IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 21 June 2023, accepted 9 July 2023, date of publication 14 July 2023, date of current version 24 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3295434

UNSOLVED PROBLEM

On Nonlinear Learned String Indexing

PAOLO FERRAGINA[1], MARCO FRASCA[2], GIOSUÈ CATALDO MARINÒ[2], AND GIORGIO VINCIGUERRA[1]

[1]Department of Computer Science, University of Pisa, 56127 Pisa, Italy
[2]Department of Computer Science, University of Milan, 20133 Milan, Italy

Corresponding author: Giorgio Vinciguerra (giorgio.vinciguerra@unipi.it)

This work was supported by the European Union – Horizon 2020 Program under the scheme "INFRAIA-01-2018-2019 – Integrating Activities for Advanced Communities", Grant Agreement n. 871042, "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (http://www.sobigdata.eu), by the NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics" – Prot. IR0000013 – Avviso n. 3264 del 28/12/2021, by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU – PNRR, by the Italian Ministry of University and Research "Progetti di Rilevante Interesse Nazionale" project: "Multicriteria data structures and algorithms" (grant n. 2017WR7SHH).

ABSTRACT We investigate the potential of several artificial neural network architectures to be used as an index on a sorted set of strings, namely, as a mapping from a query string to (an estimate of) its lexicographic rank in the set, which allows solving some interesting string-search operations such as range and prefix searches. Our evaluation on a variety of real and synthetic datasets shows that learned models can beat the space vs error trade-off of the classic (possibly compressed) trie-based solutions for relatively dense datasets only, while being slower to be trained and queried. This leads us to conclude that learned models are not yet competitive with classic trie-based solutions, and thus cannot completely replace them, but possibly only integrate them. Although our study does not settle the question conclusively, it highlights appropriate methods, provides a baseline for comparison, and introduces several open problems, thereby serving as a starting point for future research.

INDEX TERMS String dictionaries, string search, prefix search, tries, neural networks, machine learning, data structures, learned indexes.

I. INTRODUCTION

We deal with the problem of indexing a sorted set $\mathcal{S}$ of n strings $s_1, s_2, \ldots, s_n$ drawn from an alphabet Σ, with $|\Sigma| = \sigma$, in order to efficiently answer the query rank(q), whose output is the number of strings in $\mathcal{S}$ which are lexicographically smaller than or equal to a given query string q. Formally, rank(q) $= |\{s \in \mathcal{S} \mid s \leq q\}|$.

The rank query is powerful enough to allow solving several other operations on $\mathcal{S}$ such as the *lookup*, which determines whether a given string exists in $\mathcal{S}$, the *prefix search*, which finds all the strings in $\mathcal{S}$ having a given pattern as a prefix, and the *range search*, which finds all the strings in $\mathcal{S}$ that fall in a given query range.[1] Given this, it comes as no surprise that the problem of supporting rank arises frequently in applications, including, to name but a few, databases (e.g. to implement range queries [1]), bioinformatics tools (to find the rank or to count k-mers [2]), and search engines (to enable query autocompletion [3] or to index the vocabulary [4]).

The classic data structure to solve this problem is the trie [5], [6], a multiway tree that stores each string $s \in \mathcal{S}$ as a root-to-leaf path, and whose branches are labelled with a character from Σ (see Section II). Since its inception in the sixties, the trie has undergone many significant developments that improved its query or space efficiency such as

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

[1]A lookup amounts to compute $i = \text{rank}(q)$ and check whether $s_i = q$. A prefix search is given by the strings $\{s_i \in \mathcal{S} \mid \text{rank}(q) \leq i \leq \text{rank}(q\#)\}$, where # is a character larger than all the characters in Σ. A range search with endpoints l and r is given by the strings $\{s_i \in \mathcal{S} \mid \text{rank}(l) \leq i \leq \text{rank}(r)\}$.

compacting its paths [7] or whole subtries [8], [9], [10], [11], using adaptive representations for its nodes [12], [13], [14], succinct representations of its topology [1], [15], and cache-aware or disk-based layouts [16], [17].

A recent and quite different trend to solve the problem of indexing $\mathcal{S}$ consists instead in learning an approximation of rank via a model f, typically trained on a dataset of input-target pairs $(s, \text{rank}(s))$ for $s \in \mathcal{S}$, and then using the output $f(q)$ of this model as an estimate for the rank of string q [18]. Clearly, the model might incur in an error, measured as $|f(q) - \text{rank}(q)|$, but as long as $\mathcal{S}$ is stored somewhere, the correct rank of q can be determined by comparing q with a range of strings $\ldots, s_{p-1}, s_p, s_{p+1}, \ldots$ around the predicted position $p = f(q)$. Since such comparisons can be computationally expensive, one should aim at keeping the range size small by training a model f with a low error, which, on the other hand, often entails increasing the model complexity of f, thus possibly increasing the time to compute a prediction and decreasing the space efficiency due to the storage of additional model parameters. As a result, the design and training of f should carefully counterbalance both the model complexity and the model errors.

As of today, the learned indexing approach has mostly been successfully applied to integers (i.e. short strings of 4 or 8 bytes) such as in [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], and [29], possibly stored in compressed form [18], [30], [31] (see [32] for an introductory survey). Overall, these studies have shown that learned indexes can be as time efficient as classic indexes while being orders of magnitude more space efficient, because they can exploit new patterns and regularities in the data that classic solutions are unable to detect.

For the case of variable-length strings, on the other hand, the literature is scarce and the experimental results that so far have been obtained [19], [33], [34], detailed in Section II, show a general difficulty in beating the space-time performance of classic solutions. Furthermore, unlike for other indexing problems [35], [36], no study has systematically evaluated the impact of different model architectures (especially nonlinear ones) to be used in learned string indexes.

To fill this gap, we take a step back from the design and implementation of learned string indexes and focus on the more fundamental question of whether, and in which cases, learned models can provide accurate enough estimations for the task of indexing a string set under a given space budget for the model parameters. As a first attempt to answer this very difficult question, we conducted experiments: 1) on various nonlinear model architectures, including a new one, named Stair Multi-Layer Perceptron, that we have specifically tailored for the task at hand; 2) on datasets from four diverse domains and applications (natural language, geographic information systems, search engines, and genome sequences); and 3) by using a very succinct representation of tries as a non-learned baseline.

Our results show that learned models can beat the space vs error trade-off of the classic trie-based solutions for somewhat dense datasets only, while being slower to be trained and queried. This leads us to conclude that learned models are not yet competitive with classic solutions, and thus cannot completely replace them, but possibly only integrate them.

Although our study does not settle the question conclusively, it highlights appropriate methods, provides a baseline for comparison, and introduces several open problems, thereby serving as a starting point for future research.

To summarise our contributions in points:

- We review the classic approaches to deal with the problem at hand, followed by a critical discussion of the recently-proposed learning-based ones, highlighting the importance of using a succinct trie implementation as a baseline for comparison, as we do in our study.
- We consider a variety of existing nonlinear model architectures, including multi-layer perceptrons, convolutional neural networks, and (bidirectional) long short-term memory networks.
- We design a new model architecture, the Stair Multi-Layer Perceptron, that achieves a space vs error performance on par with or better than the above neural network approaches by leveraging the specificity of the problem.
- We identify four string datasets with widely different characteristics and introduce a synthetic dataset generator to provide a broad picture of the performance of the experimented approaches.
- We propose six open questions to encourage further research on the problem.
- We release our source code, trained models, and datasets publicly at https://github.com/giosumarin/ANN-string-indexes.

A. PAPER OUTLINE

Section II provides some background and discusses related work. Section III describes the methods of our study, datasets, and experimental setting. Section IV presents the results. Section V summarises our findings. Section VI proposes the open questions.

II. BACKGROUND AND RELATED WORK

A trie [5], [6] is a multiway tree that stores each string $s \in \mathcal{S}$ as a root-to-leaf path, and whose branches are labelled with characters from Σ. If a string in $\mathcal{S}$ is a prefix of another, we can make sure that it does not end at an internal node by appending a special character \$ smaller than all other characters in Σ, as depicted in Figure 1. A traversal of the trie guided by the characters in a query string q allows determining the lexicographic position of q among the strings in $\mathcal{S}$, from which the result of rank can be easily derived (e.g. by storing the rank information at the n leaves).

The indexing scenario, we consider here, aims at storing $\mathcal{S}$ somewhere (e.g. on a disk, possibly in compressed form) and

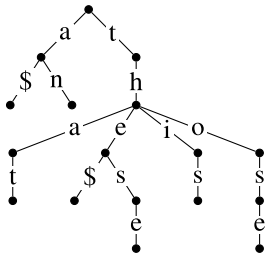

FIGURE 1. A trie on the set $\mathcal{S} = \{$a, an, that, the, these, this, those$\}$.

building a small index data structure (e.g. small enough to fit in main memory or in GPU memory) on $\mathcal{S}$ to enable efficient rank queries. In this scenario, a trie $\mathcal{T}$ can be built on the subset $\mathcal{S}'$ of $\mathcal{S}$ given by the strings at positions multiples of a given sampling rate r. Then, a rank(q) query can be answered in two steps. First, we determine the rank of q among the strings in $\mathcal{S}'$ via a traversal of $\mathcal{T}$; say this rank is i. Second, we determine the answer to the rank(q) query by comparing q with the strings $s_{ir}, \ldots, s_{(i+1)r-1}$. Clearly, the sampling rate r allows us to trade the space needed by the index $\mathcal{T}$ with the number of string comparisons done in the second step. Note also that, to save space, the strings in $\mathcal{S}'$ can be truncated to their distinguishing prefixes, i.e. the smallest prefix distinguishing a string from the others in $\mathcal{S}'$, so that the resulting $\mathcal{T}$ has no (unary) paths with a single descendant leaf (such as the path "ose" in the rightmost path of the trie in Figure 1).

As mentioned in the introduction, the trie has undergone many significant developments since its inception. For the purpose of this paper, we now recall one recent and representative example of these developments, named Fast Succinct Trie (FST) [1], [37], that we will use in our experimental evaluation as a non-learned baseline. In FST, the trie is not encoded via space-consuming pointers but via two succinct representations, named LOUDS-Sparse and LOUDS-Dense. The former representation concatenates the branching characters in level-wise left-to-right-order into a vector of characters, and employs two bitvectors of the same size: the first bitvector marks whether the branch points to an internal node or to a leaf, the second marks whether a branching character is the first one of a node. For example, the first two levels of Figure 1 are represented in the FST by the vector of characters `at$nh`, and the bitvectors 11001 and 10101, respectively. In LOUDS-Dense, instead, a single node is represented using three components: a bitmap of size 256 that has the ith bit set if the node has a branch with label i (thus, FST assumes an alphabet over bytes), a bitmap of size 256 that marks whether a branch points to an internal node or to a leaf, and a bit that indicates whether the prefix that leads to the node is a string in $\mathcal{S}$. The three components of the LOUDS-Dense encoding of the various nodes in the trie, again visited in level-wise left-to-right-order, are concatenated into three separate vectors. Both LOUDS-Dense and LOUDS-Sparse use the so-called rank-select primitives on bitvectors [15] to allow efficient trie navigation. FST employs (the more query-efficient) LOUDS-Dense for the top levels of the trie and (the more space-efficient) LOUDS-Sparse for the bottom levels, where the dividing point is determined by guaranteeing that LOUDS-Dense takes less than 2% of the trie space, while still covering the frequently-accessed top levels.

For what concerns the learned indexing approaches for strings, the results available in the literature are, to the best of our knowledge, RMI [19], RSS [33], and SIndex [34].

RMI [19] truncates the strings to a fixed length and uses their ASCII representation as feature vectors, which are then fed to a hierarchy of learned models (implemented as neural networks with 1 or 2 hidden layers) to predict the rank. The models in the hierarchy that have a high maximum error are replaced with B-trees [38]. An evaluation on document-ids from a web index shows modest improvements or no improvement in time with respect to standard B-tree indexes, but up to 2.7× less space. However, B-tree indexes are too space-inefficient for string keys because they do not take advantage of the common string prefixes [17], which is why we use succinct tries as our non-learned baseline. Also, the RMI implementation for strings is not open source.

RSS [33] is a trie in which each node uses 8 (or 16) bytes for branching. If these 8 bytes are insufficient to determine the rank of an input string within a prescribed maximum error, then it builds another child node with the descending strings having that 8-byte prefix, otherwise it feeds these 8 bytes (that fit into a computer word) to an integer-based linear learned index that predicts the rank of the string. The evaluation shows that the lookup performance of RSS is worse than a trie-based index (HOT [39]), but one or two orders of magnitude more compressed. However, the use of succinct tries (which we found to be 3.2–8.3× smaller than HOT on the full datasets of Table 1) and the space-time trade-off given by varying the sampling rate have not been evaluated, and we cannot in turn do it because the RSS implementation is not open source.

SIndex [34] greedily partitions the sorted input strings into groups so that: 1) the range of characters that must be examined to determine the lexicographic position of a string within a group is below a given threshold (where the range is given by the minimum lcp and the maximum lcp of the strings in the group); and 2) a linear model built to predict the rank of the strings in the group has a mean error below a given threshold. Then, it creates a root node, implemented as a piecewise linear regression model, on the set of group pivots to route the queried string to the correct group. The evaluation shows that, on a dataset of URLs, SIndex has a lower query performance than a classic index (Wormhole [40], which is a hybrid of hash tables, tries, and B-trees) but better performance on randomly generated strings. The space usage has not been evaluated in the paper.

However, since the implementation of SIndex is open source, we ran it on the dataset of Table 1 and measured its space conservatively by considering just the space of the

pivots and of the linear model's parameters, and ignoring other data structure metadata and buffers to handle insertions. On average it took 263.8× more space than the corresponding (i.e. same mean error) FST, so we do not consider it further.

III. METHODS

As mentioned in Section I, to solve our string indexing problem we aim to learn an approximation of rank: $\Sigma^* \to \{0, \ldots, n\}$ via a model f trained on the dataset of input-target pairs $(s, \text{rank}(s))$ for $s \in \mathcal{S}$.

As a technicality, we will actually facilitate the training of our model, implemented as an Artificial Neural Network (ANN), by scaling the targets to the range of reals $[0, 1]$ and scaling back the output of the model to the range of integers $\{0, \ldots, n\}$.

We evaluate f in terms of space and error. The space is given by the storage of the model's parameters in memory, without compression, which in turn depends on the chosen model architecture and hyperparameters, and thus it is fixed (details below). The error is measured in terms of *mean (absolute) error*

$$\varepsilon = \frac{1}{n} \sum_{s \in \mathcal{S}} |\text{rank}(s) - f(s)|, \tag{1}$$

and *maximum (absolute) error*

$$\varepsilon_{\max} = \max_{s \in \mathcal{S}} |\text{rank}(s) - f(s)|. \tag{2}$$

We ought to observe that, differently from a classic machine learning task, the model f should overfit as much as possible the input dataset to provide effective indexing (i.e. to be close to the target function rank), which is why we do not split $\mathcal{S}$ into training, validation and test set, but we use it in its entirety to train and evaluate f [19], [32].

Note also that, in some applications, the query strings do not necessarily belong to $\mathcal{S}$ but to its superset Σ^*, thus giving rise to other definitions of ε and $\varepsilon_{\max}$ that take into account the error on these strings too. We discuss this issue in Section VI.

We compare the space and error of the various ANN architectures with the ones incurred by FST [1], built on a sample of the input dataset given by the strings at positions multiples of a given sampling rate r (see also Section II). In this setting, the value r also corresponds to the *maximum error incurred by FST*, because a search in such an FST can determine the rank of a query string q as one of the sampling positions $0, r, 2r, 3r, \ldots, \lfloor n/r \rfloor r, n$, and thus the determined rank would be far from the correct $\text{rank}(q)$ by at most r. On the other hand, if we assume queries are generated by a random process such that the answers follow a uniform distribution over the range $\{0, \ldots, n\}$, then the *mean error incurred by FST* can be defined as $r/2$, which is the value we report in our figures. Finally, for what concerns the space occupied by the FST, as motivated in Section II, we do not consider the space for storing the string suffixes past the distinguishing prefix.

In the following, we discuss datasets (Section III-A), how we encode the inputs to the ANNs (Section III-B), which ANN architectures we use (Section III-C), and the experimental setting (Section III-D).

TABLE 1. Characteristics of the real datasets.

Dataset	n	Avg. length	Avg. LCP	Alphabet size
WORDS	895 612	8.0	5.7	26
GEO	7 240 280	13.1	8.1	91
URL	988 299	19.7	15.1	93
DNA	13 745 061	12.0	10.6	15

A. DATASETS

To best evaluate our methods on different kinds of string distributions, we use four real datasets coming from different domains and applications (natural language, geographic information systems, search engines, and genome sequences):

1) WORDS contains natural language words derived from the Google Books 1-gram dataset.[2]
2) GEO contains the names of geographic locations throughout the world.[3]
3) URL contains URLs from a Web crawl of the .uk domain.[4]
4) DNA consists of all unique 12-mer in a DNA sequence.[5]

To keep the training times and the space of the ANN models low, we truncate the strings to a length of 20 in all the above datasets except DNA.[6] While this might be seen as a limitation, actually, it will allow us to factor out the length of the string when we compare our methods on different datasets. Furthermore, we notice that longer strings could be handled recursively by building a trie-like data structure where the nodes correspond to the models (see Section II).

The features of our datasets after this preprocessing are shown in Table 1.

Other than real datasets, we introduce a synthetic dataset generator that creates length-L strings over an alphabet of a given size σ, whose corresponding trie is full in the first P levels and sparsely populated in the remaining levels. Specifically, we first generate all the σ^P possible length-P strings from that alphabet of size σ, and we complete each of these strings to a length L by appending $L - P$ randomly-chosen characters. Then, given a density factor $D \in [0, 1)$, we further add $\lceil D\sigma^L \rceil$ strings composed of L randomly-chosen characters. We will vary L, σ, and D, while keeping $P = \lfloor L/2 \rfloor$.

B. ANN INPUT CODING

Two approaches to feed the ANN with strings $\mathcal{S}$ are proposed: the classical *one hot* and *binary* encodings. They differ

[2] https://storage.googleapis.com/books/ngrams/books/datasetsv3.html
[3] http://download.geonames.org/export/
[4] http://data.law.di.unimi.it/webdata/uk-2002
[5] http://pizzachili.dcc.uchile.cl/
[6] We also remove common URL prefixes of the kind "http(s)://(www.)".

from each other in the way they encode individual characters in Σ.

- *One hot encoding* (ohe). This is a typical way to transform a character into a numerical binary format. Each symbol $a_i \in \Sigma$ is transformed into a σ-dimensional binary vector $v_{a_i}^{\text{ohe}}$ made up of all zeros except for a 1 in position i.
- *Binary encoding* (bin). Each symbol $a_i \in \Sigma$ is converted into the binary representation $v_{a_i}^{\text{bin}}$ of the integer i. The length of $v_{a_i}^{\text{bin}}$ is $\lceil \log \sigma \rceil$.

Each string $s \in \Sigma^*$ then is encoded as the binary string obtained by concatenating the encodings of its characters $s_1, \ldots, s_{|s|}$, and denoted by $C_X(s) = v_{s_1}^X \cdot v_{s_2}^X \cdot \ldots \cdot v_{s_{|s|}}^X$, where $X \in \{\text{ohe}, \text{bin}\}$, and $\cdot$ is the concatenation operator. When it is clear from the context, we omit to specify the encoding type X in this definition.

Strings are then truncated to m, the maximum longest common prefix (LCP) between any two strings of $\mathcal{S}$, augmented by 1, since by definition of LCP their ranking is fully determined by their first m characters. In order to have inputs of fixed length, necessary to build an ANN model, padding with a neutral symbol is performed to achieve the length m for strings shorter than m.

C. ANN MODELS

The ANN models selected here have an input of size $m\sigma$ when strings (of length m) undergo ohe encoding and of size $m\lceil \log \sigma \rceil$ in the case of bin encoding. We considered a variety of ANN models, covering recurrent, convolutional and feed-forward neural networks, having as a central goal that of yielding an effective but also succinct model. The latter poses clear constraints to the configuration of the model architecture. Indeed, we excluded from our analysis ANN architectures that have an intensive memory usage, like Transformers and their extensions [41], [42], [43].

- Multi-Layer Perceptron (MLP). A multi-layer perceptron is the most known and most frequently used type of feed-forward neural network, composed of an input layer, a last single-unit output layer (in our setting), and at least one hidden layer between input and output layers [44]. The input layer is fed with the binary string $C(s)$ (Section III-B), where each bit corresponds to a dedicated input neuron. To infer a prediction in the interval $[0, 1]$, a sigmoid activation function is used for the output unit (Figure 2 (a)).
- Convolutional Neural Networks (CNNs). CNNs are a kind of feed-forward neural network able to extract features from input data with convolution structures exploiting spatial relationships [46]. In particular, 1D CNNs are suitable to process 1D data, e.g. data represented through vectors, and are effective when extracting features from a fixed-length segment. This is the case, for instance, of signal identification [47]. Multiple convolutional layers are commonly placed in sequence, each able to extract different features at different levels of abstraction. In our setting, such an architecture is suitable because each input character is encoded by a fixed-length vector, whose components are spatially related. We designed a convolutional block sliding over adjacent characters, having three 1D convolutional layers followed by one or more fully-connected (FC) layers to perform classification. The last layer is a sigmoid output unit (Figure 2 (b)), like the previous models described in this section.
- Long short-term memory (LSTM). Recurrent neural networks, and in particular LSTMs [48], have been investigated intensively in recent years in text classification due to their ability to model long-range dependencies [49]. Given a string $s \in \mathcal{S}$, individual coded characters v_{s_i} are fed to a dedicated LSTM cell at each timestep, for a total of m timesteps. Two LSTM layers are stacked atop each other (depth in RNNs helps to capture temporal hierarchies [50]). Finally, a single unit FC layer (if necessary preceded by other FC hidden layers) is applied on the output of the rightmost cell (Figure 3 (a)). Moreover, analogously to [51], we consider two model variants: *LSTM-multi*, which includes explicit dependence among all characters by concatenating the outputs of all LSTM cells in the last layer into an additional FC layer, since typically considering just the last LSTM cell outputs suffers from the vanishing gradients issue (Figure 3 (b)); *BiLSTM*, a bidirectional LSTM where each LSTM layer is duplicated to read the input string even in reversed order. Gated Recurrent Unit (GRU), a variant of LSTMs, is discarded here due to its lower performance in this experiment, confirming the results shown in [49].
- *Stair* MLP (SMLP). This an architecture we tailored leveraging the specificity of the problem. In particular, we observe that the rank of a string mainly depends on its leftmost characters. Our idea is to consider a multi-input model where each character is supplied to the network at a separate level, starting from the rightmost character to the leftmost one. This counter-intuitive order represents an explicit way to foster the network to pay more attention to the leftmost characters during the training. The last coded character v_{s_m} is fed to an FC layer, whose output, concatenated to the next (to the left) coded character $v_{s_{m-1}}$, becomes the input for a second FC layer. Then a third FC layer receives the output of this layer and the third last coded character $v_{s_{m-2}}$, and so on (Figure 4 (a)). The leftmost coded character v_{s_1} is concatenated to the output of the FC layer associated with the previous character (on the right) and fed to its FC layer, whose output undergoes the classification block made up, even in this case, by a sigmoid unit preceded by zero or more FC hidden layers (Figure 4 (b)).
  Another benefit of this stair multi-input architecture is the possibility to assign a different number of hidden units to individual characters, so as to dedicate more space/parameter resources to the most relevant

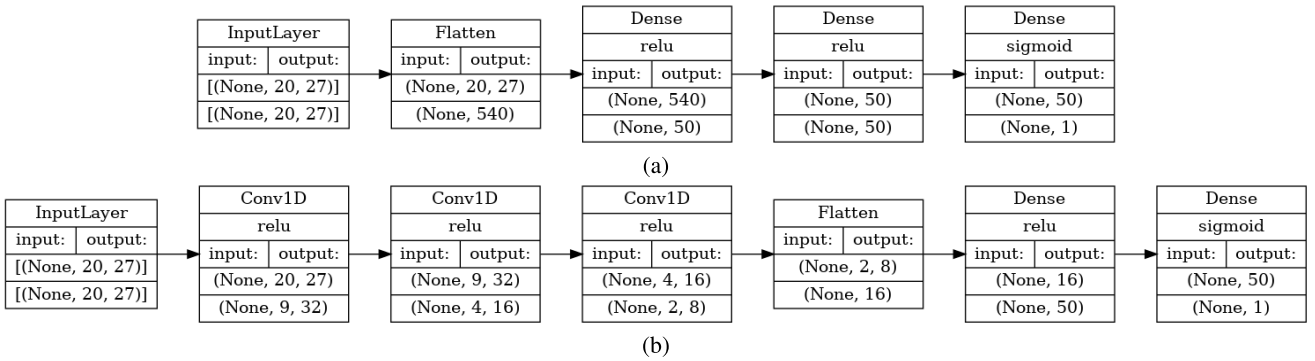

FIGURE 2. Example of MLP (a) and CNN (b) architectures to classify strings in the dataset *az-words*, when characters are encoded via one-hot encoding. Strings are fed to both models (*InputLayer*) as a binary matrix of shape (None, 20, 27), whose components refer to, respectively, the variable training batch size, the length of the string, and the alphabet size plus one (because of the added null character). For MLP, the input is effectively the binary string obtained by concatenating (*Flatten* layer) the encoding of the corresponding characters, while CNN *Conv1D* operates a convolution on entire characters (thus, no flattening is needed). Each box/layer shows also its output shape (last row in the box), in addition to the activation function (second row), when applicable. Both architectures are obtained from the Keras API [45].

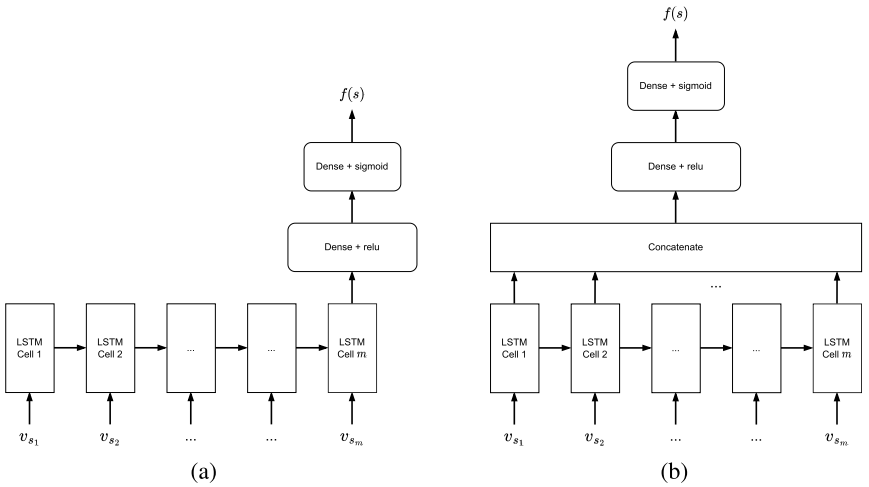

FIGURE 3. Example of LSTM (a) and LSTM-multi (b) architectures to rank string $s \in \mathcal{S}$ of length m, where v_{s_i} is the encoding of the ith character of s.

characters for determining the string rank, also preventing the model size to explode. Indeed, in the MLP model, all characters are assigned to the same number of hidden units, causing the size of the model to easily increase when just a few hidden units are added. Clearly, the possible configurations of hidden units across characters are too many. Accordingly, we adopt a simple scheme to assign hidden units that only needs to fix the number of hidden units b parsing the leftmost character, and the number of units d to be reduced for the next character (on the right). Thus, the ith character from the left will be parsed by an FC layer having $\max\{b-(i-1)d, 1\}$ units. We name such a model $\text{SMLP}_{b,d,h}$ for short, assuming a unique h-unit hidden layer is added before the output layer.

Finally, to help prevent vanishing gradient problems, we employ ReLU activation function for all FC layers but the output one.

It is worth noting that the architecture of the SMLP model is the result of a substantial design and modelling effort, which investigated many different DNN architectures and their variants, which either performed worse or did not introduce any advantage w.r.t. SMLP. Among them the most relevant are the following ones: 1) a reversed SMLP, where characters are read in the "usual" order, i.e. from left to right (actually, this variant performs very poorly); 2) a multi-input architecture in which, like SMLP, all characters are parsed by individual FC blocks, but (unlike SMLP) all inputs are at the same level; 3) a stair multi-input model, like SMLP, but where each character is parsed by an entire MLP with one output neuron; 4) a variant of 3, where skip connections convey ahead the output MLPs associated with individual characters, namely to a character placed at a fixed number of positions forward—like for SMLP, forward means from right to left in the string; and 5) a combined model, where the whole input is parsed simultaneously by a LSTM block and by a CNN block, whose

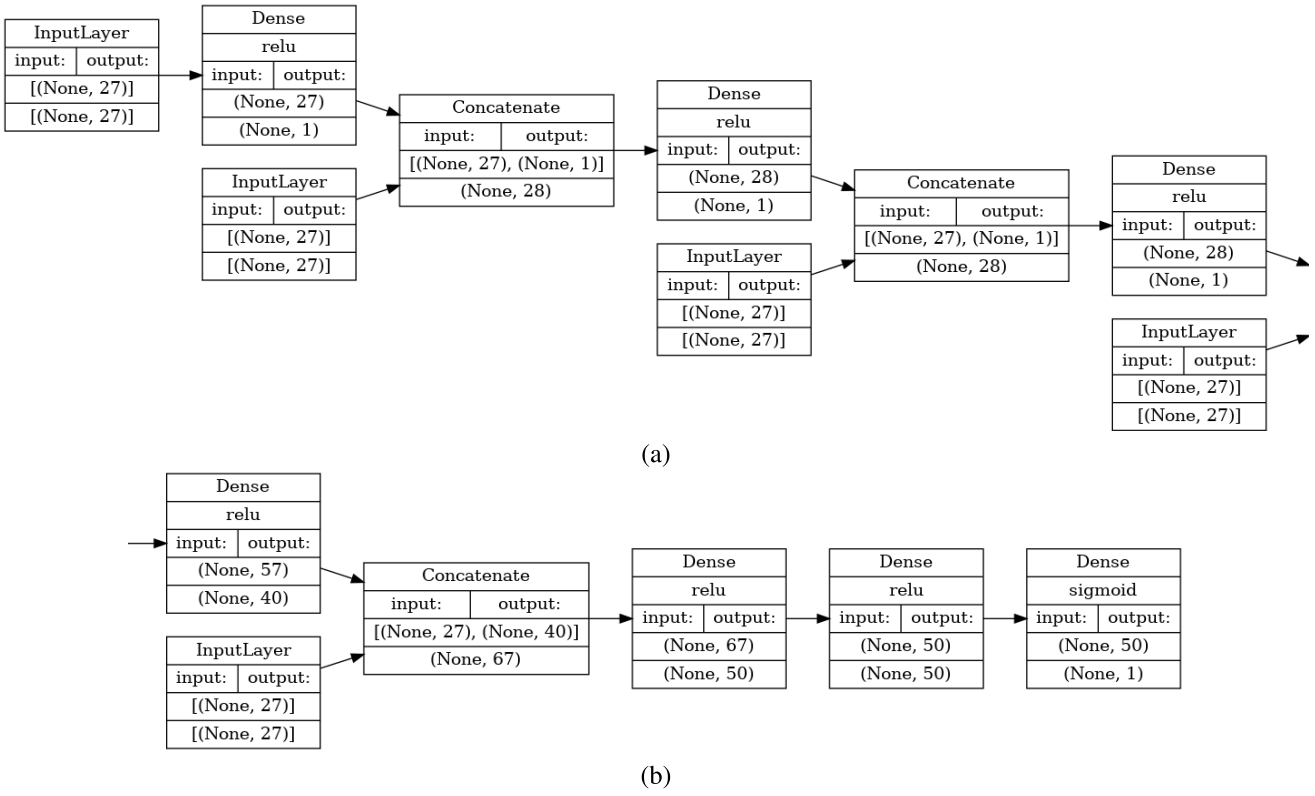

FIGURE 4. **SMLP$_{50,10,50}$ model for *az-words* strings. Beginning of the model (a) and its last layers (b) Box/layer formats are the same as in Figure 2.**

TABLE 2. **Specifications for the models. We use the following notation: hidden units for FC layers (h), kernel size (k), stride (s), and hidden state size (u) for LSTM cells.**

Model	Specifications
MLP	h: $(50, 50)$
LSTM	u: $(20, 20)$, h: 50
LSTM-multi	u: $(20, 20)$, h: 50
BiLSTM	u: $(20, 20)$, h: 50
CNN	(k, s): $(4, 1)$, $(2, 1)$, $(2, 1)$, h: 50

outputs are concatenated and fed to a MLP block. LSTM and CNN modules are configured as described in Section III-D.

Furthermore, we also implemented a variant of the training procedure, decomposing the problem into m sub-problems, each considering only the prefixes of a given length $l \in \{1, \ldots, m\}$ of the strings in $\mathcal{S}$. The rationale is verifying whether the model could learn more efficiently simpler problems, and at the same time to have a significant initialization of FC layer weights. This can be verified by training one character at a time (via the corresponding sub-problem), while clamping or just slightly refining the blocks already trained and relative to shorter string prefixes. We discarded such an approach because it was much more computationally intensive, while leading to models with similar predictive capabilities.

D. EXPERIMENTAL SETTING

The specifications of each layer in each architecture are presented in Table 2. Since in this evaluation errors are measured on the strings in $\mathcal{S}$, this set is used for the training, and we aim to overfit it, in order to minimize the average error. To this end, we run multiple incremental learning steps until convergence for the Adam optimizer [52], with gradually reduced learning rates, starting from 5E-04 to 1E-06. Furthermore, to attempt reducing the maximum error, an enriched variant of the training has also been tested, in which the strings with the highest error after each training step (i.e., after the training with a given learning rate until convergence) are replicated multiple times. This is equivalent to using different sample weights during training.

More in general, all the m^σ strings in Σ^m could be used for training on a given data set $\mathcal{S} \subset \Sigma^m$, and then evaluate the model just on strings in $\mathcal{S}$. However, this would sensibly slow down the training. To avoid an excessive computational burden, we tested two variants of such an enrichment approach: the first one replicates top-0.05% error strings proportionally to their error; the second one instead replicates top-0.05% error strings a fixed number of times (identical for all strings). Experimentally, we decided to retain the latter, since it has fewer hyperparameters and leads to similar results. The percentage of strings to be replicated after each run has been experimentally tuned, and increasing it tends to deteriorate

the average error ε, which is a behaviour to prevent in this setting.

The configuration of the remaining hyperparameters (loss function, batch size, patience, etc.) for all models has been decided based on a small toy data set. The *mean absolute error* (MAE) loss resulted in a lower average error with regard to mean squared error (MSE) loss, which in turn tends to yield a lower maximum error (this is expected since larger errors tend to have a larger impact on the gradients). For the $\text{SMLP}_{b,d,h}$ model, proposed here, different pairs (b, d) have been tested in order to better investigate its behaviour. The pairs are selected with the rationale of yielding models from less to more succinct while attempting to ensure parsing with more than one hidden unit the most significant characters, roughly until the average LCP of the current dataset (Table 1). To ensure a fair comparison, the dimension of the final FC classification block is set as for the other compared models ($h = 50$).

IV. RESULTS

A. REAL DATA

Our results on the four real datasets show that, although the ANNs are capable to solve the problem with relatively low errors, FST is more efficient in three out of four datasets (Figure 5). On GEO and URL data the gap in favour of FST is larger, and we conjecture that it might depend on two related factors, namely the *alphabet size* σ and the *string density*, intended as the ratio n/σ^m. Indeed, their σ is much larger than that of WORDS and DNA data, and this in turn induces much lower string density (4.77E-33 for GEO and 4.21E-34 for URL), and larger one-hot encoding dimensions for the input characters in ANNs. As a consequence, although the error ε of ANNs is relatively small with respect to n on both GEO (around $200 \leq \varepsilon \leq 800$) and URL (around $35 \leq \varepsilon \leq 175$) data, these models cannot exploit the sparsity of the strings like FST does. This is coherent with the results on the remaining two datasets: on WORDS data, characterized by a lower σ and higher density (4.49E-23) w.r.t. GEO and URL data, ANNs obtain a space-error trade-off closer to that of FST; on DNA data, the densest dataset of those considered (density 1.06E-07), neural networks are instead much better than FST. Indeed, DNA data are even more dense (82%), if we consider just the 4 distinct characters ATGC, which make 99.98% of the data.[7] To corroborate this analysis, in Section IV-B we explore more in detail on synthetic data the behaviour of ANNs and FST with regard to string density, alphabet size and string length.

Concerning the results of ANNs, we can emphasize the following relevant points.

1) ENRICHING TRAINING DATA

Because of their reduced size, the impact of training data enrichment (see Section III-D) has been evaluated on WORDS and URL data, specifically by comparing the results of three SMLP models ($\text{SMLP}_{100,10,50}$, $\text{SMLP}_{50,5,50}$, $\text{SMLP}_{50,10,50}$) with and without replication of strings with the highest error (the latter denoted as *no enrich* in Figure 5). After tuning on a small subset of data, we decided to replicate each of these strings 20 times. We have observed (results not shown) that increasing this value more tends to increase the mean error ε (while obviously decreasing $\varepsilon_{\max}$, see Section VI). On WORDS data, both variants show similar results in terms of mean error ε, while on URL data the "enriched" variants are always more effective. For this reason, we decided to adopt the enriched variant to train all ANN models.

2) SMLP VS OTHER ANN MODELS

The results of SMLP model suggest that our idea of designing an architecture inherently aware of the character "relevance" can improve the capability in ranking strings.

In all datasets, classical MLP exhibits much worse space-error trade-offs than SMLP. On the other hand, LSTMs and BiLSTMs perform well on this task, confirming their known ability in handling NLP tasks: The former is close to SMLP performance, while the latter is worse than SMLP due to its larger size. Due to their high training time, we have tested CNN and LSTM-multi models only on WORDS data: the former achieves results in line with those of LSTMs and SMLPs, while the latter is less efficient than the baseline LSTM, mainly because of its excessive size. For the same reason, we have tested the binary encoding of characters only for $\text{SMLP}_{20,2,50}$ (denoted as *bin input* in Figure 5): it yields a notable reduction in size, which however is counterbalanced by lower effectiveness in ranking the strings, that makes the overall error/space compromise in line with that of one hot encoding.

3) SMLP BEHAVIOUR ANALYSIS

The configurations selected for SMLP aim thereby at investigating what is the most convenient strategy to distribute neurons (and, accordingly, model parameters). Indeed, unlike MLP, SMLP can assign a different number of neurons to process individual characters, thus potentially designing more complex blocks to parse the leftmost characters in the string (see Figure 4). We have experimentally verified that such a strategy pays more than equally distributing neurons across characters, and in Figure 5 we show just a representative of such a "flat" strategy on WORDS data, $\text{SMLP}_{20,0,50}$, which considers dense layers of the same width (20) to process any input character. While not improving the performance of $\text{SMLP}_{20,2,50}$ model, it has a bit/string ratio more than $3\times$ higher. This behaviour is visible even comparing $\text{SMLP}_{50,10,50}$ and $\text{SMLP}_{50,5,50}$ models: the former allocates most of the space budget to the first 5 leftmost characters, and results in a better space-error trade-off. An exception is represented by DNA data, where $\text{SMLP}_{50,10,50}$ is much more inefficient than all other ANN models. We believe this "unexpected" behaviour is due to the specific nature of DNA

[7] Given the presence of other symbols to denote sequencing errors, see e.g. https://en.wikipedia.org/wiki/Nucleic_acid_notation.

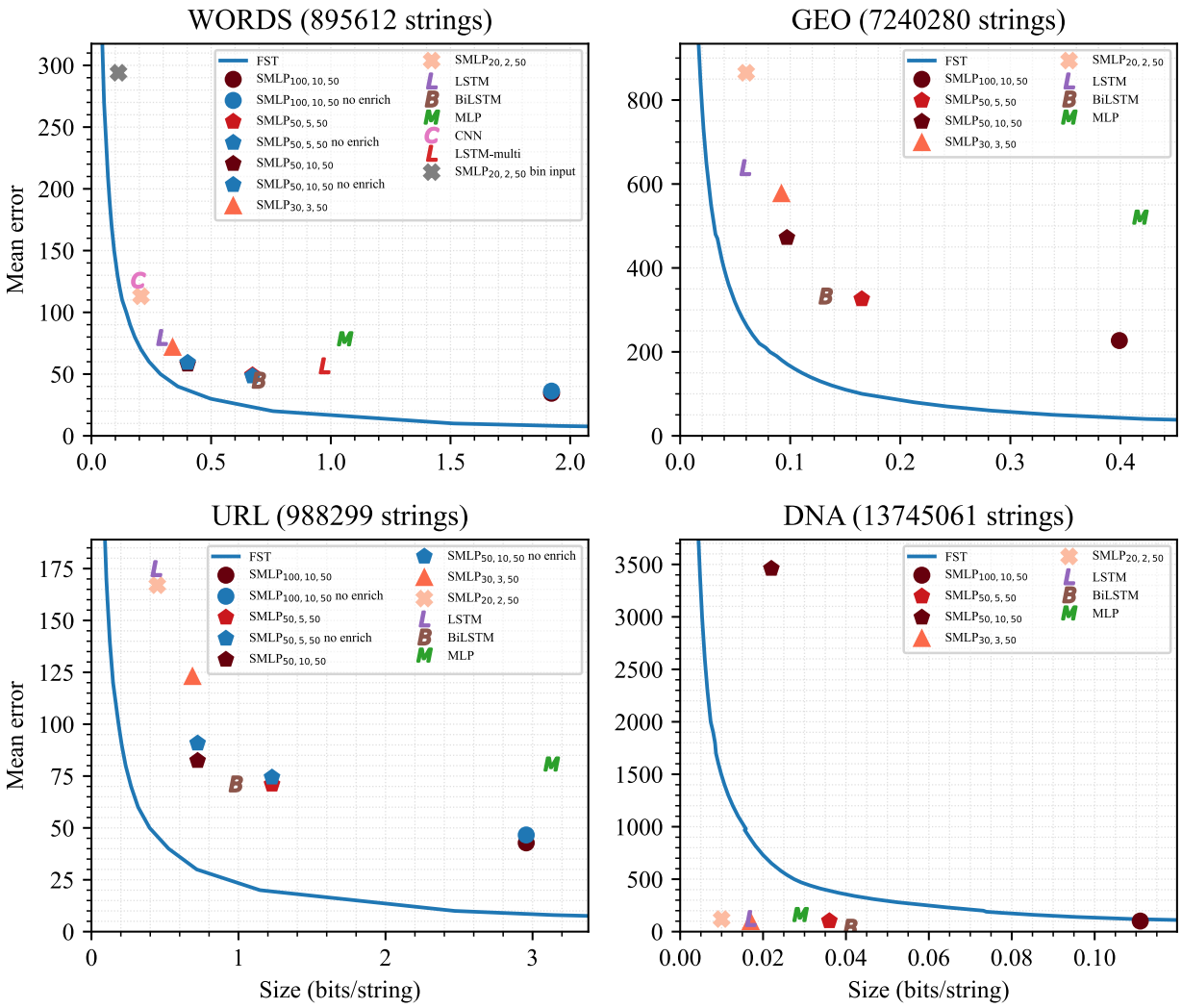

FIGURE 5. Results on real data.

data: unlike the other three datasets, it has an average LCP of almost 11 and thus close to the average (and maximum) string length of 12. This means that assigning blocks of adequate complexity just to the first 5 characters is a choice that does not allow the model to discriminate among a large subset of strings. Results of $SMLP_{20,2,50}$ and $SMLP_{30,3,50}$ models support such an interpretation: they have a much lower mean error and are more succinct than $SMLP_{50,10,50}$, but they "cover" with more than one neuron characters until the average LCP.

Summarizing, our tests suggest that we need to pay attention to data set characteristics to configure an SMLP model in relation to the available space/neuron budget, meanwhile configuring it so as to allocate as much computational power as possible to the leftmost characters.

4) QUERY TIME

We show in Table 3 the time taken by the various ANN models to compute a prediction on the batch of WORDS strings, averaged on 10 repetitions, both on a 1.80 GHz Intel Core i7-10510U CPU and an NVIDIA GeForce GTX 1650 Max-Q GPU. SMLPs perform similarly to LSTMs and CNNs in terms of space vs error trade-off, but with a significantly lower query time. Indeed, SMLP is around one order of magnitude faster than the former, and at least twice faster than the latter.

TABLE 3. Time (in seconds per query) taken by ANN models to query all WORDS strings.

Model	Time GPU	Time CPU
$SMLP_{100,10,50}$	1.08E-06	1.82E-06
$SMLP_{50,5,50}$	9.73E-07	1.38E-06
$SMLP_{50,10,50}$	9.25E-07	1.24E-06
$SMLP_{30,3,50}$	9.31E-07	1.25E-06
$SMLP_{20,2,50}$	9.16E-07	1.19E-06
$SMLP_{20,0,50}$	1.03E-06	1.48E-06
$SMLP_{20,2,50}$ bin input	3.68E-07	5.28E-07
LSTM	2.33E-06	1.55E-05
BiLSTM	3.80E-06	3.10E-05
LSTM-multi	2.38E-06	1.59E-05
MLP	1.16E-06	1.42E-06
CNN	1.68E-06	3.91E-06

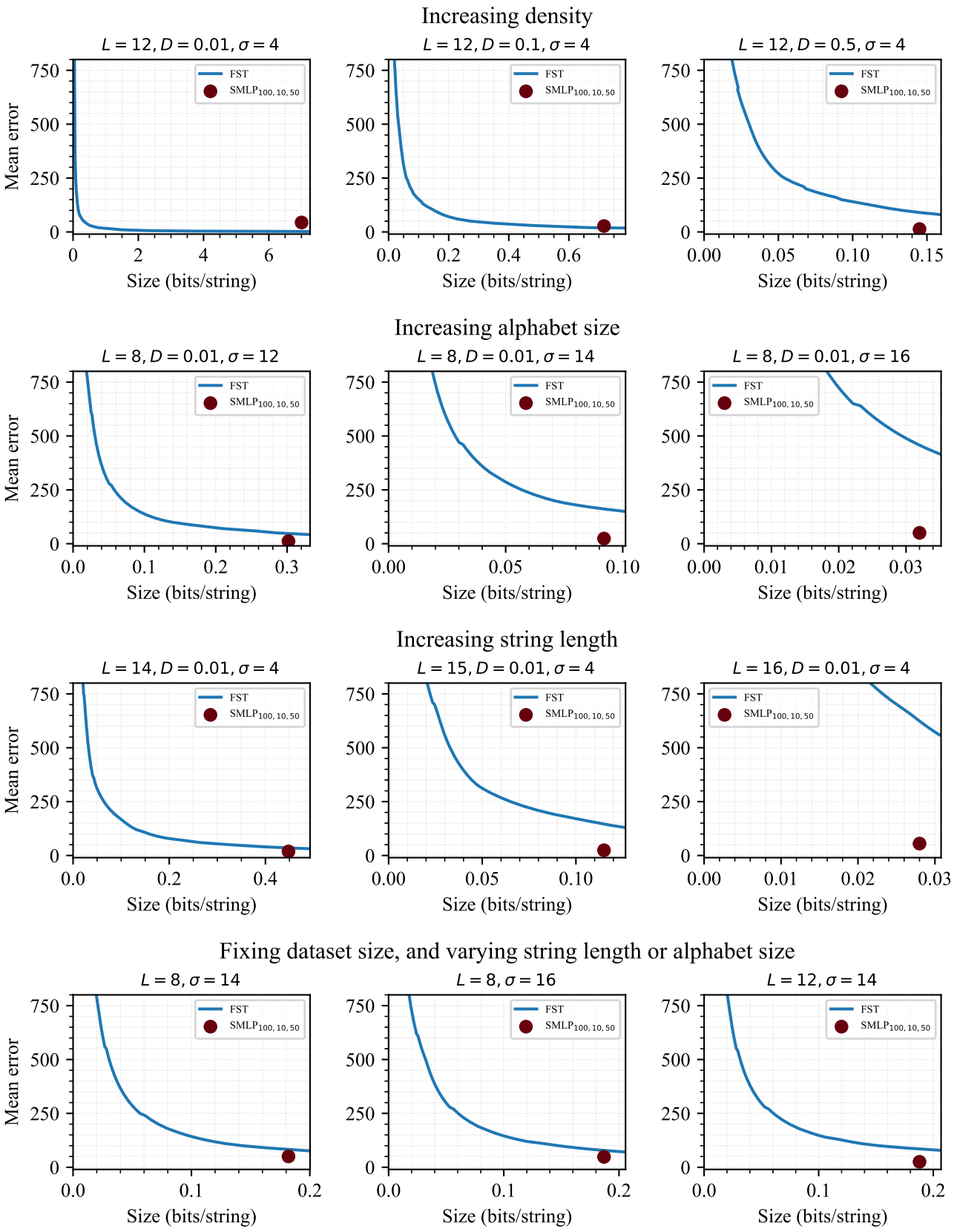

FIGURE 6. Results on synthetic datasets.

Concerning FST, which runs on CPU and is implemented in C++ (unlike ANNs, implemented in Python 3, using TensorFlow and Keras), the two configurations yielding an error similar to the most and least accurate ANNs ($SMLP_{100,10,50}$ and $SMLP_{20,2,50}$ - bin input) can be traversed on average in 2.80E-07 and 1.99E-07 seconds per query, respectively Therefore, compared to the above two ANNs run on CPU, FST is 6.5× and 6.0× faster, and compared to the above two ANNs run on GPU, FST is 3.9× and 4.6× faster.

B. SYNTHETIC DATA

This set of experiments is devoted to studying whether it is possible to detect some characteristics of the input dataset which can favour the effectiveness of ANNs over FSTs. In particular, we considered 3 groups of 3 synthetic datasets (generated according to the procedure detailed in Section III-A), each group fixing two out of three features among density (D), string length (L) and alphabet size (σ), while varying the remaining one, plus a fourth group, where instead we fix the total number of strings n and one of L and σ. The rationale of the latter is to assess whether potential trends found in the first three groups of experiments are due just to the dataset size (and accordingly to the bit/string ratio). In order to reduce the already high computational burden (some datasets contain tens of millions of strings), we only consider the $SMLP_{100,10,50}$ model as the ANNs representative.

The following behaviours can be identified: 1) SMLP's mean error ε reduces with the increase of string density (Table 4, rows 1–3), which is quite natural since the network can exploit a better approximation of the desired output; 2) when the string density is fixed, the mean error of SMLP reduces either when the string length or the alphabet size get smaller (rows 4–9); 3) when fixing the number of strings, but not the density, SMLP behaves not so in line with results emphasized at point 2), or even inverse when the string length rises (rows 10 and 12). Interpreting such results is not immediate, and some observed trends are likely to be partially induced by the randomness of both the training procedure of the model (weight initialization, sample permutation, etc.) and the synthetic data generation (random string sampling). Concerning the maximum error, we note that in general it is around two orders of magnitude larger than ε, and for a further discussion about it see Section VI.

On the other hand, we can get a more detailed and clear comparison between SMLP and FST by looking at Figure 6; in particular:

1) SMLP is favoured more than FST by an increase of string density (row 1 in Figure 6) but fixing L and σ;
2) the behaviour with regard to σ instead is fluctuating, since its increase favours SMLP when fixing L and D (row 2), whereas it is almost negligible when fixing L and n (row 4), but still in favour of SMLP for increasing L;
3) longer strings favour SMLP more than FST when σ and one between D (row 3) or n (row 4) are fixed. The latter trend might be related to the rise of the number of nodes in the FST that longer strings yield.

TABLE 4. Results of $SMLP_{100,10,50}$ model on synthetic datasets.

Dataset	n	ε_{max}	ε
$L = 12, D = 0.01, \sigma = 4$	1.71E+05	2.43E+03	4.39E+01
$L = 12, D = 0.1, \sigma = 4$	1.68E+06	6.66E+03	2.74E+01
$L = 12, D = 0.5, \sigma = 4$	8.39E+06	5.97E+02	1.35E+01
$L = 8, D = 0.01, \sigma = 12$	4.32E+06	6.75E+02	1.18E+01
$L = 8, D = 0.01, \sigma = 14$	1.48E+07	7.29E+02	2.39E+01
$L = 8, D = 0.01, \sigma = 16$	4.30E+07	9.02E+02	5.05E+01
$L = 14, D = 0.01, \sigma = 4$	2.70E+06	4.58E+03	1.87E+01
$L = 15, D = 0.01, \sigma = 4$	1.07E+07	5.74E+02	2.43E+01
$L = 16, D = 0.01, \sigma = 4$	4.30E+07	1.54E+03	5.52E+01
$L = 8, \sigma = 14$	8.00E+06	1.08E+03	5.04E+01
$L = 8, \sigma = 16$	8.00E+06	5.31E+02	4.81E+01
$L = 12, \sigma = 14$	8.00E+06	5.99E+02	2.53E+01

The results about the density variation (expected, as mentioned above) underline an interesting fact, that is, SMLP has only benefits from a rise of string density, unlike FST: indeed, SMLP size does not increase with the number of strings, while FST size does, and SMLP performance improves (Table 4). Moreover, these results provide a potential explanation of why SMLP outperforms FST just on DNA data, which indeed is the one having the highest string density among the tested real datasets.

V. CONCLUSION

We studied the potential of learned nonlinear models to represent, in a small space and with a low error, an approximate mapping from a query string to its lexicographic position in a sorted string set. This task, motivated by the recent achievements in learned indexing data structures for *integer* data [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], has received little attention in the literature [19], [33], [34], possibly because of its difficulty. The preliminary attempts in the literature, indeed, showed a general difficulty in beating classic (non-learned) solutions, even the space-inefficient ones.

To investigate these difficulties, we provided a systematic experimental analysis of several nonlinear model architectures (including a newly proposed one) on different kinds of datasets, and we compared the performance of these models against a strong classic solution, namely a succinct encoding of the classic trie data structure [1]. Our results show that learned models can beat the space vs (mean) error trade-off of the classic trie-based solutions for relatively dense datasets only, while requiring higher training and query time. This leads us to conclude that learned models are not yet competitive with classic solutions, and thus cannot completely replace them yet.

VI. OPEN QUESTIONS

While our evaluation provides new insights, it is not without limitations. In particular, we identified the following open questions to encourage future research:

- *Are ANNs competitive only for input datasets having a string density higher than a given threshold?* Our results show that the density of the string dataset is a key element for ANNs to outperform FST in terms of error/bit per string. We conjecture that it should be possible to determine, for any given pair (Σ, m) of alphabet and maximum string length, a density level $\bar{d} \in [0, 1]$ such that ANNs can beat a succinct trie in representing any set $\mathcal{S} \subset \Sigma^m$ whose density d is higher than $\bar{d}$. This conjecture is a direct consequence of the fact that in principle, fixed the maximum string length and the alphabet size, ANNs benefit twice from increasing the string density: their size does not increase, unlike what typically happens for FST, and they become more accurate (see for instance Figure 6, first row);
- *Can we design more effective ANN architectures for the problem?* Although in this study we examined dozens of ANN models, in our opinion much still needs to be done in this sense. The SMLP model, proposed here, resulted at least as efficient as LSTM and CNNs, while being more than twice faster than CNNs, and one order of magnitude faster than LSTM. However, its results can be considered sub-optimal. This suggests an interesting open problem: how to distribute the space/neuron budget across characters? For instance, one could partition the space budget so that a character position i receives a portion of the budget (in terms of model parameters) on the basis of the maximum error in which we might incur when classifying strings $\mathcal{S}$ by considering only the characters forward from position i. This is equivalent to consider the maximum number of strings in a sub-tree rooted at level i, when considering the trie of $\mathcal{S}$.
  A different, still interesting open problem is how to deploy ANNs compression, which we did not consider here, to further reduce their space (e.g., [53], [54]).
- *Can we lower the maximum error?* The maximum error has been considered only partially here, and there are some applications where instead it is relevant or even more important than the mean error. In such cases, the training of ANNs should take it into account, since we have experimented that the maximum error usually is at least two orders of magnitude larger than the mean error. For instance, one could adopt suitable loss functions (e.g., MSE instead of MAE), or implement data-enriching procedures to foster the model to improve its ability to classify the strings corresponding to the highest errors. Although we mainly focused on the mean error, we have also verified that a weak string enrichment can already help in this sense.
- *Are ANNs robust to query on strings that do not belong to the input set?* In this paper, we defined and measured the mean and maximum error of the model on the input set $\mathcal{S}$, cf. (1) and (2). In some applications, the model could be fed with strings that do not belong to $\mathcal{S}$ but to its superset Σ^*. If the model is not monotonic, the errors on these strings could be much larger than the errors computed on $\mathcal{S}$, and this weakness could be exploited by an adversary to create a batch of strings on which the model incurs very large errors (i.e. close to n) thus making the model useless for any indexing task. This highlights the need to investigate ways of analysing and fostering the robustness of the models to such inputs [55], [56], [57].
- *Do learned models have an advantage over classic trie-based solutions when the string set is updated?* We found that learned models, such as SMLP, require some hours to be trained, whilst the construction time of FST takes just a few seconds. On the other hand, inserting new strings in a succinctly-encoded trie requires complex and inefficient restructuring operations [58], while a learned model could update its weights more efficiently via incremental or transfer learning [59]. We leave for future work the study of these and other approaches to update a model and their comparison with classic trie-based solutions.
- *Can ANN compression methods improve the space-error trade-off of ANNs?* Several works recently proposed strategies to reduce the space demand of ANN models to the detriment of negligible or small accuracy drops (e.g. [60]). Such approaches, e.g., weight and neuron pruning, weight quantization, skeletonization, or knowledge distillation, can be investigated to assess how the gain in space efficiency compensates for the induced accuracy decay. Such methodologies well performed in classical holdout or cross-validation settings, but their effectiveness has not yet been proven in a setting, like ours, where we need to overfit data, and where even minimal changes in the model (architectural or simply in the learnt parameters) might induce large changes in the model prediction.

REFERENCES

[1] H. Zhang, H. Lim, V. Leis, D. G. Andersen, K. Keeton, and A. Pavlo, "Succinct range filters," *ACM SIGMOD Rec.*, vol. 48, no. 1, pp. 78–85, Nov. 2019.
[2] R. Chikhi, J. Holub, and P. Medvedev, "Data structures to represent a set of k-long DNA sequences," *ACM Comput. Surv.*, vol. 54, no. 1, pp. 1–22, Mar. 2021.
[3] U. Krishnan, A. Moffat, and J. Zobel, "A taxonomy of query auto completion modes," in *Proc. 22nd Australas. Document Comput. Symp.*, Dec. 2017, pp. 1–8.
[4] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surveys*, vol. 38, no. 2, p. 6, Jul. 2006.
[5] R. D. La Briandais, "File searching using variable length keys," in *Proc. 20th Western Joint Comput. Conf.*, 1959, pp. 295–298.
[6] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960.
[7] D. R. Morrison, "PATRICIA—Practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.
[8] T. Takagi, S. Inenaga, K. Sadakane, and H. Arimura, "Packed compact tries: A fast and efficient data structure for online string processing," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E100.A, no. 9, pp. 1785–1793, 2017.

[9] P. Bille, I. L. Gørtz, and F. R. Skjoldjensen, "Deterministic indexing for packed strings," in *Proc. 28th Annu. Symp. Combinat. Pattern Matching (CPM)*, vol. 78, 2017, p. 6:1–6:11.

[10] K. Tsuruta, D. Köppl, S. Kanda, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda, "C-trie++: A dynamic trie tailored for fast prefix searches," *Inf. Comput.*, vol. 285, May 2022, Art. no. 104794.

[11] A. Boffa, P. Ferragina, F. Tosoni, and G. Vinciguerra, "Compressed string dictionaries via data-aware subtrie compaction," in *Proc. 29th Int. Symp. String Process. Inf. Retr. (SPIRE)*, 2022, pp. 233–249.

[12] A. Acharya, H. Zhu, and K. Shen, "Adaptive algorithms for cache-efficient trie search," in *Proc. Int. Workshop Algorithm Eng. Experimentation (ALENEX)*, 1999, pp. 300–315.

[13] D. Baskins. (2002). *A 10-Minute Description of How Judy Arrays Work and Why They are so Fast*. [Online]. Available: http://judy.sourceforge.net/doc/10minutes.htm

[14] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 38–49.

[15] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. 30th Annu. Symp. Found. Comput. Sci.*, 1989, pp. 549–554.

[16] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter, "On searching compressed string collections cache-obliviously," in *Proc. 27th ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, Jun. 2008, pp. 181–190.

[17] P. Ferragina and R. Grossi, "The string B-tree: A new data structure for string search in external memory and its applications," *J. ACM*, vol. 46, no. 2, pp. 236–280, Mar. 1999.

[18] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 470–481, May 2011.

[19] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2018, pp. 489–504.

[20] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska, "ALEX: An updatable adaptive learned index," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 969–984.

[21] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, Apr. 2020.

[22] P. Ferragina, F. Lillo, and G. Vinciguerra, "On the performance of learned data structures," *Theor. Comput. Sci.*, vol. 871, pp. 107–120, Jun. 2021.

[23] M. Maltry and J. Dittrich, "A critical analysis of recursive model indexes," *Proc. VLDB Endowment*, vol. 15, no. 5, pp. 1079–1091, Jan. 2022.

[24] C. Ma, X. Yu, Y. Li, X. Meng, and A. Maoliniyazi, "FILM: A fully learned index for larger-than-memory databases," *Proc. VLDB Endowment*, vol. 16, no. 3, pp. 561–573, Nov. 2022.

[25] Z. Zhang, Z. Chu, P. Jin, Y. Luo, X. Xie, S. Wan, Y. Luo, X. Wu, P. Zou, C. Zheng, G. Wu, and A. Rudoff, "PLIN: A persistent learned index for non-volatile memory with high performance and instant recovery," *Proc. VLDB Endowment*, vol. 16, no. 2, pp. 243–255, Oct. 2022.

[26] X. Zhong, Y. Zhang, Y. Chen, C. Li, and C. Xing, "Learned index on GPU," in *Proc. IEEE 38th Int. Conf. Data Eng. Workshops (ICDEW)*, May 2022, pp. 117–122.

[27] D. Amato, R. Giancarlo, and G. L. Bosco, "Learned sorted table search and static indexes in small-space data models," *Data*, vol. 8, no. 3, p. 56, Mar. 2023.

[28] D. Amato, G. L. Bosco, and R. Giancarlo, "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. Exper.*, vol. 53, no. 2, pp. 318–346, Feb. 2023.

[29] S. Zeighami and C. Shahabi, "On distribution dependent sub-logarithmic query time of learned indexing," in *Proc. 40th Int. Conf. Mach. Learn. (ICML)*, 2023, pp. 1–12.

[30] A. Boffa, P. Ferragina, and G. Vinciguerra, "A learned approach to design compressed rank/select data structures," *ACM Trans. Algorithms*, vol. 18, no. 3, pp. 1–28, Oct. 2022.

[31] P. Ferragina, G. Manzini, and G. Vinciguerra, "Compressing and querying integer dictionaries under linearities and repetitions," *IEEE Access*, vol. 10, pp. 118831–118848, 2022.

[32] P. Ferragina and G. Vinciguerra, "Learned data structures," in *Recent Trends in Learning From Data*, L. Oneto, N. Navarin, A. Sperduti, and D. Anguita, Eds. Cham, Switzerland: Springer, 2020, pp. 5–41.

[33] B. Spector, A. Kipf, K. Vaidya, C. Wang, U. F. Minhas, and T. Kraska, "Bounding the last mile: Efficient learned string indexing," in *Proc. 3rd Int. Workshop Appl. AI Database Syst. Appl. (AIDB)*, 2021, pp. 1–5.

[34] Y. Wang, C. Tang, Z. Wang, and H. Chen, "SIndex: A scalable learned index for string keys," in *Proc. 11th ACM SIGOPS Asia–Pacific Workshop Syst.*, Aug. 2020, pp. 17–24.

[35] D. Amato, G. Lo Bosco, and R. Giancarlo, "On the suitability of neural networks as building blocks for the design of efficient learned indexes," in *Proc. 23rd Int. Conf. Eng. Appl. Neural Netw. (EANN)*, 2022, pp. 115–127.

[36] G. Fumagalli, D. Raimondi, R. Giancarlo, D. Malchiodi, and M. Frasca, "On the choice of general purpose classifiers in learned Bloom filters: An initial analysis within basic filters," in *Proc. 11th Int. Conf. Pattern Recognit. Appl. Methods*, 2022, pp. 675–682.

[37] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "SuRF: Practical range query filtering with fast succinct tries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2018, pp. 323–336.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[39] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A height optimized trie index for main-memory database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2018, pp. 521–534.

[40] X. Wu, F. Ni, and S. Jiang, "Wormhole: A fast ordered index for in-memory data management," in *Proc. 14th EuroSys Conf.*, Mar. 2019, pp. 1–16.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. U. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 5998–6008.

[42] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *Proc. 8th Int. Conf. Learn. Represent. (ICLR)*, 2020, pp. 1–12.

[43] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *Proc. AAAI Conf. Artif. Intell.*, vol. 35, no. 12, 2021, pp. 11106–11115.

[44] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ, USA: Prentice-Hall, 1994.

[45] *Keras API*. Accessed: Feb. 1, 2023. [Online]. Available: https://keras.io/api/utils/model_plotting_utils/#plot_model-function

[46] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 12, pp. 6999–7019, Dec. 2022.

[47] Q. Zhang, D. Zhou, and X. Zeng, "HeartID: A multiresolution convolutional neural network for ECG-based biometric human identification in smart health applications," *IEEE Access*, vol. 5, pp. 11805–11816, 2017.

[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[49] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2017, pp. 1442–1451.

[50] M. Hermans and B. Schrauwen, "Training and analysing deep recurrent neural networks," in *Proc. 27th Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 26, 2013, pp. 190–198.

[51] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, "DeepZip: Lossless data compression using recurrent neural networks," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2019, p. 575.

[52] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, Y. Bengio and Y. LeCun, Eds., 2015, pp. 1–15.

[53] G. C. Marinó, A. Petrini, D. Malchiodi, and M. Frasca, "Deep neural networks compression: A comparative survey and choice recommendations," *Neurocomputing*, vol. 520, pp. 152–170, Feb. 2023.

[54] M. Gupta and P. Agrawal, "Compression of deep learning models for text: A survey," *ACM Trans. Knowl. Discovery From Data*, vol. 16, no. 4, pp. 1–55, Jan. 2022.

[55] X. Wang, J. Li, X. Kuang, Y.-A. Tan, and J. Li, "The security of machine learning in an adversarial setting: A survey," *J. Parallel Distrib. Comput.*, vol. 130, pp. 12–23, Aug. 2019.

[56] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, "A survey on adversarial attacks and defences," *CAAI Trans. Intell. Technol.*, vol. 6, no. 1, pp. 25–45, Mar. 2021.

[57] E. M. Kornaropoulos, S. Ren, and R. Tamassia, "The price of tailoring the index to your data: Poisoning attacks on learned index structures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2022, pp. 1331–1344.
[58] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.
[59] M. Kurmanji and P. Triantafillou, "Detect, distill and update: Learned DB systems facing out of distribution data," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 1–27, May 2023, Art. no. 33, doi: 10.1145/3588713.
[60] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

PAOLO FERRAGINA received the Ph.D. degree in computer science from the University of Pisa, in 1996. He was a Postdoctoral Researcher with Max-Planck Institut für Informatik, Saabrücken, Germany, from 1997 to 1998. He is currently a Professor of algorithms with the University of Pisa, where he founded and leads the Acube Laboratory, whose research activities regard the design of algorithms for big data, mainly in the form of texts and graphs, in collaboration with companies worldwide, such as Bloomberg, European Broadcasting Union (EBU), Google, Tiscali, and Yahoo!. He has (co)authored more than 170 (refereed) publications, some books, and chapters, achieving an H-index of 34 on Scopus and more than 10,000 citations on Google Scholar. His research results got four U.S. patents and some international awards, such as the 1995 Best Land Transportation Paper Award from IEEE Vehicular Technology Society, the 1997 Best Ph.D. Thesis in Theoretical Computer Science by the Italian Chapter of the EATCS, the 1997 Philip Morris Award on Science and Technology, the Yahoo! Research Faculty Award, the three Google research awards, and the 2022 ACM Paris Kanellakis Award. He is serving on the editorial board of the *Journal of Graph Algorithms and Applications* (JGAA). He is an Area Editor of *Encyclopedias of Algorithms* (Springer) and *Encyclopedias of Big Data Technologies* (Springer).

MARCO FRASCA received the Ph.D. degree in computer science from the University of Milan, Italy, in 2012. Since 2017, he has been an Assistant Professor with the Department of Computer Science, University of Milan, where he is currently a member of the Anacleto Laboratory, whose research activities regard the field of machine learning applied in biology and medicine. He has been an Invited Research Visitor at several universities, including the Terrence Donnelly Centre for Cellular and Biomolecular Research, University of Toronto, and the Institute of Molecular Biology, Johannes Gutenberg University of Mainz. He contributed to consolidating the application of Hopfield networks to classification and ranking problems with the development of single- and multi-task parametric Hopfield models. His current research interests include the design and analysis of new machine-learning methods, with applications in bioinformatics, computational biology, and medicine.

GIOSUÈ CATALDO MARINÒ received the B.Sc. degree in computer science from the University of Milan, Italy, where he is currently pursuing the master's degree in computer science. His current research interests include machine learning and the compression of neural network models.

GIORGIO VINCIGUERRA received the Ph.D. degree in computer science from the University of Pisa, in 2022. He is currently a Research Fellow with the University of Pisa. His current research interests include compressed data structures, data compression, and algorithm engineering. His thesis was awarded the 2022 Best Ph.D. Thesis in Theoretical Computer Science by the Italian Chapter of the EATCS.