazionali per il supporto alle decisioni nella Medicina di Precisione". Giosuè Lo Bosco is also supported by the University of Palermo FFR (Fondo Finalizzato alla ricerca di Ateneo) year 2023.

Funding Open access funding provided by Università degli Studi di Palermo within the CRUI-CARE Agreement.

Data Availability The data that support the findings of this study are available on request to the corresponding author.

References

1. Abadi M (2015) Tensorflow: large-scale machine learning on heterogeneous distributed systems. http://download.tensorflow.org/paper/whitepaper2015.pdf
2. Aho AV, Hopcroft JE, Ullman JD (1974) The design and analysis of computer algorithms
3. Amato D (2022) A tour of learned static sorted sets dictionaries: from specific to generic with an experimental performance analysis. Ph.D. thesis
4. Amato D, Lo Bosco G, Giancarlo R (2021) Learned sorted table search and static indexes in small model space. In: AIxIA 2021—advances in artificial intelligence: 20th international conference of the Italian association for artificial intelligence, virtual event, December 1–3, 2021, Revised Selected Papers. Springer, Berlin, Heidelberg, pp 462–477
5. Amato D, Lo Bosco G, Giancarlo R (2022) On the suitability of neural networks as building blocks for the design of efficient learned indexes. In: Iliadis L, Jayne C, Tefas A, Pimenidis E (eds) Engineering applications of neural networks. Springer, Cham, pp 115–127
6. Amato D, Lo Bosco G, Giancarlo R (2023) Standard versus uniform binary search and their variants in learned static indexing: the case of the searching on sorted data benchmarking software platform. Softw Pract Exp 53(2):318–346
7. Amato D, Giancarlo R, Lo Bosco G (2023) Learned sorted table search and static indexes in small-space data models. Data 8(3)
8. Bishop CM (1995) Neural networks for pattern recognition. Oxford University Press, New York
9. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
10. Boffa A, Ferragina P, Vinciguerra G (2021) A "learned" approach to quicken and compress rank/select dictionaries. In: Proceedings of the SIAM symposium on algorithm engineering and experiments (ALENEX)
11. Broder A, Mitzenmacher M (2003) Network applications of bloom filters: a survey. Internet Math 1(4):485–509
12. Chen DZ, Wang H (2009) Approximating points by a piecewise linear function. Algorithmica 66:682–713
13. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms. The MIT Press, New York
14. Dai Z, Shrivastava A (202) Adaptive learned bloom filter (Ada-BF): efficient utilization of the classifier with application to real-time information filtering on the web. In: Larochelle H, Ranzato M, Hadsell R, Balcan MF, Lin H (eds) Advances in neural information processing systems, vol 33. Curran Associates, Inc., pp 11700–11710
15. Ding J, Minhas UF, Yu J, Wang C, Do J, Li Y, Zhang H, Chandramouli B, Gehrke J, Kossmann D, Lomet D, Kraska T (2020) Alex: an updatable adaptive learned index. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data, SIGMOD '20, New York. Association for Computing Machinery, pp 969–984
16. Ferragina P, Vinciguerra G (2020) Learned data structures. In: Recent trends in learning from data. Springer, pp 5–41
17. Ferragina P, Vinciguerra G (2020) The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. PVLDB 13(8):1162–1175
18. Freedman D (2005) Statistical models: theory and practice. Cambridge University Press, Cambridge
19. Fumagalli G, Raimondi D, Giancarlo R, Malchiodi D, Frasca M (2022) On the choice of general purpose classifiers in learned bloom filters: an initial analysis within basic filters. In: Proceedings of the 11th international conference on pattern recognition applications and methods (ICPRAM), pp 675–682
20. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. The MIT Press, New York
21. http://tinyurl.com/bench-atomic-learned-indexes. Last accessed 06, Feb 2023
22. Khuong PV, Morin P (2017) Array layouts for comparison-based searching. J Exp Algorithmics 22:1.3:1-1.3:39
23. Kipf A, Marcus R, van Renen A, Stoian M, Kemper A, Kraska T, Neumann T (2020) Radixspline: a single-pass learned index. In: Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management, aiDM '20. Association for Computing Machinery, pp 1–5
24. Kipf A, Marcus R, van Renen A, Stoian M, Kemper A, Kraska T, Neumann T. SOSD Leaderboard. https://learnedsystems.github.io/SOSDLeaderboard/leaderboard/
25. Knuth DE (1976) The art of computer programming, vol 3 (Sorting and Searching)
26. Kraska T, Alizadeh M, Beutel A, Chi EH, Ding J, Kristo A, Leclerc G, Madden S, Mao H, Nathan V (2021) Sagedb: a learned database system

27. Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N (2018) The case for learned index structures. In: Proceedings of the 2018 international conference on management of data. ACM, pp 489–504
28. Kraska T (2021) Towards instance-optimized data systems. Proc. VLDB Endow. 14(12):3222–3232
29. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436
30. Maltry M, Dittrich J (2022) A critical analysis of recursive model indexes. Proc VLDB Endow 15(5):1079–1091
31. Marcus R, Kipf A, van Renen A, Stoian M, Misra S, Kemper A, Neumann T, Kraska T (2020) Benchmarking learned indexes. Proc VLDB Endow 14(1):1–13
32. Marcus R, Zhang E, Kraska T (2020) CDFShop: exploring and optimizing learned index structures. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data, SIGMOD'20, pp 2789–2792
33. Mehlhorn K, Tsakalidis A (1991) Data structures. In: Handbook of theoretical computer science, vol. A: algorithms and complexity. MIT Press, Cambridge, pp 302–341
34. Mitzenmacher M (2018) A model for learned bloom filters and optimizing by sandwiching. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds) Advances in neural information processing systems, vol 31. Curran Associates, Inc
35. Mitzenmacher M, Vassilvitskii S (2020) Algorithms with predictions. CoRR: abs/2006.09123
36. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 38:8
37. Neumann T, Michel S (2008) Smooth interpolating histograms with error guarantees
38. Ohn I, Kim Y (2019) Smooth function approximation by deep neural networks with general activation functions. Entropy 21(7):627
39. Peterson WW (1957) Addressing for random-access storage. IBM J Res Dev 1(2):130–146
40. Sato K, Young C, Patterson D (2017) An in-depth look at Google's first tensor processing unit. https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu
41. Schulz L, Broneske D, Saake G (2018) An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. Proc VLDB Endow 11:1550–1562
42. Vaidya K, Knorr E, Kraska T, Mitzenmacher M (2020) Partitioned learned bloom filter. ArXiv: abs/2006.03176
43. Wang B (2017) Moore's law is dead but GPU will get 1000x faster by 2025. https://www.nextbigfuture.com/2017/06/moore-law-is-dead-but-gpu-will-get-1000x-faster-by-2025.html