Universiteit Leiden
The Netherlands

Opleiding Informatica

Probably Approximately Correct
Automata Extraction from Neural Networks

Lucas Zuurmond
3178250

Supervisors:
Marcello Bonsangue & Chase Ford

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/08/2024

Abstract

The L* algorithm constructs minimal deterministic finite automata (DFA) for regular languages through iterative membership and equivalence queries by treating the target language as a black-box. Recently, it has been employed to extract finite state models from recurrent neural networks (RNNs). This thesis integrates the L* algorithm with PAC (Probably Approximately Correct) learning to evaluate DFA extraction from RNNs trained on regular and context-free languages in a black-box setting. Our experiments show that while our algorithm accurately learns regular languages, it struggles with context-free languages due to limitations in approximating equivalence queries. The results highlight the potential and challenges of using L* and PAC for automata extraction.

Contents

1 Introduction

Neural networks are increasingly employed in safety-critical technologies, such as self-driving cars [2] and malware detection [21]. As such, the problem of interpretability becomes increasingly important. A survey from Zhang et al. [22] shows the importance of interpretability: "the requirement of high reliability systems, ethical/legal requirements and knowledge finding for science".
One promising approach to this problem is based on abstracting automata from *recurrent neural networks* (RNNs). As automata can be represented as a labelled graph encoding the initial, final and transition structures, it can be easier to interpret an automaton than an RNN. Several approaches have been considered for different types of automata such as *deterministic finite automata* (DFA), *residual finite-state automata* (RFSA) [5] and *weighted automata* (WA) [13]. These works often use L^*, introduced by Angluin [1], as a learning algorithm [11, 18] or an algorithm based on L^*. L^* is a learning algorithm that uses a *minimally adequate teacher* to learn an unkown regular language from queries and counterexamples in the form of a minimal DFA. This, however, comes with a few challenges.
For example, the equivalence queries employed by L^* are difficult to answer because we do not know the exact language from an RNN. For this reason, these queries are usually approximated. A survey from Bollig et al. [4] show several approaches to this, including PAC learning [14], statistical model checking [11] and abstraction refinement [18]. For a detailed discussion of the L^* algorithm, see Section 3.

Exact Learning

Exact learning is the process of precisely identifying an unknown target concept, such as a *regular language*. The aim is to identify a hypothesis that is close enough to the target language that it only makes small errors. This process involves interactions with a *teacher* that provides the necessary information to a learner to construct a hypothesis DFA that as close as desired to the target language at cost of sufficient data and computational resources. This process can be seen as a game between the teacher and the learner, where the learner keeps getting closer to the teacher until their languages are equivalent. A survey from Vaandrager [16] highlights the current state-of-the-art in the field of exact (model) learning. There are two primary approaches to exact learning:

Active Learning is the process of learning a regular language by interacting with the teacher by making specific queries to gain information about the target language.
For this, Angluin introduced the *minimally adequate teacher* (MAT) model, a teacher that answers queries from the learner with the minimal information required to learn the target language. This model can answer two types of queries:

- **Membership queries**: The learner asks the teacher whether a word belongs to the target language.
- **Equivalence queries**: The learner proposes a minimal DFA and asks the teacher if it matches the target language. If the hypothesis is incorrect, the teacher additionally provides a counterexample, a word on which the DFA and the target language disagree on, where the hypothesis and the target language differ.

The L^* algorithm, is a well-known example of active learning for regular languages. Several variations of this algorithm have been created to learn different types of automata, such as PDFA [19] and NFA [20].

Passive Learning is another approach to exact learning, where the model learns from a fixed set of data without interacting with a teacher. Unlike active learning, the model does not query for specific information but instead tries to infer the target concept solely from the provided examples. This method relies heavily on the quality and quantity of the initial dataset. Passive learning is very efficient in cases where the teacher does not know the exact concept and is therefore unable to answer equivalence queries as needed in active learning [10].
Neural networks are a common example of the type of models that are used in passive learning. These models are trained on large datasets where the classification of each input is known. Despite not querying a teacher, they can effectively learn complex patterns, and concepts, such as natural languages [8] and image recognition [6] from the data.
Another example of passive learning is DeLeTe2, an algorithm proposed by Denis et al.[5]. Unlike L*, DeLeTe2 learns an RFSA using passive learning. An RFSA is a special type of nondeterministic automaton that is expressively equivalent to a DFA. They share the feature that for every regular language, there exists a unique minimal automaton for the language. However, an RFSA can be exponentially more succinct than a DFA [3].

Active learning is efficient because it allows the learner to focus on the most informative examples, while passive learning is more suited for approximate learning as we only need data to be able to learn the concept and not a way to exactly answer queries.

Related Work

The extraction of finite automata from RNNs has been an area of active research [4]. While there are many types of neural networks, RNNs are particularly suited for DFA extraction as both are stateful models that can be viewed as language acceptors.
One of the most notable works in this field is by Weiss et al.[18], who demonstrated an effective method for extracting DFA from RNNs using abstraction refinement, a state clustering technique introduced by Omlin and Giles [15]. IThis method assumes acces to the internal states and structure of the RNN is assumed to be accessible, allowing for a precise mapping from the RNN's behavior to the automaton. However, this approach relies on having detailed access to the RNN's architecture, which may not always be feasible.
In our work, we want to adopt a blackbox approach to DFA extraction from RNN using PAC learning instead of state clustering. More specifically, we will interact with the RNN solely through its input-output behavior, without any knowledge of its internal states or structure. This also allows our method to be applied to other types of networks, as it only relies on the input/output. For the research, we set ourself the following question: How efficiently can we learn an automata from a neural network in a black box setting?
Despite the limitation, our approach achieves similar qualitative results as those achieved by Weiss et al. [18].

2 Preliminaries

In this section, we review the relevant background on RNN and DFA. While in this work later we focus on the state-of-the-art model LSTM [7], here we introduce RNN at an abstract level. We review a mathematical defenition of both DFA and RNN to show their similarities as stateful models that can be used as a language acceptor.

Languages

An *alphabet* is a nonempty finite set of symbols or characters called *letters*. A *word* is a sequence $a_1, \dots a_n$ of letters. We define the length of a word $w = a_1, \dots a_n$ as $|w| = n$ with the unique word of length 0 called the *empty word*, denoted by ε. The concatenation of two words v, w is denoted as vw. We define the set of all words of any length over an alphabet Σ as Σ^*. Any subset of Σ^* is called a *language* (over Σ). On a two languages L_1, L_2 we define the symmetric difference using the set difference:

$$L_1 \oplus L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1))$$

2.1 Deterministic finite automata

A *Deterministic Finite Automaton* (DFA) is a computational model that can be seen as a *language-acceptor*, it determines whether a given word is part of a language. A DFA is represented by a tuple

$$A = (\Sigma, Q, q_0, F, \delta)$$

where:

- Σ is a finite set of characters, known as the alphabet.
- Q is a finite set of *states*.
- $q_0 \in Q$ is the *initial state*.
- $F \subseteq Q$ is the set of *final (accepting) states*.
- $\delta : Q \times \Sigma \to Q$ is the *transition function*, which takes a state and a character and returns the next state.

The transition function δextends to a map to operate on words instead of just letters $\delta^* : Q \times \Sigma^* \to Q$. This is defined recursively as follows:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, \sigma w) = \delta^*(\delta(q, \sigma), w)$$

where $\sigma \in \Sigma, w \in \Sigma^*$.

A DFA accepts a string if, after reading the entire string from the initial state q_0, the automaton ends in one of the final states in F. Otherwise, it rejects the string. We use this for our classification function $f_A : \Sigma^* \to \{Accept, Reject\}$ that is defined on A as follows:

$$f_A(w) = \begin{cases} Accept & \text{if } \delta^*(q_0, w) \in F, \\ Reject & \text{otherwise} \end{cases}$$

As DFAs represent regular languages, we want to be able to talk about the languages they recognize. For a given DFA A, the language it recognizes is defined by

$$L(A) = \{w \in \Sigma^* \mid f_A(w) = Accept\}$$

We say a DFA A is minimal if of all DFAs representing the same language $L(A)$, none of them have less states than A. In other words, A has the minimal required states to represent the language $L(A)$.

Important to note is that not every language can be represented with a DFA. Some languages, such as the set of palindromes over $\Sigma = \{a, b\}$ need a stack-like stucture to maintain information. We call languages that can be represented by a DFA *regular languages*. Languages that need a stack-like structure are called *context-free*.

For convenience, DFAs are often represented using a graph. This shows all information of a DFA and allows for interpretation. The states are represented with a circle each, with final states a double circle. The transition function is represented with labeled arrows in between states and the initial state is shown using an arrow going into the first state.

Examples of minimal DFAs and their representation

In this section, we give a few minimal automaton for each of the following languages on a two letter alphabet $\Sigma = \{a, b\}$:

- a^*: The language of words containing only a's.
- $(ab)^*$: The language of words containing only the subword ab repetetively
- $xABBxBABx$: The language of words containing the subword abb and later the subword bab.

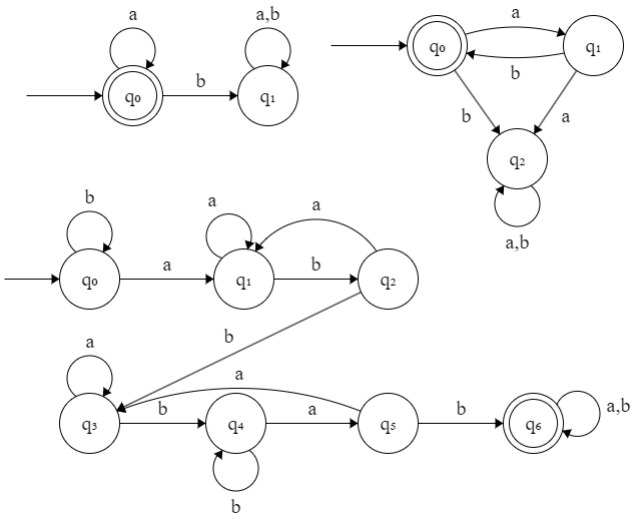

Figure 1: Graphical representation of a^* (top-left), $(ab)^*$ (top-right) and $xABBxBABx$ (bottom)

2.2 Recurrent neural networks

A Recurrent Neural Network (RNN) is a type of neural network particularly well-suited for processing sequential data. It uses a *recursive layer* to repeatedly process new input vectors. Concretely, an RNN is a tuple $\mathcal{R} = (m, n, f, g, h_0)$ where:

- m is the dimension of the state space $\mathbb{R}^m$, equal to the amount of nodes in the recursive layer
- n is the dimension of the input vector space $\mathbb{R}^n$
- $f : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$ is a function that represents the recursive layer of the network
- $g : \mathbb{R}^m \to [0, 1]$ is a function that represents the final layer of the network
- $h_0 \in \mathbb{R}^m$ is the initial state vector

An RNN takes an input vector $x_t \in \mathbb{R}^n$ at time step t and a state vector $h_t \in \mathbb{R}^m$ (which captures information from previous time steps) and returns a new state vector $h_{t+1} \in \mathbb{R}^m$. The update rule for the state vector is given by the function:

$$h_{t+1} = f(h_t, x_t)$$

Here, we typically view the inputs of neural networks as a word over a fixed alphabet, so we use the notation $x = (x_1, \dots x_n)$ to represent it as a finite sequence of vectors. We call the space of these sequences S, with the unique sequence of length 0 denoted by Λ.
By recursively applying the update function f on every input vector $x_1, \dots x_n$ and using each output for the next input, we get a sequence of state vectors $(h_1, h_2, \dots h_n)$. The final state vector h_n encapsulates information about the entire input sequence. In a similar way to DFA's, we extend the update function to the function $f^* : \mathbb{R}^m \times S \to \mathbb{R}^m$ the following way:

$$f^*(h_t, \Lambda) = h_t$$
$$f^*(h_t, (x_t, \dots x_n)) = f^*(f(h_t, x_t), (x_{t+1}, \dots x_n))$$
$$h_n = f^*(h_0, x)$$

In this research, we focus on learning regular languages, so our input sequence will consist of characters from a predefined set (alphabet). These characters are typically mapped to vectors using *one-hot encoding*, where each character in the alphabet is represented by a unique binary vector.
To classify the output of the RNN, we define a classification function $f_R : h_n \to \{Accept, Reject\}$. This is implemented using the final network layer that maps the final state vector h_n to a single value between 0 and 1, which can then be thresholded to decide acceptance or rejection. Formally, we can define:

$$f_{\mathcal{R}}(h_n) = \begin{cases} \text{Accept} & \text{if } \phi(g(h_n)) \geq \alpha, \\ \text{Reject} & \text{otherwise} \end{cases}$$

where ϕ is the activation function, which squashes the output to a range between 0 and 1 and α is the acceptance threshold, which is usually 0.5.
We define the language accepted by an RNN $\mathcal{R}$ as

$$L(\mathcal{R}) = \{w \in \Sigma^* \mid f_{\mathcal{R}}(f^*(h_0, w)) = \text{Accept}\}$$

3 *L** algorithm

As mentioned in the introduction, L^* is an exact learning algorithm for inferring an unknown regular language from equivalence queries and counterexamples. This algorithm learns a regular language in the form of a minimal DFA through interaction with a minimally adequate teacher. The goal of the L* algorithm is to construct a minimal DFA that accurately represents the target language.
In this work, we use a trained RNN as a teacher for L^*. The RNN can trivially answer membership queries by feeding the string to the RNN for classification. Answering equivalence queries, however, is more challenging and typically involves a process of approximation. One approach from Weiss et al.[18] uses abstraction refinement, based on a clustering algorithm from Omlin and Giles [15]. This method clusters different states of the RNN together to build up a DFA that represents the network and compares that to the hypothesized DFA. This, however, requires access to the RNN's internal states and can therefore only be used in settings with access to this information. Here, we assume a black box setting where the only interaction with the RNN is by observing its input-output behaviour. We address the equivalence queries using Probably Approximately Correct (PAC) learning methods, this is explained later in subsection 4.4.

3.1 Algorithm details

We first introduce first introduce a few concepts used by L*. Suppose L in the target language. The L* algorithm operates by keeping track of two sets:

- A set of *access words* Q, with $\varepsilon \in Q$.
- A set of *test words* T.

The set Q is used to store the different states for a hypothesized DFA and the set T is used to distinguish words in Q. To do this, we say two words $v, w \in \Sigma^*$ are T-*equivalent*, denoted $v \equiv_T w$ if

$$vu \in L \iff wu \in L \quad \text{for all } u \in T.$$

Note that this can be checked with $2 \cdot |T|$ membership queries by querying vu and wu for all $u \in T$. We are going to be interested in the following conditions on the set of access words:

- **Separability:** No two different words in Q are T-equivalent.
- **Closedness:** For every $q \in Q, a \in \Sigma$ there exists $q' \in Q$ such that $qa \equiv_T q'$

If Q and T are both separable and closed, we can create a *Hypothesis DFA* $\mathcal{H}(Q, T)$ as followed: Every element in Q becomes a state, since Q is separable, all states are unique. We use $\varepsilon \in Q$ as the initial state. Since Q is closed, there is a unique (up to T-equivalence) state $\delta(q, a) \in Q$ such that $qa \equiv_T \delta(q, a)$. This yields a function that sends every $q \in Q, a \in \Sigma$ to a $q \in Q$. We use this as the transition function. Then, we evaluate every state to find all the accepting states $F = \{q \in Q \mid q \in L\}$. Note that this can be done by using a membership query for every $q \in Q$.

Using these definitions and method to construct a hypothesis, we can define our algorithm, split into 5 steps. For a more detailed breakdown, see the pseudocode below. Line numbers added to indicate where each step is located.

1. (line 21-22) Initialize $Q, T = \{\varepsilon\}$, by definition this is separable and closed.
2. (line 24-28) Using membership queries, repeatedly enlarge Q such that Q remains separable until Q and T are separable and closed.
3. (line 30-31) Compute the hypothesis DFA $\mathcal{H}$ and make an equivalence query.
4. (line 33-34) If answer is yes, terminate algorithm
5. (line 35-47) If answer is no, use membership queries to expand Q and T such that they remain separable, then go back to step 2

To reason about the correctness of the algorithm, we use 3 propositions:

First, we are going to show that any correct DFA hypothesised by the algorithm is minimal. We do this by showing that any DFA for the target language L must have at least $|Q|$ states. As any hypothesised DFA $\mathcal{H}(Q, T)$ has exactly $|Q|$ states, it is minimal.

Proposition 1. If Q and T are separable, then $|Q|$ is at least the number of states of a minimal DFA for L.

Proof. Using the separability of Q and T we know that any two words $q_1, q_2 \in Q$ can always distinguished by T as there exists some $u \in T$ where $f_A(q_1 u) \neq f_A(q_2 u)$. This means that any DFA for L needs at least $|Q|$ states, and thus $|Q|$ is at least the number of states of a minimal DFA for L. □

Proposition 2. If Q is T-separable and not T-closed, then there exists a T-separable extension of Q that we can find using membership queries. If Q is T-separable and T-closed, we are done. So we assume that Q is not T-closed.

Proof. Since Q is not T-closed, there exists $q \in Q$ and $a \in \Sigma$ such that qa is not T-equivalent to any $q' \in Q$. As both Q, Σ are finite, we can find qa using only membership queries by checking T-equivalence for every qa and q. By construction, qa is not T-equivalent to any $q \in Q$, therefore by adding qa to Q, Q remains T-separable.

□

Next, we are going to show that whenever $\mathcal{H}(Q, T)$ admits a counterexample w, we can extend Q and T according to the target language. We do this by finding the first point in the counterexample at which point the language disagrees with the hypothesis and adding that as a new state to Q. For this, we define the states that w passes through: for $i = 1, \dots n$, we define $q_i = \delta^*(q_0, w_1, \dots w_i)$. Then, we define the *correctness* of these states: after reading up until w_i we end up in the state q_i, we concatenate the word representing q_i and the remainder. We say that q_i is correct if this concatenation and w have the same classification in L. Formally we define q_i to be correct when

$$q_i w_{i+1}, ...w_n \in L \iff w \in L$$

Note that the correctness can be checked using a membership query for w and $q_i w_{i+1}, ...w_n$.

Proposition 3. Suppose Q and T are separable and closed and let $\mathcal{H}$ be the hypothesis DFA. Given a counterexample $w = w_1, \dots w_n \notin L(\mathcal{H}(Q, T))$, using membership queries we can find $q \in \Sigma^*$ and $t \in \Sigma^*$ such that $Q \cup \{q\}$ and $T \cup \{t\}$ is separable.

Proof. Obviously $q_0 = \varepsilon$ is correct as the remainder is the entire counter example ($q_0 w_1, \dots w_n = w$) and q_n is incorrect as it is the state that w ends in and by definition has a different classification as w. Using membership queries, we can find the first incorrect q_i by checking the correctness of each q_i for $i = 1, \dots n$.

Now we define $Q' = Q \cup \{q_i w_i, \dots w_n\}$ and $T' = T \cup \{w_{i+1}, \dots w_n\}$.
By definition of the transition function of $\mathcal{H}(Q, T)$, q_i is T-equivalent to $q_{i-1} w_i$. T-separability of Q then says q_i is not T-equivalent to any $q \in Q \setminus \{q_{i-1} w_i\}$.
At the same time $w_{i+1}, \dots w_n \in T'$ distinguishes q_i and $q_{i-1} w_i$ by construction of q_i, so we conclude that Q' and T' are separable.

□

Note that these propositions all rely on T-separability. By definition, Q is T-separable at the start. As any modifications done to Q and T are done through propositions 2 or 3 and Q remains T-separable in these propositions, Q and T admit seperable extensions at all finite stages of the algorithm execution.

The function Find_T_equivalent attempts to find a $q' \in Q$ which is T-equivalent to q_a. This is used to determine whether Q is T-closed and when creating a hypothesis DFA $\mathcal{H}(Q, T)$ to define the transition function.

Algorithm 1 L^* Algorithm for Learning a DFA

function FIND_T_EQUIVALENT(T, Q, q_a)
    for all $q' \in Q$ **do**
        if $q_a \equiv_T q'$ **then**
            return True, q'
        end if
    end for
    return False
end function

function IS_CLOSED(Q, T)
    for all $q \in Q$ **do**
        for all $a \in \Sigma$ **do**
            if not FIND_T_EQUIVALENT(T, Q, $q + a$) **then**
                return False, q, a
            end if
        end for
    end for
end function

function LSTAR(Σ, member, equiv)
    Initialize $Q \leftarrow \{\varepsilon\}$
    Initialize $T \leftarrow \{\varepsilon\}$
    while True **do**
        $res \leftarrow$ IS_CLOSED(Q, T)
        while not res **do**
            $Q \leftarrow Q \cup \{q_{res} + a_{res}\}$
            $res \leftarrow$ IS_CLOSED(Q, T)
        end while

        $H \leftarrow$ CREATE_HYPOTHESIS(Q, T)
        $res \leftarrow$ EQUIV(H)

        if res **then**
            return H
        else
            $w \leftarrow cex_{res}$
        end if

        for $i = 0$ to w **do**
            $qi \leftarrow \delta_H^*(w[:i])$
            if MEMBER(w) == MEMBER($qi + w[i:]$) **then**
                continue
            end if
            $Q \leftarrow Q \cup \delta_H^*(w[:i-1]) + w[i]$
            $T \leftarrow T \cup w[i:]$
            break
        end for
    end while
end function

4 Experiment setup

In this section, we explain the processes taken to setup the experiments. This includes data preparation as well as training of the RNNs. Last, we also talk about our approach to answering the queries made by L*.

4.1 Test languages

In our experiments, we focus on learning regular and context-free languages, as this allows us to look at both simple and complex networks. We consider only languages over the alphabet $\Sigma = \{a, b\}$ apart from language 6. Some of the languages that we test on have already been described in Section 2.1.

Regular languages

Language 1: a^* As described in Section 2.1, this language consists of all words composed entirely of the character a, including the empty word. This is a baseline for evaluating the performance of L*.

Language 2: $(ab)^*$ The language $(ab)^*$ comprises all words that are repetitions of the subword ab, including the empty word. This language is slightly more complex than a^*, as it involves a specific pattern within the strings.

Language 3: $xABBxBABx$ The language that consists of words with the subword abb followed by the subword bab. This language is a lot more complex than the other regular languages, as it requires 7 states for a minimal DFA. This shows that L* can also learn more difficult DFA.

Context-free languages

Language 4: XX^R The language XX^R represents the set of palindromes over the alphabet $\{a, b\}$. A string belongs to this language if it reads the same forwards and backwards. This language is context-free and significantly more complex than the regular languages, as it requires maintaining a stack-like structure as long as the length of X to recognize palindromes, which is impossible in a DFA because it has only a fixed finite number of states independent from the length of the input.

Language 5: A^nB^n The language A^nB^n consists of strings that have n occurrences of the character a followed by n occurrences of the character b, for any non-negative integer n. This is a classic example of a context-free language, as it cannot be recognized by a finite automaton but can be recognized by a pushdown automaton.

Language 6: counting To show that the algorithm also works with larger alphabets, which would be closer to a real world scenario, we are testing on a language with alphabet $\Sigma = \{a, b, \dots z, 1, \dots 5\}$. Specifically, the language consists of words with the sequence 12345 with the letters $a - z$ scattered inside them.

4.2 Data collection

Now that we have the languages we are testing on, we need to generate the data to train the networks on and use for DFA extraction for each of the languages. This data is divided into two types: training data and test data. The training data is used for training the RNNs and example generations for PAC. Test data is used for measuring the performance of RNNs and DFA.

Training data

The training datasets were created by uniform randomly generating 20000 words uwith a maximum length of 50 letters. These words were then classified using a *regular expression pattern* as either positive or negative examples of the target language. To ensure a sufficient number of examples, the ratio of positive to negative samples was limited to 1:50 by forcing a string of the opposite classification to be generated if the ratio was too large. This imbalance is aimed to reflect the lower density of positive samples in the target languages, but also leaves enough room for positive patterns to be included, thus there are at least 4000 positive samples.

Test data

The test datasets were constructed to provide a comprehensive evaluation of the trained models. For each length up to 50 characters, we generated 100 uniform random positive and 100 uniform random negative samples, removing all duplicates. In languages 1, 2 and 5 there is only a maximum of 1 positive sample for each length. So the positive examples is limited to 50. To account for this, we also limited the negative samples to 1 per size to keep it balanced, resulting in a test size of 100.

Preprocessing

Before training the networks, all examples were padded with a special character to a length of 50 to accommodate the varying word lengths. Each character in the strings was then transformed into an input vector using one-hot encoding. Specifically, the character a was encoded as $[1, 0, 0]$, b as $[0, 1, 0]$ and the padding character as $[0, 0, 1]$. For language 6, this is extended to the size of the new alphabet.

4.3 Training the RNN

With our data formatted, the next step was to train the RNNs. We used Long Short-Term Memory (LSTM) [9] networks due to their effectiveness in handling sequences with long-term dependencies, such as regular languages. Our goal is to train every network up to 100% accuracy as we want to focus on the abstraction of DFA's.

For all languages, we used $m = 100$ for the dimension of the state space. The dimension of the input space for languages 1-5 is 3 and for language 6 it is 32, matching the size of the alphabets used for the language +1 for the padding character.
We use the sigmoid activation function and apply a dropout rate of 0.5 to prevent overfitting. Because the networks do binary classification, we used binary cross-entropy (BCE) as a loss function. We used the Adam optimizer [12] with a learning rate of 0.001.

We attempted to train every network as close to 100% accuracy on the train set. Due to overfitting, we could not get all of them to 100%. After, we evaluated all of them on the test set. The following are the performances of the networks on the generated test sets:

Language	Test size	TP	FP	TN	FN	Accuracy	Precision	F1-Score
1	100	50	1	49	0	99.0%	98.04%	99.01%
2	74	26	0	38	0	100%	100%	100%
3	8558	4100	19	4400	39	99.32%	99.54%	99.3%
4	8313	3860	17	4395	41	99.3%	99.56%	99.25%
5	77	26	1	50	0	98.7%	96.3%	98.11%
6	8928	4440	3	4450	35	99.57%	99.93%	99.57 %

4.4 Answering Equivalence Queries with Probably Approximately Correct Learning

While answering membership queries made by L^* applied to learn a neural network is simple as we can ask the RNN for classification, equivalence queries from L^* do not have a straightforward method for exact answers. To address this, we use an approach proposed by Mayr and Yovine [14], which uses Valiant's model of *Probably Approximately Correct (PAC) learning* [17].
PAC learning is an approximate learning method typically employed when equivalence queries are not easily answerable. The idea is to replace a single equivalence query with a series of membership queries on semi-randomly selected words. If the hypothesis DFA disagrees with any of these words, it is returned as a counterexample. If the hypothesis agrees with a sufficiently large number of these words, we can assert with high confidence that the DFA will only make minor mistakes.
While this approach does not guarantee that the DFA is entirely correct, it ensures a high probability of accuracy within a specified margin of error. The number of membership queries needed for a single hypothesis depends on various parameters, such as the desired confidence level and the maximum allowed error rate.

Mathematical Definition

To make the above idea work, PAC assumes a probability distribution $\mathcal{D}\colon \Sigma^* \to [0,1]$. This distribution allows us to determine the probability that the language of a DFA A differs from the language of an RNN $\mathcal{R}$ by summing all the words where the two disagree:

$$\mathbb{P}_{\mathcal{D}}(A \text{ differs from } \mathcal{R}) = \sum_{w \in L(A) \oplus L(\mathcal{R})} \mathcal{D}(w)$$

If this is less than our error parameter $\varepsilon \in (0,1)$, we can say that A is ε-*approximately correct*. This means that a randomly chosen word w from our probability distribution $\mathcal{D}$ has a probability of less than ε that it differs A from $\mathcal{R}$.
By fixing an additional confidence parameter $\gamma \in (0,1)$, we can replace an equivalence query with a series of randomly generated membership queries that is large enough to conclude that A is approximately correct with a confidence of $1-\gamma$. (in $1-\gamma$ of cases A is approximately correct)
As the automata hypothesized by L^* increase in size with each equivalence query, the number of membership queries required must also increase to maintain the same level of confidence. This

growth ensures that the larger hypothesis spaces are adequately tested. The formula [17] for the number of membership queries r required for the i-th equivalence query is:

$$r_i = \left\lceil \frac{1}{\varepsilon}(i \cdot \ln 2 - \ln \gamma) \right\rceil$$

While the actual distribution $\mathcal{D}$ does not affect the theoretical correctness of PAC, it does influence the learned DFA. A common approach is to use a combination of two distributions: one for the length of the word, often modeled with a negative exponential distribution, and another to fill each position in the word with a random character, typically using a uniform distribution. This approach works well in many cases but has a notable limitation: if certain types of words are too uncommon and do not appear in the membership queries, the DFA might fail to learn these types. For example, if words of length 19 $\mod 20$ have a special property but these words are rarely or never sampled, the algorithm will not learn this property.
Additionally, there is a risk that the algorithm may not terminate, particularly if the target language is not regular. This issue can be mitigated by imposing limits, such as the maximum number of equivalence queries or the maximum size of the DFA. We address this by limiting the runtime of L* to 5 minutes.

5 Experimental results

Initial Experiment

Initially, we used confidence $\gamma = 0.05$ and error rate $\varepsilon = 0.01$ as parameters for PAC, see section 4.4 for more details about this. We chose these values as we thought it would be a good default that would work on simple languages, but not quite at more difficult langues. It is worth noting that the target for the abstraction is the network and not the original language.
With these settings, L* was able to learn languages 2 perfectly, returning a correct minimal DFA representing the network. For languages 1 and 3, however, the abstraction terminated after finding a minimal DFA for the original languages. As the networks make mall mistakes, the abstracted DFA do not exactly represent the networks, but they still fall within the error rate ε that is fixed at 1%. The algorithm learning these mistakes seems inconsistent as that exact example has to show up in one of the equivalence queries. Intrestingly however, this could also be used to find examples that the network classifies incorrectly.
Even though language 6 is also regular, L* was not able to learn language 6, as the DFA had a small error that PAC did not pick up on. Specifically, upon encountering a number that would not follow according to the pattern, it would return to the initial state rather than immediately rejecting the word, allowing for words that contain '12345' after this to still be accepted. Despite not learning this pattern, the hypothesis DFA still falls within the 1% error rate that we fixed as the pattern occurs in only 0.2% (24 of 8928) of the examples, so it is still approximately correct.

Context-Free Language

For the context-free languages 4 and 5, L* terminated after 96 and 31 approximate equivalence queries, respectively. This was because the hypothesized DFAs were accurate up to a certain point,

but due to the low number of membership queries with PAC (6954 and 2449 respectively), it was unable to find a counterexample. Other runs of L* resulted in similar numbers of equivalence queries before stopping.
We found that decreasing the error rate ε to 0.0001 and the confidence γ to 0.01 caused L* to not terminate within the time limit of 5 minutes. The DFAs obtained from L* generally classified words of shorter lengths correctly, but for longer words, they rejected most of the strings. This behavior is observed for language 4 in Figure 2.

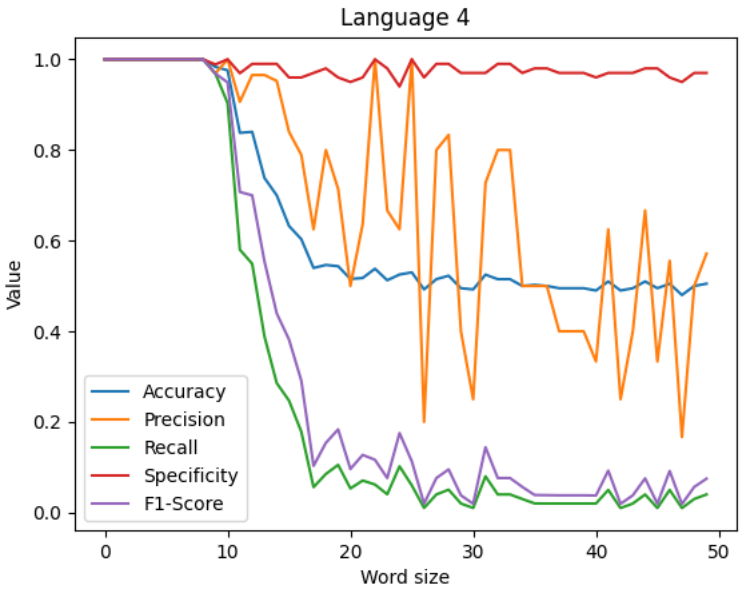

Figure 2: Metrics of Language 4 based on word size

After seeing the falloff in accuracy between word sizes 9 to 20 in the figure above, we tried to change the time limit of L* to see how this would influence the falloff. We found that it goes up slowly, but after 50 seconds, the differences became negligible. Even though the number of queries kept growing, the falloff start seemed limited to 11. This behavior is illustrated in Figure 3. This could be due to the nature of the language or the way our training set was generated. It could also be that it does keep growing, but it is not feasible to test as it would take too long before

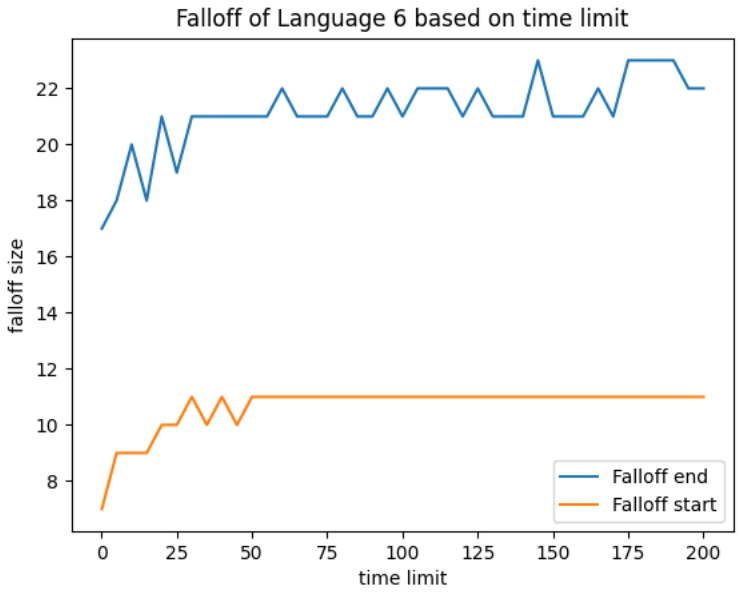

Figure 3: Falloff based on runtime of L* on language 4

Language 6

While the initial run of L* did learn an approximately correct DFA, we looked further into this experiment and tried another run. As the algorithm is probabilisitic, it might provide a different result. The second time time, the error mentioned above did not happen and the algorithm provided a DFA that perfectly matched the original language. Following this, we decided to run L* 200 times on language 6 and in 119 (59.5%) of the runs, the abstracted DFA matched the original language. In 5 of these runs, the algorithm learned at least one of the words that the network had incorrectly learned. When we modified the confidence parameter γ to 0.01, none of the runs had the error and 8 DFA had learned a mistake in the RNN.

Further Research

The results from our experiments indicate several directions for further research. Most notably looking into finding examples on which the network is trained incorrectly. We have seen that often a DFA will not learn such examples and this could be used to find examples where a network makes a mistake.
Another interesting point is seeing how changing the size of the training set, particularly changing the word size limit, would affect the learning process of context-free languages as this could potentially allow a higher variety of words and maybe increase the falloff start.

References

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[3] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *IJCAI*, volume 9, pages 1004–1009, 2009.

[4] Benedikt Bollig, Martin Leucker, and Daniel Neider. A survey of model learning techniques for recurrent neural networks. *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, pages 81–97, 2022.

[5] François Denis, Aurélien Lemay, and Alain Terlutte. Learning regular languages using rfsas. *Theoretical computer science*, 313(2):267–294, 2004.

[6] Michael Egmont-Petersen, Dick de Ridder, and Heinz Handels. Image processing with neural networks—a review. *Pattern recognition*, 35(10):2279–2301, 2002.

[7] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[8] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[10] Sanjay Jain and Frank Stephan. *Query-Based Learning*, pages 820–822. Springer US, Boston, MA, 2010.

[11] Igor Khmelnitsky, Daniel Neider, Rajarshi Roy, Xuan Xie, Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Martin Leucker, and Lina Ye. Property-directed verification and robustness certification of recurrent neural networks. In *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*, pages 364–380. Springer, 2021.

[12] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Clara Lacroce, Prakash Panangaden, and Guillaume Rabusseau. Extracting weighted automata for approximate minimization in language modelling. In *International Conference on Grammatical Inference*, pages 92–112. PMLR, 2021.

[14] Franz Mayr and Sergio Yovine. Regular inference on artificial neural networks. In *Machine Learning and Knowledge Extraction: Second IFIP TC 5, TC 8/WG 8.4, 8.9, TC 12/WG 12.9 International Cross-Domain Conference, CD-MAKE 2018, Hamburg, Germany, August 27–30, 2018, Proceedings 2*, pages 350–369. Springer, 2018.

[15] Christian W Omlin and C Lee Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural networks*, 9(1):41–52, 1996.

[16] Frits Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[17] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[18] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pages 5247–5256. PMLR, 2018.

[19] Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. *Advances in Neural Information Processing Systems*, 32, 2019.

[20] Takashi Yokomori. Learning non-deterministic finite automata from queries and counterexamples. In *Machine Intelligence 13*, pages 169–189. Citeseer, 1992.

[21] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 371–372, 2014.

[22] Yu Zhang, Peter Tiňo, Aleš Leonardis, and Ke Tang. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(5):726–742, 2021.