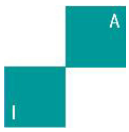
Inteligencia Artificial 26(71) (2023), 59-74
doi: 10.4114/intartif.vol26iss71pp59-74



INTELIGENCIA ARTIFICIAL

http://journal.iberamia.org/

Learning Picture Languages Using Dimensional Reduction

David Kuboň, František Mráz, Ivan Rychtera

Charles University, Prague, Czech Republic
{dkubon,mraz,rychtera}@ksvi.mff.cuni.cz

Abstract One-dimensional (string) formal languages and their learning have been studied in considerable depth. However, the knowledge of their two-dimensional (picture) counterpart, which retains similar importance, is lacking. We investigate the problem of learning formal two-dimensional picture languages by applying learning methods for one-dimensional (string) languages. We formalize the transcription process from a two-dimensional input picture into a string and propose a few adaptations to it. These proposals are then tested in a series of experiments, and their outcomes are compared. Finally, these methods are applied to a practical problem and an automaton for recognizing a part of the MNIST dataset is learned. The obtained results show improvements in the topic and the potential to use the learning of automata in fitting problems.

Keywords: Learning, Grammatical inference, Automata, Formal languages. Picture languages.

1 Introduction

A considerable amount of research has been done in the field of one-dimensional formal languages, which now have a substantial position in the foundations of Computer Science. However, much less is known about formal languages in two dimensions, even though they have both theoretical and practical importance comparable to one dimension. For example, we could list the automatic detection of different shapes (e.g., road signs) or, more generally, any problem on two-dimensional data with some pattern regularity [12].

In some literature and also throughout this paper, the terms two-dimensional and *picture* languages will be used interchangeably. To distinguish them from pictures in the broader, common sense, formal picture languages have a formally exact mathematical description and are not defined as sets of pictures containing, for example, cars, which cannot be defined rigorously. Therefore, deep neural networks, typically very efficient with recognizing objects in images [6], mostly fail to learn picture languages in the formal sense. However, powerful models of automata exist that work on picture languages but lack the efficiency and determinism needed for more practical applications.

Several papers have already been published [8] focusing on finding methods to learn picture languages from positive and negative examples. This process of learning a model (grammar) for a target language based on some information about the words of the language is called grammatical inference [2]. There are multiple known grammatical inference algorithms for a number of classes of languages in the one-dimensional domain of the problem, but almost no knowledge in two (or more) dimensions.

Of course, how we represent pictures influences the class of picture languages we can work with and the possible grammatical inference methods we can use. In the literature, the formal representation of

ISSN: 1137-3601 (print), 1988-3064 (on-line)

pictures differs. One option is generative, which describes how a picture can be generated from a string. Freeman [4] introduced an 8-letter alphabet with moves representing all eight directions (north, south, east, west, northeast, southeast, northwest, and southwest). Later Maurer et al. [11] simplified the alphabet into a 4-letter alphabet (up, down, left, right). Both alphabets can represent the way in which a picture is drawn. In order to generate colored pictures, the latter approach was extended with labels by Costagliola [1]. In either of these representations, a picture language is a set of strings describing all pictures in the language.

A second way to represent a picture is closer to the common form – a rectangular array of symbols that could be interpreted as colors of pixels in the image. In this representation, a picture language is the set of pictures accepted by an automaton working on two-dimensional inputs. Examples of such automata are extensions of finite automata for working on two-dimensional tape, e.g. deterministic four-way automaton [5], returning finite automaton [3] or boustrophedon finite automaton [3]. Much more powerful automata for two-dimensional inputs are a non-deterministic online tessellation automaton [5], an even more powerful sgraffito automaton [13] or a two-dimensional limited context restarting automaton [7]. The problem with these automata is their high complexity, as the problem of deciding whether an input image is accepted by any of them is NP-complete.

This paper follows up on an earlier study [8] that proposed a new representation for picture languages. It pursues the second way of representing pictures but instead of designing an automaton working on two-dimensional tape, it uses a function T called transcriptor that rewrites any two-dimensional picture p into a string $T(p)$ and a one-dimensional language L. The set of all pictures p for which $T(p)$ is in L defines the picture language. We propose a general transcription-evaluation framework in which T is any mapping from pictures to string and L is represented by a (string) automaton. As such framework is too general, here we restrict our attention to transcriptors implemented as a scanner automaton that decides in which order the symbols of an input picture will be inspected and a sequence dictionary that defines how a symbol and symbols in its neighborhood are rewritten into a string. Then we conduct multiple experiments with different languages, learning algorithms, and transcription mechanisms.

The paper is structured as follows. Section 2 introduces basic definitions for pictures and picture languages. Section 3 presents definitions related to the transcription-evaluation framework. Section 4 describes the experiments and the obtained results, and Section 5 concludes the paper.

2 Definitions

We define picture languages in a fashion that corresponds to pictures in common sense. A *picture* p over a finite alphabet Σ is a two-dimensional rectangular array of elements from Σ – see [5]. Let $\mathbb{Z}$ and $\mathbb{N}$ denote the sets of integers and non-negative integers, respectively. For $m, n \in \mathbb{N}$, we say that p has dimensions (m, n) if it has m rows and n columns. Then $p_{i,j}$ from Σ, $0 \leqslant i < m, 0 \leqslant j < n$ denotes the symbol at position j in row i. The set of all rectangular pictures over Σ of dimensions (m, n) will be denoted as $\Sigma^{m,n}$, and the set of all rectangular pictures over Σ of any dimension will be denoted as $\Sigma^{*,*}$. A *picture language* is any subset of $\Sigma^{*,*}$.

Any automaton working on a picture p of dimensions (m, n) needs to know where is the border of the picture. Therefore, the picture is typically surrounded by sentinels #, where $\# \notin \Sigma$. Delimited picture p is called a *boundary picture* $\widehat{p}$ over $\Sigma \cup \{\#\}$ of dimensions $(m + 2, n + 2)$:

#	#	#	$\cdots$	#	#	#
# $\vdots$ #			P			# $\vdots$ #
#	#	#	$\cdots$	#	#	#

A particular class of formal languages is the class of *locally k-testable languages*, a subclass of regular languages that is learnable from positive samples ([2]), which will be used later in the paper.

A k-testable language L is characterized by a set T of strings of length k, a set I of prefixes and a set F of suffixes with length $k - 1$, and a set C of accepted words shorter than k. This means that each

word from L longer than k has only substrings from T and a prefix and a suffix from the limited sets I and F. Therefore, any word can simply be checked using the sets if it fulfills these conditions.

Definition 1 ([8]). *A (string) language $L \subseteq \Sigma^*$ is called k-testable if there exist four finite sets of words: $I \subseteq \Sigma^{k-1}, F \subseteq \Sigma^{k-1}, C \subseteq \Sigma^{<k}$, and $T \subseteq \Sigma^k$ such that a word belongs to L if it is from C, or its prefix of length $k-1$ is in I, its suffix of length $k-1$ is in F, and all substrings of length k belong to T.*

Given a set of positive samples, we can extract the set of all present prefixes and suffixes of length $k-1$ as I and F, respectively, the set of substrings of length k as T and a set of words shorter than k letters as C. Using this knowledge base, for any new word, we can decide if it belongs to the language in question.

3 Transcription-evaluation Framework

We dedicate this section to methods for recognizing picture languages that convert (transcribe) pictures into strings and subsequently apply a string automaton to recognize them. Our approach aims to leverage our knowledge and expertise in solving problems in the domain of one-dimensional languages to aid us in tackling more complex challenges associated with picture languages.

The approach presented in [8] has shown to be promising; thus, we will explore it more in this paper. In the article, various methods of transcribing the pictures into strings are used and followed by an algorithm to construct a deterministic finite automaton (DFA) to classify the resulting strings.

Instead of simply copying the symbols of the input picture row-by-row or column-by-column, the proposed transcription methods have used a particular order that concatenates together the contents of overlapping 3-by-3 windows moving on the picture separated by a special symbol. The authors theorized this should have led to a better generalization of other simpler approaches as, for each symbol in the picture, the information from neighboring positions will remain in the neighborhood of the transcribed symbol.

Some preliminary experiments have shown that rather than simply rewriting the symbols in the original picture into one dimension, we can achieve better performance by transcribing the picture into a string over another larger alphabet. A letter of this alphabet should reflect the complete contents of a 3-by-3 window. As many DFA learning algorithms use a prefix tree built from obtained sample words, the large alphabet reduces the depth of the prefix trees and speeds up their processing.

Definition 2. *Let Σ and Γ be alphabets, and Σ does not contain the symbol $\#$. Then* transcriptor-evaluator machine for picture languages *(TEMPL) is a pair $M = (T, E)$, where T is a map from $(\Sigma \cup \{\#\})^{**}$ to Γ^* and E is a string automaton accepting a language $L(E) \subset \Gamma^*$.*

We say that M accepts a picture language $L(M) = L(T, E) = \{p \in \Sigma^{**} \mid T(p) \in L(E)\}$.

In the above definition, T defines a *transcriptor* that converts a two-dimensional input picture into a word over Γ, and the automaton E is an *evaluator* that decides whether to accept or to reject. In general, the transcriptor T can be any mapping of pictures into strings. In this paper, we will limit the transcription to a two-part process. First, a scanning sequence is designed using a simple automaton. Then a constant dictionary is used to map fixed-size fragments of the picture into substrings according to the order determined by the scanning sequence. We call these parts *scanner* and *sequence dictionary*.

A scanning sequence for a picture p of dimension (m, n) is a finite sequence, where each element is a pair of integers $(i, j); 0 \leqslant i < m + 2, 0 \leqslant j < n + 2$. The pair (i, j) represents a position on the boundary picture $\hat{p}$. A scanning sequence can be obtained, e.g., by recording each position in a picture row-by-row and left-to-right. However, any fixed strategy can limit the power of TEMPL. Hence, we introduce a more general way of producing a scanning sequence.

Definition 3. *A four-way scanner automaton (4SA) is a system $M_s = (Q, \Sigma, \#, \Delta, q_0, q_h, \delta)$, where Q is a set of states, Σ is an input alphabet, $\#$ is the border symbol, $\# \notin \Sigma$, $\Delta = \{\ell, \mathrm{r}, \mathrm{u}, \mathrm{d}\}$, q_0 is the starting state, q_h is the halting state and $\delta \subset Q \times (\Sigma \cup \{\#\}) \to Q \times \Delta \times \{\mathrm{pos}, \varepsilon\}$ is the transfer function.*

A 4SA is a finite-state device with a head that scans one position on a boundary picture. In each step of its computation, the scanner changes its state and position according to the transfer function δ.

If a is the symbol under the head and $q \in Q$ is its current state, and $\delta(q, a) = (q', d, \text{pos})$, the automaton appends its current position (i, j) to its output, it enters state q' and moves its head to the neighboring position according to the direction d. The possible values $\ell, \text{r}, \text{u}, \text{d}$ of d correspond to the directions left, right, up, and down, respectively. If $\delta(q, a) = (q', d, \varepsilon)$, the automaton continues as above, with the exception that it does not append its position to the output sequence.

In this paper, we add a further constraint on δ: for each state $q \in Q$ and symbols $a, b \in \Sigma$, we require that $\delta(q, a) = \delta(q, b)$. The scanner can only differentiate if it reads a symbol from the picture or the border symbol. Hence if two pictures have the same dimensions, the scanner produces for them identical scanning sequences.

For an input picture p, a computation of a 4SA starts in the initial state q_0 at position $(1, 1)$ of the boundary picture $\hat{p}$ and ends by entering the halting state q_h. The output of the computation is a scanning sequence. We call the elements of this sequence *anchors*.

We place an additional constraint on the scanner: the output scanning sequence must contain each position of the picture p exactly once. Besides the positions on the picture p, the scanning sequence can contain also some positions on the boundary of $\hat{p}$.

Definition 4. *A sequence dictionary is a tuple $D = (\Sigma, \Gamma, \#, w, t, k)$ where Σ is the input alphabet, Γ is the output alphabet, $\#$ is the border symbol, $\# \notin \Sigma$, $w = ((r_1, d_1), \ldots, (r_\ell, d_\ell))$ for $\ell \geqslant 0, r_i, d_i \in \mathbb{Z}$ is a sequence of relative positions of length ℓ, $t : (\Sigma \cup \{\#\})^\ell \to \Gamma^k$ is a map, and k is a constant.*

For an input picture $\hat{p}$ and a position (i, j) on $\hat{p}$ from a scanning sequence, we apply the sequence dictionary $D = (\Sigma, \Gamma, w, t, k)$ with $w = ((r_1, d_1), \ldots, (r_\ell, d_\ell))$ in the following way. We concatenate symbols of $\hat{p}$ from the positions relative to (i, j) according to the sequence of relative positions w into a word $v \in (\Sigma \cup \{\#\})^l$. Thus, $v = \hat{p}_{i+r_1, j+d_1} \cdots \hat{p}_{i+r_\ell, j+d_\ell}$. Then we append its transcription $t(v)$ to the transcription of the picture.

The transcriptor consisting of a 4SA M_s and sequence dictionary D will be denoted as $T[M_s, D]$. For an input picture $p \in \Sigma^{*,*}$, the transcriptor $T[M_s, D]$ produces the string $T[M_s, D](p) = z_1 \cdots z_m$, where m is the length of scanning sequence $\{(i_1, j_1), \ldots, (i_m, j_m)\}$ produced by M_s for input picture p, and

$$z_a = t(\hat{p}_{i_a+r_1, j_a+d_1} \hat{p}_{i_a+d_2, j_a+r2} \cdots \hat{p}_{i_a+r_\ell, j_a+d_\ell}), \text{ for } a = 1, ..., m.$$

Perhaps the most obvious 4-way scanner automaton M_{rr} scans the input picture row-by-row from left to right. Formally, $M_{rr} = (Q_{rr}, \Sigma, \#, \Delta, q_0, q_h, \delta_{rr})$, where $Q_{rr} = \{q_0, q_r, q_\ell, q_\#, q_h\}$, $\Delta = \{\ell, \text{r}, \text{u}, \text{d}\}$, and the transition function δ_{rr} is defined as follows:

$$\begin{array}{lcl}
\delta_{rr}(q_0, x) & = & (q_r, \text{r}, \text{pos}), \text{ for all } x \in \Sigma, \\
\delta_{rr}(q_r, x) & = & (q_r, \text{r}, \text{pos}), \text{ for all } x \in \Sigma, \\
\delta_{rr}(q_r, \#) & = & (q_\ell, \text{l}, \varepsilon), \\
\delta_{rr}(q_\ell, x) & = & (q_\ell, \text{l}, \varepsilon), \\
\delta_{rr}(q_\ell, \#) & = & (q_d, \text{d}, \varepsilon), \\
\delta_{rr}(q_d, \#) & = & (q_0, \text{r}, \varepsilon), \\
\delta_{rr}(q_0, \#) & = & (q_h, \ell, \varepsilon),
\end{array}$$

The 4SA starts at the top-left corner of an input picture p in the initial state q_0. It outputs its position $(1, 1)$ (relative to $\hat{p}$), enters the state q_r and moves to the right. Then it continues moving right in the state q_r while outputting positions until it enters the right border of $\hat{p}$. There M_{rr} changes its state into q_ℓ and starts to move to the left without outputting positions. At the left border, the automaton moves down into the state q_d and then moves right into the initial state q_0. If the automaton is in the initial state q_0 and sees the border symbol $\#$, it enters the halting state q_h and finishes its computation.

Suppose a sequence dictionary $D = (\{a\}, \{0, 1\}, \#, ((-1, -1), (-1, 0), (0, -1), (0, 0)), t)$, where for every word $v \in \{a, \#\}^4$ it holds $t(v) = 1$ if $v = \#\#\#a$ and $t(v) = 0$ otherwise. The sequence of the dictionary corresponds to an upper-left 2-by-2 square relative to the anchor.

Let p be the square picture over $\{a\}$ of dimensions $(2, 2)$. On this picture, the 4SA M_{rr} produces the scanning sequence $\{(1, 1), (1, 2), (2, 1), (2, 2)\}$. The strings obtained from the relative positions to the four anchors in the scanning sequence are $\#\#\#a$, $\#\#aa$, $\#a\#a$, and $aaaa$, resulting in the output word 1000 (see Figure 1 with marked positions relative to the first and last anchors).

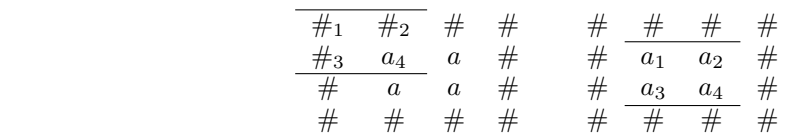


Figure 1: The boundary picture $\hat{p}$ with symbols marked at relative positions $\{(-1,-1),(-1,0),(0,-1),(0,0)\}$ with respect to anchors $(1,1)$ and $(2,2)$.

In our experiments described in Section 4, we use 4SA M_{rr} with two sequence dictionaries D_{oo} and D_{mo}. Both sequence dictionaries use the same sequence of relative positions $w = ((-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1))$ that encompasses the 3-by-3 window around the anchor row-by-row from left to right and from top to bottom.

We refer to the first sequence dictionary $D_{oo} = (\Sigma, \Sigma \cup \{\#\}, \#, w, t_{oo}, 9)$ as a *one-to-one encoder*, as its map t_{oo} is the identity mapping that maps the contents of the window into the string of length 9.

The second sequence dictionary D_{mo} maps the contents of a scanning window into a single symbol from the alphabet $(\Sigma \cup \{\#\})^9$. As the alphabet is of size $(|\Sigma|+1)^9$, its symbols can be represented as integers between 0 and $(|\Sigma|+1)^9 - 1$. This sequence dictionary is referred to as a *many-to-int encoder*.

An evaluator used in TEMPL can be an arbitrary automaton accepting words (one-dimensional strings of symbols) over a finite alphabet. In this paper, we concentrate on simple evaluators that can be represented by deterministic finite automata. As there are several known grammatical inference algorithms able to learn finite automata from sets of positive and negative samples, we will use them for learning TEMPLs for picture languages. While using deterministic finite automata as evaluators seems to be a severe restriction, we will see that such TEMPLs can still simulate some known types of picture automata and accept an interesting class of picture languages.

3.1 TEMPL Can Simulate Returning Finite Automaton

A returning finite automaton [3] is a finite-state automaton working on two-dimensional input. It scans the picture row-by-row from left to right. When it arrives at the sentinel # on the right end of an extended picture, it returns to the leftmost position on the next row of the picture. If the row is the last row of the picture, the automaton halts and accepts if it is in an accepting state or rejects otherwise.

The following definition differs from the source [3] since the original seemed overly complicated for our purpose. However, our definition results in an equivalent class of accepted picture languages. The original definition includes rewriting symbols visited by the automaton with a special symbol $\square \notin \Sigma \cup \{\#\}$. However, the rewritten symbols serve just for determining the next visited position of the picture. Our definition avoids such rewriting.

Definition 5. *A deterministic* returning finite automaton *(RFA) is a 6-tuple* $R = (\Sigma, Q, \delta, q_0, Q_f, \#)$, *where* Σ *is an alphabet,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $Q_f \subset Q$ *is a set of accepting states, and* $\delta : Q \times (\Sigma \cup \{\#\}) \to Q$ *is a transition function and* # *is a special symbol not in* Σ.

For an input picture p of dimension $(m.n)$, RFA R works on the picture with added columns of #'s on the left and on the right. Such an extended picture, denoted as p', has dimension $(m, n+2)$. It is actually the boundary picture $\hat{p}$ but without the first and last row of #'s.

The automaton R starts with its head at position $(0,1)$ of the picture p' in the initial state q_0. It scans the symbol $p'_{0,1} = p_{0,0}$ under its head, enters the state $\delta(q_0, p'_{0,1})$ and moves to the right. In the same way, it reads the whole first row from left to right while changing its state according to its transition function δ. When it arrives at the right border of p' in state q, it changes its state to $\delta(q, \#)$ and places its head at the first symbol of p in the next row and continues again with scanning the row from left to right. When R arrives at the right border in the last row of p' in a state q', it halts and accepts p if $q' \in Q_f$, otherwise, it halts and rejects p.

To illustrate the computational power of TEMPL, we claim that it can simulate any returning finite automaton.

Theorem 1. *For each RFA* R, *there exists a TEMPL* M *such that* $L(M) = L(R)$.

Proof. *Let $R = (\Sigma, Q_r, \delta_r, q_0, Q_f, \#)$ be a RFA. We will construct a TEMPL $M = (T[M'_{rr}, D_r], E_r)$ such that it accepts the same picture language as RFA R.*

The four-way scanner automaton M'_{rr} should enable transcription symbol-by-symbol in each row from left to right, top to bottom. Rows need to be separated by the # symbol because it signals the RFA that it has reached the end of a row. Hence, the 4SA M'_{rr} is almost identical to the 4SA M_{rr} from the previous section. It differs only in that it outputs position also when it scans # at the right border of the boundary picture. Hence, $M'_{rr} = (Q_{rr}, \Sigma, \#, \Delta, q_0, q_h, \delta'_{rr})$, where all components of M'_{rr} are the same as in M_{rr} and δ'_{rr} differs from δ_{rr} only in that $\delta'_{rr}(q_r, \#) = (q_\ell, \ell, \text{pos})$.

The sequence dictionary D_r will only copy the scanned symbol from the anchor position. Formally, $D_r = (\{(\Sigma \cup \{\#\}), (\Sigma \cup \{\#\}), \{(0,0)\}, e, 1)$, where e is the identity mapping on $\Sigma \cup \{\#\}$. The evaluator E_r can be obtained from R as a deterministic finite automaton with input alphabet $\Sigma \cup \{\#\}$, the same set of states, the same initial state, the same set of accepting states, and the same transition function as RFA R; that is, $E_r = (\Sigma \cup \{\#\}, Q_r, \delta_r, q_0, Q_f)$.

Because D_r is a direct one-to-one mapping, E_r will process input in the order specified by R. And since E_r uses the same components as R, we only need to prove that the scanning sequence produced by M'_{rr} on $\hat{p}$ are the positions scanned by RFA R on p'.

M'_{rr} starts at the position $(1,1)$ of $\hat{p}$, which corresponds to the position $(0,1)$ on the picture p'. If the picture is not empty, it scans the whole row until the border. While scanning the row, all positions are outputted, including the position of the symbol # at the right border. Then the scanner automaton returns to the first column and moves down into the state q_d and further into the initial state q_0. The computation of M'_{rr} continues similarly as of M_{rr} until each position of p is scanned.

Therefore, the scanning sequence of M'_{rr} on p corresponds to the sequence of positions visited by RFA R. D_r simply copies the symbols visited by RFA R. Thus, the input $T[M'_{rr}, E_r](p)$ for E_r is exactly the sequence of symbols visited by RFA R, and E_r accepts it if and only if R accepts it. This shows that $L(T) = L(T[M'_{rr}, D_r], E_r) = L(R)$. □

In [3], the authors study also another type of automaton working on pictures called the boustrophedon finite automaton. A boustrophedon automaton differs from an RFA in its scanning strategy. Namely, it scans even rows of p' from left to right and the odd rows from right to left. Boustrophedon automata and returning finite automata accept the same class of picture languages (see [3]). It is easy to see that for a given boustrophedon automaton, we can directly construct an equivalent TEMPL by modifying the used four-way scanner automaton.

4 Experimental Results

Our ultimate goal is to learn a target picture language from positive and negative examples of pictures for the target picture language. For simplicity, we will fix a scanner and a sequence dictionary. This enables us to transcribe all sample pictures into strings. In this way, we obtain a set of positive and negative examples for the (string) language that should be accepted by an evaluator. Subsequently, the evaluator is trained (learned) from the positive and negative (string) samples. For learning evaluators, we use learning k-locally testable languages and a version of a state-merging algorithm.

This section is divided into the following subsections. In Section 4.1, we describe several sample picture languages and the way how we have created train and test sets for the sample picture languages. The experiment setup is presented in Section 4.2. The following Section 4.3 and Section 4.4 are devoted to the results obtained using one-to-one and many-to-one encoders, respectively. In Section 4.5 we briefly compare results obtained with different four-way scanner automata. Additionally, we have applied our method for learning for a subset of the MNIST dataset in Section 4.6. Finally, Section 4.7 summarizes the results of the experiments.

4.1 Datasets

To verify the suitability of our new representation of picture languages for their learning, we conducted a series of experiments with seven picture languages over the binary alphabet $\{▫, ▪\}$ corresponding to white and black pixels, respectively (see Figure 2 for sample pictures):

L_1 is the set of all white rectangles containing a black diagonal till the border of the picture. The diagonal can start in either of the top corners.

L_2 is the set of all white pictures of dimensions at least $(3, 3)$ with a black border of one-pixel width.

L_3 is the set of all pictures with a positive number of black rows followed by a positive number of white rows.

L_4 is the set of all pictures with a regular chessboard pattern of ▫-s and ▪-s. The top-left corner of such a picture can contain ▫ or ▪, but the whole picture must have the chessboard pattern.

L_5 is the set of all pictures where the top left quadrant is black and the rest white.

L_6 is the set of all pictures with alternating black and white rows. The first row can be either black or white.

L_{mnist1} is based on the dataset of handwritten digits MNIST [10]. All pictures of digit one are treated as positive, and all other digits as negative examples. We adapted the pictures from gray-scale to black and white.

We also added a noisy version for each crisp language $L_1, \ldots, L_6$. Each picture belonging to a noisy language must not differ from a positive crisp example by more than a given noise threshold – in our experiments, it was 5%.

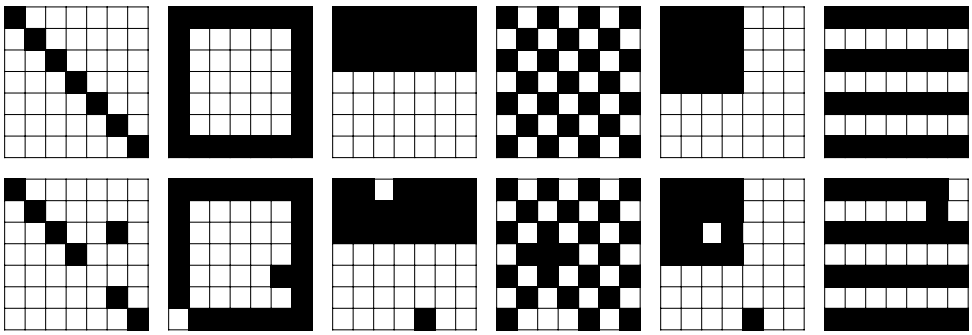


Figure 2: Sample pictures from languages $L_1, \ldots, L_6$. In the first row, there are samples from the crisp picture languages, and in the second row, there are samples of noisy picture languages.

Experimental datasets for $L_1, \ldots, L_6$ were randomly selected from a randomly generated pool of pictures of various sizes. Pictures of larger sizes are more common in order to accommodate the larger space needed to be sampled. Positive and negative examples are handled separately. Positive examples with zero noise are always included in the pool.

The languages are sparse. Hence crisp positive examples can be obtained easily. Noisy positive examples are obtained by randomly flipping pixels. Negative examples are generated from the positive ones by flipping pixels selected uniformly randomly. We discard negative examples if they accidentally become positive. For noisy languages, the number of differing pixels is counted for each crisp positive picture and then tested whether it exceeds the noise threshold of 5%.

From this pool, train, and test sets are uniformly randomly chosen. If there are too few positive examples in the pool to match the number of negative examples, negative examples are added so that the sets have the same total sizes across all languages.

The specific training and testing sets required for our experiments were generated using scripts (available upon request) and are fully reproducible. Sample sets comprised 100, 200, 400, 800, 1600, and 3200 pictures for each sample language, ranging from dimensions $(5, 5)$ to $(10, 10)$. Each set contained the same number of positive and negative examples, if possible. However, for training k-locally testable (string) languages, only positive examples in the training sets are used – see below.

4.2 Experiment Setup

The experiments were set as follows. First, a generator generates sets of positive and negative sample pictures L. Those are stored and fed into TEMPL, where each picture is transcribed into a string using a four-way scanner automaton and a scanner dictionary. In this way, we obtain a set of positive and negative sample strings S. For set S, a deterministic finite-state automaton consistent with S is learned.

In the first round of our experiments, we stick to the single scanning strategy – row-by-row and from left to right in each row – implemented as the 4SA M_{rr}. For transcription, we use sequence dictionaries D_{oo} (see Section 4.3) and D_{mo} (see Section 4.4). The first sequence dictionary simply rewrites the contents of a 3-by-3 scanning window into a string of 9 symbols. This is referred to as a one-to-one encoder. The second sequence dictionary maps the contents of a scanning window into a single symbol from the alphabet $\{\square, \blacksquare, \#\}$. As the alphabet is of size 19683, the symbol is represented as an integer between 0 and 19682. This sequence dictionary is referred to as a many-to-int encoder.

In the second round of experiments in Section 4.5, we compare several different scanning strategies.

For learning, we employed learning k-testable languages, for learning from positive examples, and a state merging algorithm `traxbar`, for learning from both positive and negative examples.

Training for k-locally testable languages consists in collecting the sets of possible prefixes and suffixes of length $k-1$, and infixes of length k from all positive samples.

The algorithm `traxbar` is our python implementation of a version of breadth-first Trakhtenbrot-Barzdin's state merging algorithm [9].

Once the learning is finished, the resulting automaton is tested on an independent test set of pictures that are rewritten into strings in the same way. As our sample languages are rather sparse, we do not report accuracy as it usually considerably differs between positive and negative samples. Instead, we use F_1-score defined as

$$\frac{Precision \cdot Recall}{Precision + Recall},$$

where

$$Precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN},$$

TP, FP, and FN stand for the number of true positive, false positive, and false negative samples, respectively.

4.3 One-to-one Encoding of Window Contents

First, we considered the combination of 4SA M_{rr} implementing the row-by-row scanning of input pictures and the sequence dictionary D_{oo} that ensures the same encoding of the contents of a scanning window into a string as in [8]. On each step of the scanning, the contents of the scanning window produced a string of length 9. In contrast to [8], the transcriptor $T[M_{rr}, D_oo]$ does not separate the consecutive contents of the scanning window by any separator.

Unsurprisingly, such encoding of the contents of the scanning window produces long strings. Training `traxbar` on a set of strings obtained from relatively small samples of pictures was prohibitively slow (see Figure 3) with a terrible accuracy (F_1-score close to zero). From the plots, we can see that the combination of one-to-one encoding and `traxbar` is unusable.

Conversely, combining one-to-one encoding with learning k-locally testable languages is feasible (see Figure 4). When using k-locally testable languages, we must also choose a proper order k for locally testable languages. In the figure, there are plotted results of experiments with $k = 2, 4, 6, \ldots, 20$ with the language L_1 of diagonals. The best F_1-score was achieved for $k = 18$ and $k = 20$. The results for the other sample languages $L_2, \ldots, L_6$ were also promising, and simultaneously, they were obtained in a very short training time.

4.4 Many-to-int Encoding of Window Contents

In the next part of our experiments, we combined 4SA M_{rr} implementing the row-by-row scanning of input pictures with the sequence dictionary D_{mo} that converts the contents of the whole window (9 symbols) around an anchor into a single symbol from a bigger alphabet. In our experiments, the

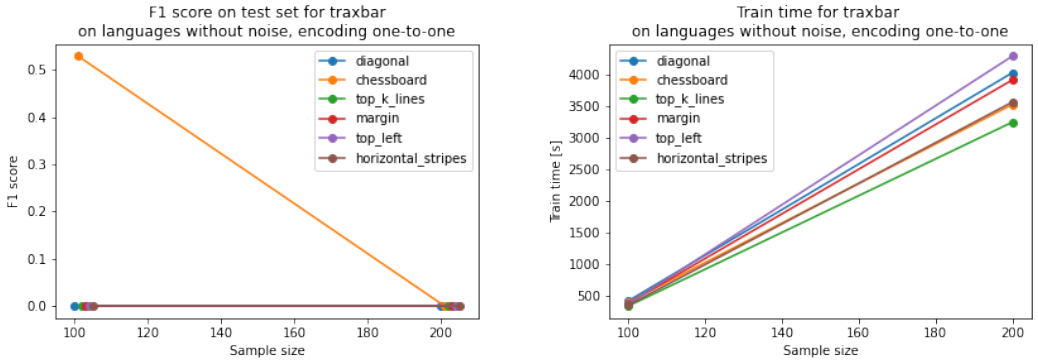


Figure 3: Results of learning picture languages by training finite automata using `traxbar` on 100 and 200 samples in train/test sets with the one-to-one encoding of the contents of the scanning window. The markers for different picture languages in the left plot are shifted a little to show overlapping marks.

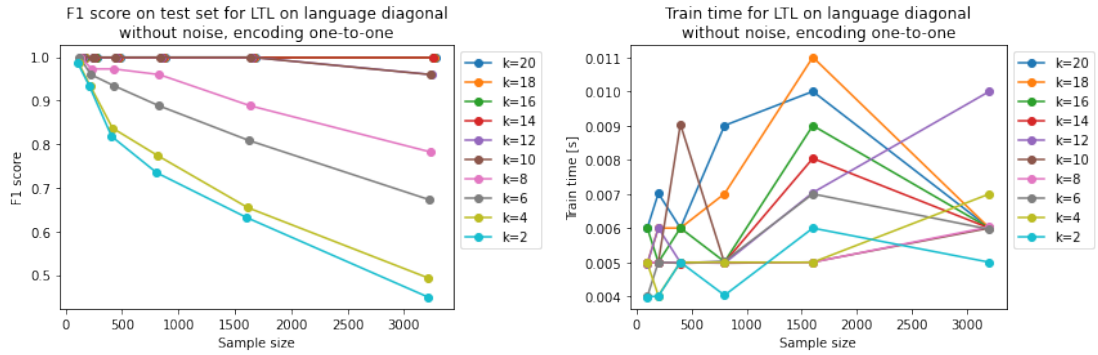


Figure 4: F_1-score on test sets on the left and time for training on the right when learning the picture language L_1 ("diagonal") by learning finite automata using k-locally testable languages on sample sets with 100, 200, 400, 800, 1600, and 3200 samples in train/test sets and with the one-to-one encoding. The markers for different values of $k = 2, 4, \ldots, 20$ are shifted a little to show overlapping marks.

number of possible contents of the scanning window has an upper limit of $3^9 = 19683$, as one field of the window can contain a white pixel, a black pixel, or the border marker #. Of course, not all combinations of pixels and border markers are possible, but still, the alphabet is quite large. Therefore, we encode one symbol of the alphabet simply as an integer.

For different randomly generated train and test sets of the same size, the resulting F_1-scores and training times are not constant. Therefore, in the following, we will plot the average F_1-score and train time from 10 randomly generated train and test sets for each size. To illustrate variance in the achieved results, we use vertical error bars of length equal to the sample standard deviation of the measurements.

Using the many-to-int encoding, `traxbar` produced a reasonable F_1-score for all tested crisp versions of the picture languages $L_1, \ldots, l_6$ and also for their noisy versions (Figure 5 and Figure 6). Except for the noisy version of the language L_5, the training time for `traxbar` is not higher than 300 seconds up to the sample size of 3200. F_1-score is between 0.5 and 0.8, which is quite good for mostly sparse languages in our samples.

Next, we experimented with learning the picture language L_1 by learning k-locally testable languages when using the many-to-int encoding. The obtained results are surprising. At first, we examined how the value of k influences the F_1-score. for the sample language L_1, we can see in Figure 7 and Figure 8 that the resulting average F_1-score is the highest for $k = 2$.

Similarly, for other sample picture languages, the value $k = 2$ together with the many-to-int encoding was the best combination. Interestingly, the value $k = 2$ with the many-to-int encoding corresponds exactly to $k = 18$ for the best results obtained with the one-to-one encoding.

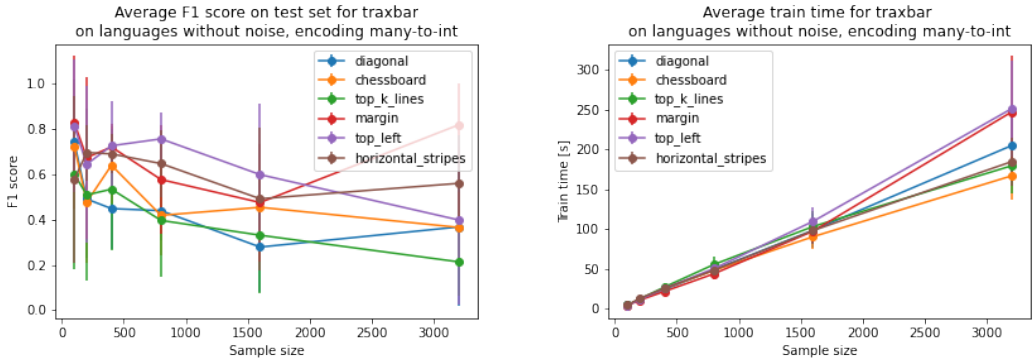


Figure 5: Results of learning crisp sample picture languages by training deterministic finite automata for using `traxbar` with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on the left and the average time for training on train sets on the right. The length of the vertical error bars is the sample standard deviation.

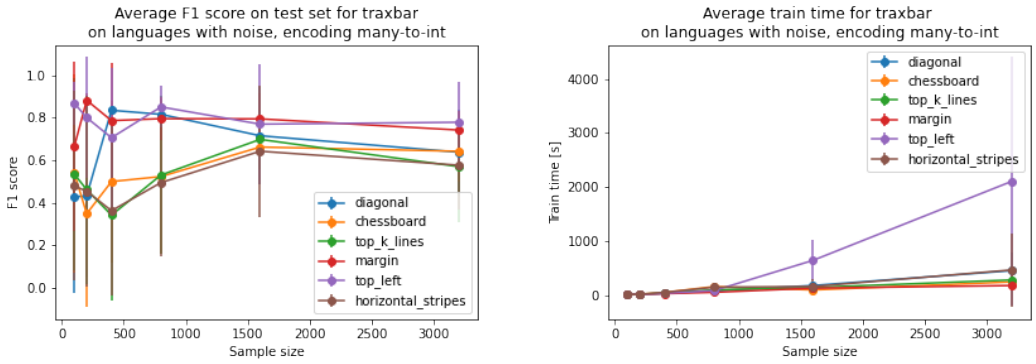


Figure 6: Results of learning noisy picture languages by training finite automata using `traxbar` with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on the left and the average time for training on train sets on the right. The length of the vertical error bars is the sample standard deviation.

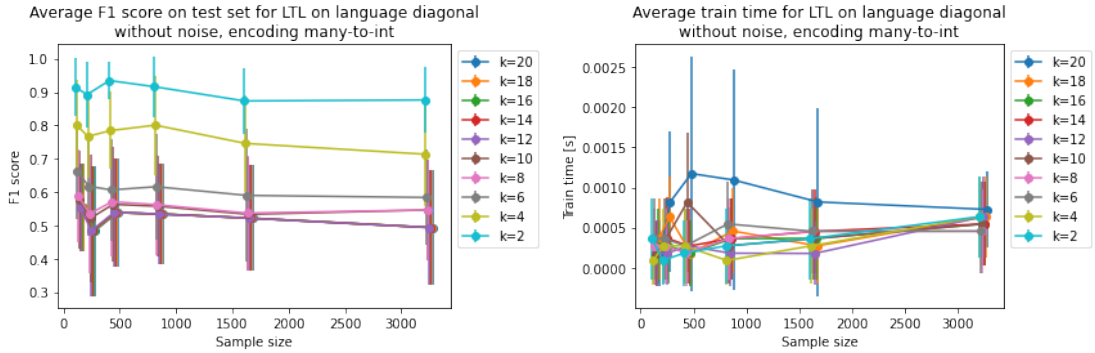


Figure 7: Results of learning crisp sample picture language L_1 by training finite automata using k-locally testable languages with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on the left and the average time for training on train sets on the right. The markers for different values of k are shifted a little to show overlapping marks.

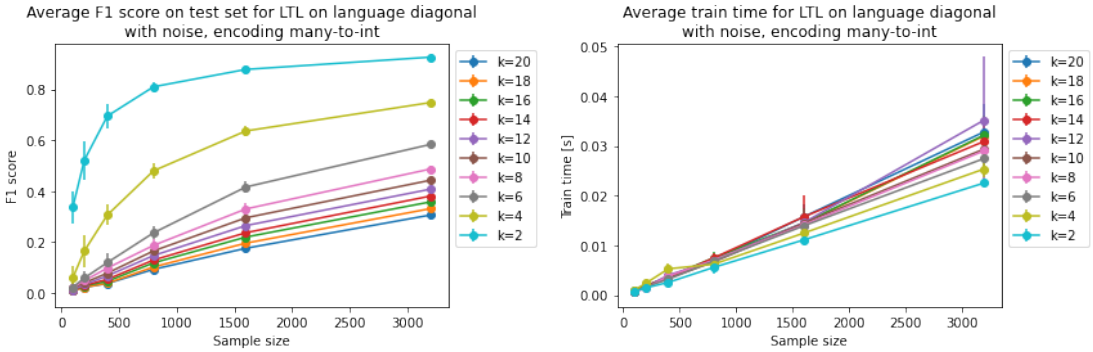


Figure 8: Results of training finite automata using k-locally testable languages for the noisy sample language L_1 with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on the left and the average time for training on train sets on the right.

For the rest of our sample languages, the performance of learning crisp and noisy sample picture languages using 2-locally testable languages is plotted in Figure 9 and Figure 10. For all crisp sample languages, the F_1-score is close to 1, except for the sample language L_1 (of diagonals), for which it achieves 0.88 only. It is probably caused by the very low number of positive samples. For all noisy sample languages, including the noisy version of L_1, the F_1-score converges to a value around 0.95 with the growing size of the training sample. The convergence is very stable, which can be seen from very short error bars.

Additionally, we can see a linear growth of the time required for training 2-locally testable languages with respect to the growing size of the training sample. Simultaneously, the training time is low. Lower than 0.03 s even for sample sets of size 3200.

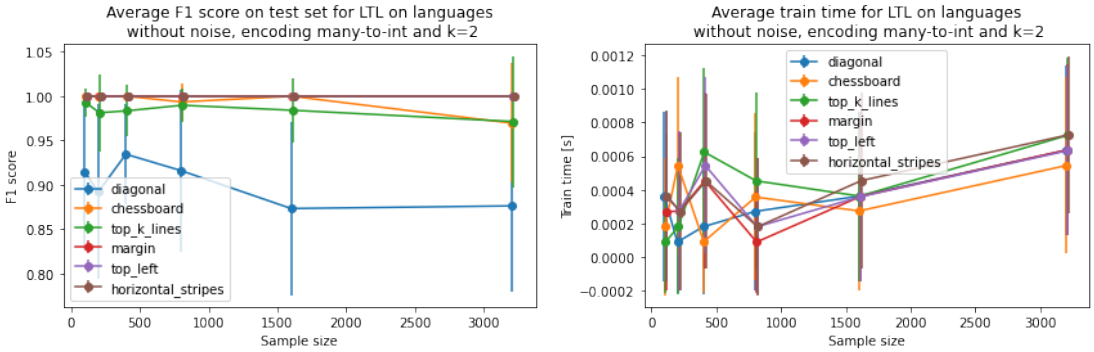


Figure 9: Results of learning sample picture languages by training finite automata using 2-locally testable languages for the crisp sample languages $L_1, \ldots, L_6$ with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on the left and the average time for training on train sets on the right. The markers for different languages are shifted a little to show overlapping marks.

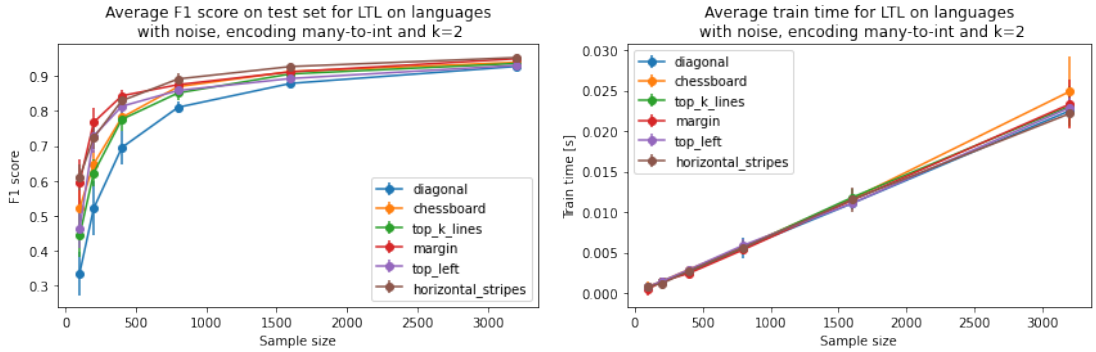


Figure 10: Results of learning sample picture languages by training finite automata using 2-locally testable languages for the noisy sample languages $L_1, \ldots, L_6$ with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on the left, and the average time for training on train sets on the right. The markers for different languages are shifted a little to show overlapping marks.

4.5 Scanner Behavior

This set of experiments is dedicated to surveying several scanning strategies used 4SAs:

1. Row-by-row implemented as the 4SA M_{rr}.

2. Column-by-column implemented by a 4SA M_{cc} that works similarly as M_{rr} but uses directions flipped around the main diagonal; that is, instead of directions right, left, up, and down in the transition function of M_{rr}, the transition function of M_{cc} uses directions down, up, left and right, respectively.

3. Snake-by-row implemented by a 4SA M_{sr} that scans the boundary picture in boustrophedon style (cf. the boustrophedon finite automaton at the end of Section 3.1) and outputs only positions of input symbols different from the boundary symbol #.

4. Snake-by-column implemented as a 4SA M_{sc} that works as M_{sr} with directions flipped around the main diagonal: instead of movements right, left, up, and down, it moves down, up, left, and right, respectively.

The aim is to infer some properties that could inform which of the strategies should be used for a given language.

When employing learning of finite automata using `traxbar`, the performance of the scanners, averaged over languages $L_1, \ldots, L_6$, can vary greatly (Figure 11), but there is no clear best or worst approach. Looking at each of the languages separately, we can observe similar behavior – see Figure 12 for the case of the picture language L_1. Note that in the left plot, Row-by-row and Snake-by-column – strategies that have the least in common – have mostly similar results.

This leads to the conclusion that the chosen scanning strategy can have a very significant impact on the performance of the model, but it can be hard to choose the correct one for a given task. The differences could be explained by the datasets being sampled in different ways leading to shifted differences in the classes becoming more apparent when we are scanning horizontally to when we are scanning vertically and vice versa, which makes a good future research topic.

From Figures 13 and 14, we can see two examples of how the scanning strategy does not have a significant impact on the performance of the k-locally testable languages. This is likely due to the relative simplicity of the languages (and symmetry for most of them); a different scanning strategy is not going to substantially change the substrings appearing in the transcribed language.

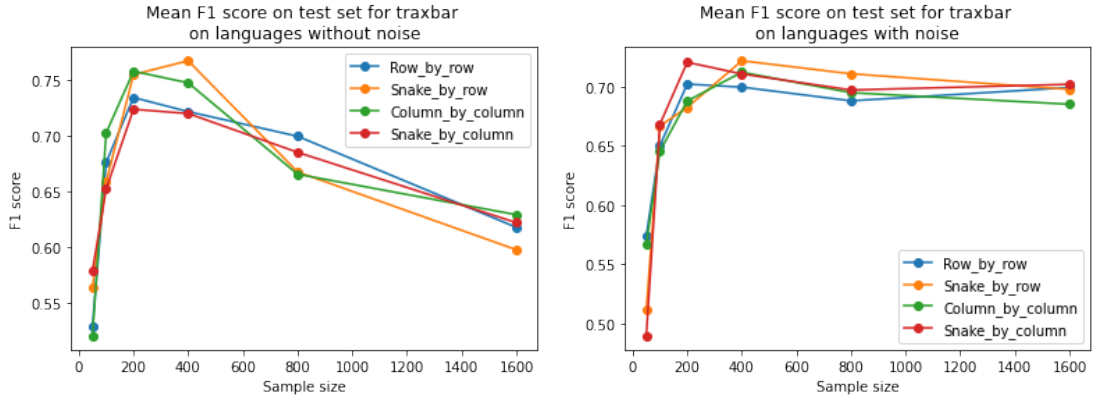


Figure 11: Comparing scanning strategies in learning picture languages by training finite automata using `traxbar` for crisp languages with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets on crisp languages $L_1, \ldots, l_6$ on the left and the average F_1-score on test sets on noisy languages on the right.

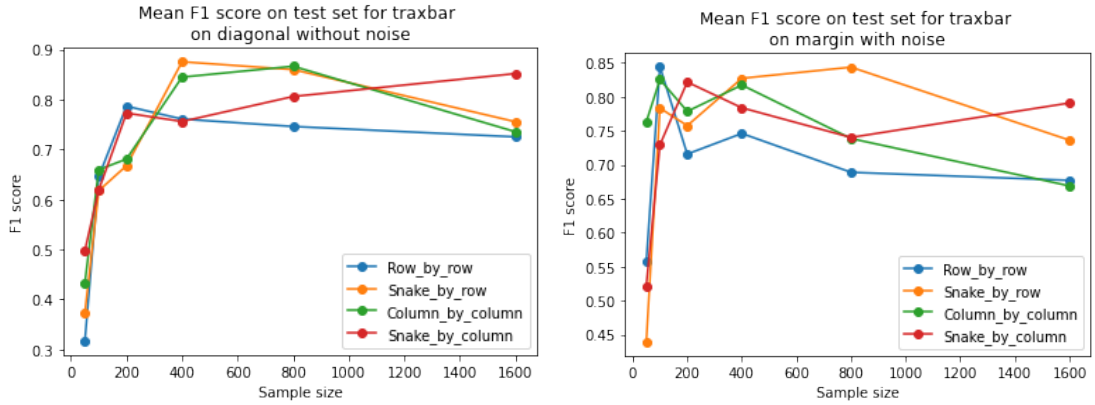


Figure 12: Comparing scanning strategies when learning picture languages by training finite automata using `traxbar` with the many-to-int encoding of the contents of the scanning window. The average F_1-score on test sets for the crisp language L_1 on the left and the average F_1-score on test sets for the noisy version of L_3 on the right.

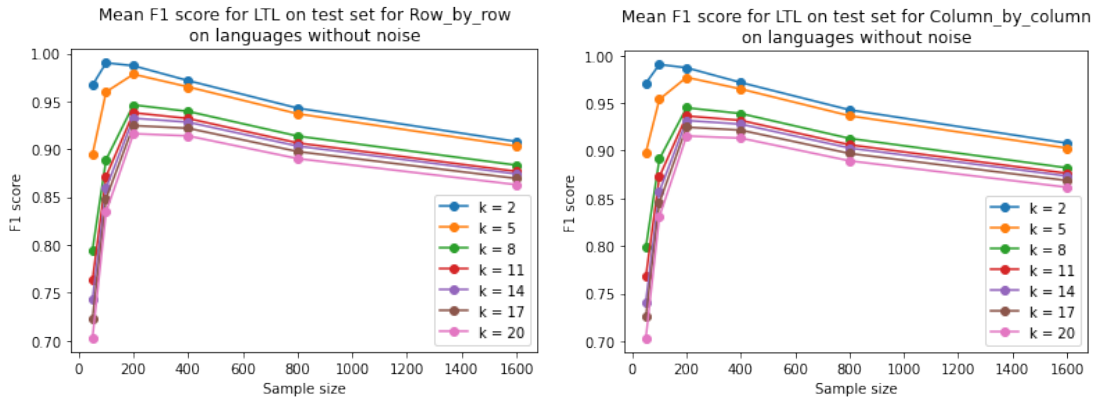


Figure 13: Results of training finite automata using k-locally testable languages for the crisp sample languages for two scanning strategies. The average F_1-score on test sets for Row-by-row on the left and Column-by-column the right.

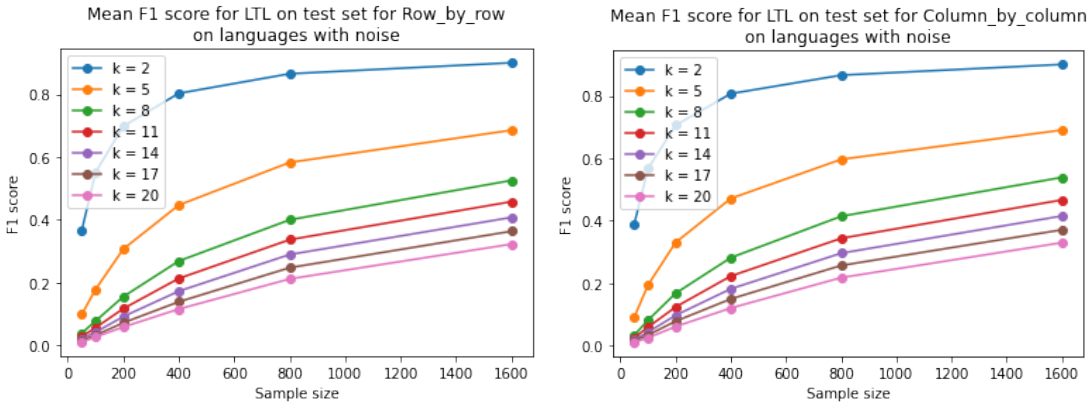


Figure 14: Results of training finite automata using k-locally testable languages for the noisy sample languages for two scanning strategies. The average F_1-score on test sets for the Row-by-row strategy on the left and Column-by-column strategy on the right.

4.6 Application on MNIST Dataset

Inspired by the high F_1-score when we applied 2-locally testable languages for training on strings obtained with many-to-int encoding, we decided to try the method on the known MNIST dataset[1] [10]. We prepared a training dataset consisting of 200 randomly selected pictures of handwritten digit one and 200 randomly selected pictures of other digits different from one. Similarly, we prepared a larger dataset consisting of 400 binarized pictures of digit one and 400 binarized pictures of other digits. Gray levels in the pixels of the original dataset were replaced by black and white pixels.

On these two datasets encoded with the many-to-int encoder, we applied `traxbar` and learning of locally testable languages. While the `traxbar` needed more than five hours to train on the second MNIST dataset, the training time for learning locally testable languages was under one second. In accordance with our experiments with the above sample languages, learning through locally testable languages achieved F_1-score more than 0.9 on the bigger MNIST dataset, while `traxbar` obtained F_1-score of only 0.45. The results are plotted in Figure 15.

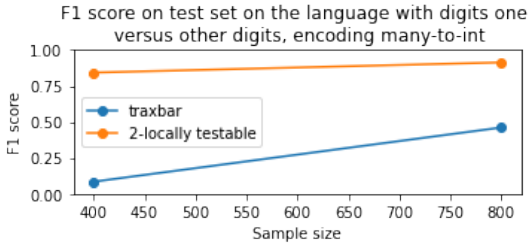


Figure 15: Results of training finite automata using `traxbar` and 2-locally testable languages for the two subsets of binarized MNIST.

4.7 Summary of Experiments

Overall, from the plots, we can see that even though the k-locally testable languages method works well on simple crisp languages, on smaller noisy datasets (up to size 400) it is outperformed by the state merging algorithms. Once the datasets for noisy languages become large enough, the state merging algorithms start to overfit, while the k-locally testable languages get enough data to improve their knowledge of the languages.

[1]The dataset is available from http://yann.lecun.com/exdb/mnist

5 Conclusion

We have proposed and formalized the transcription-evaluation framework, including the transcriptor-evaluator and the four-way scanner automaton. These help us solidify the theoretical foundations for our experiments and formalize further options for picture scanning and processing.

In the experiments, the focus was set on several aspects. First was the comparison of different alphabet sizes in the scanning dictionary (scanning window transcription). We found out that with `traxbar` the many-to-int encoding clearly outperforms one-to-one encoding, while with locally testable languages, the one-to-one encoding is perfectly sufficient.

The second aspect of interest was the behavior of the learning algorithms on our sample languages. The performance of `traxbar` with the many-to-int encoding on crisp languages varied significantly, but on languages with noise, it showed a decent performance with F_1-scores mainly in the 0.6 to 0.8 band.

As for the locally testable languages with the best-performing value of $k = 2$, the learning algorithm succeeded for all of them. However, the diagonal language without noise turned out to be slightly more difficult to learn than others.

We have learned that the selected scanning strategy can have a significant effect on the result, but it remains to be seen how that relates to language sampling or to learning algorithms.

Lastly, we wanted to verify whether learning automata from picture languages has reached a stage where it can be used for practical problems – in our case learning numbers from the MNIST dataset. Clearly, we did not expect the learned automata to compete with deep neural networks, but the obtained results offer a promise for further research of this and other practical problems.

Acknowledgements

This research was supported by the Charles University Grant Agency (GAUK) project no. 1198519 and SVV-260588.

References

[1] Gennaro Costagliola, Vincenzo Deufemia, Filomena Ferrucci, and Carmine Gravino. On regular drawn symbolic picture languages. *Information and Computation*, 187(2):209–245, 2003.

[2] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[3] Henning Fernau, Meenakshi Paramasivan, Markus L. Schmid, and Gnanaraj Thomas D. Simple picture processing based on finite automata and regular grammars. *Journal of Computer and System Sciences*, 95:232–258, 2018.

[4] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, EC-10(2):260–268, June 1961.

[5] Dora Giammarresi and Antonio Restivo. Two-dimensional languages. In *Handbook of Formal Languages: Volume 3 Beyond Words*, pages 215–267. Springer, Berlin, 1997.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Lukáš Krtek. Learning picture languages using restarting automata. Master thesis, Charles University, Faculty of Mathematics and Physics, 2014.

[8] David Kubon and Frantisek Mráz. Learning picture languages represented as strings. In Roman Barták and Eric Bell, editors, *Proceedings of the Thirty-Third International Florida Artificial Intelligence Research Society Conference*, pages 529–532. AAAI Press, 2020.

[9] Kevin J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *COLT 1992*, pages 45–52. ACM, 1992.

[10] Yann LeCun et al. Learning algorithms for classification: A comparison on handwritten digit recognition. In *Neural Networks: The Statistical Mechanics Perspective*, pages 261–276. World Scietific, 1995.

[11] Hermann A Maurer, Grzegorz Rozenberg, and Emo Welzl. Using string languages to describe picture languages. *Information and Control*, 54(3):155–185, 1982.

[12] Matteo Pradella and Stefano Crespi Reghizzi. A sat-based parser and completer for pictures specified by tiling. *Pattern Recogn.*, 41(2):555–566, February 2008.

[13] Daniel Průša, František Mráz, and Friedrich Otto. Two-dimensional sgraffito automata. *RAIRO-Theoretical Informatics and Applications*, 48(5):505–539, 2014.