Machine Learning (2024) 113:2877–2919
https://doi.org/10.1007/s10994-022-06163-2

Extracting automata from recurrent neural networks using queries and counterexamples (extended version)

Gail Weiss[1] · Yoav Goldberg[2] · Eran Yahav[1]

Received: 11 April 2020 / Revised: 8 November 2021 / Accepted: 18 December 2021 /
Published online: 8 June 2022

Abstract

We consider the problem of extracting a deterministic finite automaton (DFA) from a trained recurrent neural network (RNN). We present a novel algorithm that uses exact learning and abstract interpretation to perform efficient extraction of a minimal DFA describing the state dynamics of a given RNN. We use Angluin's L* algorithm as a learner and the given RNN as an oracle, refining the abstraction of the RNN only as much as necessary for answering equivalence queries. Our technique allows DFA-extraction from the RNN while avoiding state explosion, even when the state vectors are large and fine differentiation is required between RNN states. We experiment on multi-layer GRUs and LSTMs with state-vector dimensions, alphabet sizes, and underlying DFA which are significantly larger than in previous DFA-extraction work. Aditionally, we discuss when it may be relevant to apply the technique to RNNs trained as language models rather than binary classifiers, and present experiments on some such examples. In some of our experiments, the underlying target language can be described with a succinct DFA, yet we find that the extracted DFA is large and complex. These are cases in which the RNN has failed to learn the intended generalisation, and our extraction procedure highlights words which are misclassified by the seemingly "perfect" RNN.

Keywords Recurrent neural networks · Automata · Deterministic finite automata · Exact learning · Extraction

Editors: Olgierd Unold, François Coste, Colin de la Higuera.

✉ Gail Weiss
sgailw@cs.technion.ac.il

Yoav Goldberg
yogo@cs.biu.ac.il

Eran Yahav
yahave@cs.technion.ac.il

[1] Technion, Haifa, Israel

[2] Bar Ilan University, Ramat Gan, Israel

1 Introduction

In recent years, there has been significant interest in the use of neural models, and in particular recurrent neural networks (RNNs), for learning languages. Like other supervised machine learning techniques, RNNs are trained based on a large set of examples of the target concept.

RNNs can reasonably approximate a variety of languages, and even precisely represent a regular language (Casey 1998). However, they are in practice unlikely to generalise exactly to the concept being trained, and what they eventually learn in actuality is unclear (Omlin&Giles 2000). Indeed, several lines of work attempt to glimpse into the RNN black-box (Zeng et al. 1993; Omlin&Giles 1996; Cechin et al. 2003; Jacobsson 2005; Karpathy et al. 2015; Li et al. 2015; Linzen et al. 2016; Strobelt et al. 2016; Lei et al. 2016; Kádár et al. 2016; Shi et al. 2016; Adi et al. 2016; Murdoch&Szlam 2017; Wang et al. 2017; Arras et al. 2017).

In contrast to the supervised ML paradigm, the *exact learning* paradigm considers setups that allow learning a target language without approximation. For example, Angluin's L^* algorithm enables the learning of any regular language, provided a *teacher* capable of answering *membership* (request to label example) and *equivalence* (comparison of proposed language with target language) queries is available (Angluin 1987).

In this work we use exact learning to elicit the true concept class of a trained recurrent neural network. This is done by treating the trained RNN as the teacher of the L^* algorithm. To the best of our knowledge, this is the first attempt to use exact learning with queries and counterexamples to extract an automaton from a given RNN.

Recurrent neural networks Recurrent neural networks (RNNs) are a class of neural networks which are used to process sequences of arbitrary lengths. When operating over sequences of discrete alphabets, the input sequence is fed into the RNN on a symbol-by-symbol basis. For each input symbol the RNN outputs a *state vector* representing the sequence up to that point, combining the current state vector and input symbol at every step to produce the next one. An RNN is essentially a parameterised mathematical function that takes as input a state vector and an input vector, and produces a new state vector. The RNN is trainable, and, when trained together with a *classification component*, the training procedure drives the state vectors to provide a representation of the prefix which is informative for the classification task being trained.

Classification An RNN can be paired with a *classification component*, a classifier function that takes as input a state vector and returns a binary or multi-class classification decision. The RNN and the classifier are combined by applying the RNN to the sequence, and then the classifier to the final resulting state vector. When the classification component gives a binary classification for each state vector, the combination defines a binary classifier over sequences, which we call an *RNN-acceptor*. When the component gives a distribution over the possible next tokens, the combination defines a next-token distribution for each input sequence, which we call a *Language-Model RNN (LM-RNN)*.

A trained RNN-acceptor can be seen as a state machine in which the states are high-dimensional vectors: it has an initial state, a well defined transition function between internal states, and a well defined classification for each internal state. A trained LM-RNN is not immediately analogous to a binary state machine, but we will see in this work how it may be interpreted as a one, and under this interpretation also extracted from using our method.

RNNs play a central role in deep learning, and in particular in natural language processing. For more in-depth overview, see (Goodfellow et al. 2016; Goldberg 2016, 2017).

We now turn to the question of understanding what an RNN has actually learned. We formulate the question around RNN-acceptors, but later (in Sect. 8) show how the solution relates to LM-RNNs.

Motivation Given an RNN-acceptor R trained over a finite alphabet Σ, our goal is to extract a deterministic finite-state automaton (DFA) A that classifies sequences in a manner observably equivalent to R. (Ideally, we would like to obtain a DFA that accepts *exactly* the same language as the network, but this is a much more difficult task.[1])

Note In this work, when understood from context, we use the term RNN to mean RNN-acceptor. Additionally, we use "automata" to refer specifically to deterministic finite automata (DFAs) (as opposed to other automata variants, such as pushdown automata or weighted automata).

Previously existing techniques for DFA extraction from recurrent neural networks are based on creating an a-priori partitioning of the RNN's state space, and mapping the transitions between the resulting clusters (e.g., Omlin&Giles (1996); Zeng et al. (1993)). In this work however, we approach the question using exact learning.

Exact learning In the field of exact learning, *concepts* (sets of instances) can be learned precisely from a *minimally adequate teacher*—an oracle capable of answering two query types (Goldman&Kearns 1995):

- *membership queries* state whether a given instance is in the concept or not
- *equivalence queries* state whether a given hypothesis (set of instances) is equal to the concept held by the teacher. If not, return an instance on which the hypothesis and the concept disagree (a *counterexample*).

The L* algorithm (Angluin 1987) is an exact learning algorithm for learning a DFA from a minimally adequate teacher with knowledge of some regular language L. In this context, the concept is L, the instances are finite sequences ('words') over its alphabet, and the hypotheses are presented as automata $\mathcal{A}$ defining a regular language $L_{\mathcal{A}}$. L* completes when the oracle accepts its latest equivalence query, i.e. when $L_{\mathcal{A}} = L$.

Our approach We treat DFA extraction from RNNs as an exact learning problem. We use Angluin's L* algorithm to elicit a DFA from *any type* of trained RNN, using the RNN as a *teacher*. In doing so, we maintain only a coarse partitioning of the RNN's state space, refining it only as much as necessary to answer L*'s queries.

RNNs as teachers A trained RNN-acceptor can trivially answer membership queries, by feeding input sequences to the network for classification. Answering equivalence queries, however, is not so easy. The main challenge is that no finite interpretation of the network's

[1] In fact, given the results showing that some RNN architectures can count (Gers&Schmidhuber 2001; Weiss et al. 2018b), a DFA may not be sufficient for representing the language learned by an RNN at all.

states and transitions is given upfront: the states of an RNN are high-dimensional real-valued vectors, resulting in an infinite state space which cannot be exhaustively enumerated and compared to the hypothesis.

To address this challenge, we use a *finite abstraction* of the RNN R to answer equivalence queries: we define a finite partitioning of the state space, and create from it an automaton which can be compared to the hypothesis $\mathcal{A}$. A unique aspect of this setting compared to previous L* works is that we only observe *an abstraction* of the teacher. This means that when there is a disagreement between the teacher and the learner, it may be not that the learner is incorrect and needs to refine its representation, but rather (or also) that our abstraction of the teacher is not precise enough and must be refined. Indeed, at every equivalence query, the current finite abstraction and current proposed automaton $\mathcal{A}$ act as two hypotheses for the RNN R's ground truth, which must at least be equivalent to each other in order to both be equivalent to R. Thus, whenever the two disagree on a sample, we find its true classification in R, obtaining through this either a counterexample to $\mathcal{A}$ or a refinement to the abstraction.

Main contributions The main contributions of this paper are:

- We present a novel and general framework for extracting automata from trained RNNs, using the RNNs as teachers in an exact learning setting.
- We implement[2] the technique and show its ability to extract descriptive automata in settings where previous approaches fail. We demonstrate its effectiveness on modern RNN architectures—multi-layer LSTMs and GRUs.
- We describe how the technique can be used to learn DFAs from only positive examples, and demonstrate its effectiveness in this setting. To do so we show how to create RNN-acceptors from positive examples only, using a language modeling objective.
- We apply our technique to RNNs trained to 100% train and test accuracy on simple languages, and discover in doing so that some RNNs have not generalised to the intended concept. Our method easily reveals and produces *adversarial inputs*—words misclassified by the trained RNN and not present in the train or test set.

A basic version of this paper has been presented in ICML 2018 (Weiss et al. 2018a).

2 Preliminaries

In this paper we use the following notations and terminology.

2.1 Automaton and classification function

A deterministic finite automaton (DFA) A is a tuple $\langle \Sigma, Q, i, F, \delta \rangle$, in which Σ is the alphabet, Q the set of states, $F \subseteq Q$ the set of accepting states, $i \in Q$ the initial state, and $\delta : Q \times \Sigma \rightarrow Q$ the transition function. For a given automaton we add the notation $f : Q \rightarrow \{Acc, Rej\}$ as the function giving the classification of each state, i.e. $f(q) = Acc \iff q \in F$, and the notation $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ as the recursive application

[2] www.github.com/tech-sr/lstar_extraction

of δ to a sequence, i.e.: for every $q \in Q$, $\hat{\delta}(q, \epsilon) = q$, and for every $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\hat{\delta}(q, w{\cdot}\sigma) = \delta(\hat{\delta}(q, w), \sigma)$. As an abuse of notation, use $\hat{\delta}(w)$ to denote $\hat{\delta}(i, w)$.

The classification of a word $w \in \Sigma^*$ by a DFA A is defined $A(w) = f(\hat{\delta}(w))$, and the regular language defined by A is the set of words it accepts, $L_A = \{w \in \Sigma^* \mid A(w) = Acc\}$.

Two automata A and B are *equivalent* if $L_A = L_B$, and an automaton $A = \langle \Sigma, Q, i, F, \delta \rangle$ is *minimal* if for every automaton $A' = \langle \Sigma, Q', i', F', \delta' \rangle$ equivalent to A, $|Q| \leq |Q'|$. Two states $q_1, q_2 \in Q$ of an automaton $A = \langle \Sigma, Q, i, F, \delta \rangle$ are *equivalent* if for every $w \in \Sigma^*$, $f(\hat{\delta}(q_1, w)) = f(\hat{\delta}(q_2, w))$, and an automaton is minimal iff it has no two equivalent states.

For visual clarity, 'sink reject states'—states $q \notin F$ for which $\delta(q, \sigma) = q$ for every σ—are not drawn in images of DFAs in this paper. Thus for example the second DFA in Fig. 1 actually has 3 states, and rejects the sequence ") ".

2.2 Recurrent neural networks

An RNN R is a parameterised function $g_R(h, x)$ that takes as input a state-vector $h_t \in \mathbb{R}^{d_s}$ and an input vector $x_{t+1} \in \mathbb{R}^{d_i}$ and returns a state-vector $h_{t+1} \in \mathbb{R}^{d_s}$. An RNN can be applied to a sequence $x_1, ..., x_n$ by recursive application of the function g_R to the vectors x_i, beginning from a given initial state $h_{0,R}$ associated with the network. When applying an RNN to a sequence over a finite alphabet, each symbol is deterministically mapped to an input vector using either a one-hot encoding[3] or an embedding matrix, the method presented in this work is agnostic to this choice. For convenience, we refer to input symbols and their corresponding input vectors interchangeably.

We denote the state space of a network R by $S_R \subseteq \mathbb{R}^{d_s}$, and by $\hat{g_R} : S_R \times \Sigma^* \rightarrow S_R$ the recursive application of g_R to a sequence, i.e. for every $h \in S_R$, $\hat{g_R}(h, \epsilon) = h$, and for every $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\hat{g_R}(h, w{\cdot}\sigma) = g_R(\hat{g_R}(h, w), \sigma)$. As an abuse of notation, we also use $\hat{g_R}(w)$ to denote $\hat{g_R}(h_{0,R}, w)$.

2.3 RNN-acceptors

A binary *RNN-acceptor* is an RNN with an additional function $f_R : S_R \rightarrow \{Acc, Rej\}$ that receives a state vector h_t and returns an accept or reject decision. The RNN-acceptor R is the pair of functions g_R, f_R with associated initial state $h_{0,R}$, The classification of a word $w \in \Sigma^*$ by an RNN-acceptor R is defined $R(w) = f_R(\hat{g_R}(w))$, and the language defined by R is the set of words it accepts, $L_R = \{w \in \Sigma^* \mid R(w) = Acc\}$.

A given RNN-acceptor can be interpreted as a deterministic, though possibly infinite, automaton, which we do note is a more powerful model than that of deterministic *finite* automata.

We drop the subscript R when it is clear from context.

2.4 Multi-layer RNNs

RNNs are often arranged in layers ("deep RNNs"). In a k-layers layered configuration, there are k RNN functions $g_1, ..., g_k$, which are applied to an input sequence $x = x_1, ..., x_m$

[3] A one-hot encoding assigns each symbol in an alphabet of size v to an integer i in 1, ..., v, and maps the symbol to an indicator vector in $\mathbb{R}^v$ where the ith entry is 1 and the others are 0.

as follows: x is mapped by g_1 to a sequence of state vectors $h_{1,1}, ..., h_{1,m}$, and then each sequence $h_{i,1}, ..., h_{i,m}$ is mapped by g_{i+1} to the sequence $h_{i+1,1}, ..., h_{i+1,m}$. For such multi-layer configurations, we take the entire state-vector at time t to be the concatenation of the individual layers' state vectors: $h_t = h_{1,t} \cdot h_{2,t} \dots \cdot h_{k,t}$. Generally, the classification component of a multi-layered RNN-acceptor or LM-RNN is applied only to the final state of the top layer: $f_R(h_t) = f'_R(h_{t,x})$ for some f'_R.

2.5 RNN architectures

The parameterised functions g_R and f_R can take many forms. The function f_R can take the form of a linear transformation or a more elaborate classifier. The original form of g_R is the Elman RNN (Elman 1990), in which g_R is an affine transform followed by a non-linearity, $g_R(h, x) = \tanh(W^x x + W^h h + b)$. Here W^x, W^h and b are the parameters of the function that need to be trained, and have dimensions $d_s \times d_i$, $d_s \times d_s$, and $d_s \times 1$ respectively. Other popular forms are the Long Short-Term Memory (LSTM) (Hochreiter&Schmidhuber 1997) and the Gated Recurrent Unit (GRU) (Cho et al. 2014; Chung et al. 2014). These more elaborate functions are based on a differentiable gating mechanism, and have been repeatedly demonstrated to be easier to train than the Elman RNN, and to robustly handle long-range sequential dependencies. We refer the interested readers to textbooks such as Goodfellow et al. (2016); Goldberg (2017) or to the documentation of the PyTorch framework (Paszke et al. 2019) for their exact forms.

Our technique is agnostic to these internal differences, treating the functions f_R and g_R as black boxes. In our experiments, we use linear transformation for f_R, and the popular LSTM and GRU architectures for g_R. For the LSTM, whose transition function is often described as converting a triplet of input-vector, state-vector and memory-vector to a next state-vector and memory-vector, we treat the concatenation of the state-vector and memory-vector as a single state-vector with dimension $d_s = 2h_s$, where h_s is the hidden size of the cell.

2.6 Network abstraction

Given a neural network R with state space S and alphabet Σ, and a partitioning function $p : S \to \mathbb{N}$, Omlin and Giles (1996) presented a method for extracting a DFA for which every state is a partition from p, and the state transitions and classifications are defined by a single sample from each partition. Their method can be seen as a simple sheared exploration of the partitions defined by p. The exploration begins from the partition containing the initial state $p(h_{0,R})$, explores according to the network's transition function g_R, and shears wherever it reaches an abstract state (partition) that has already been visited. We present it as pseudocode in Algorithm 1.

We denote by $A_{R,p}$ the DFA extracted by this method from a network R and partitioning p, and denote all its related states and functions by subscript R, p.[4] Note that the algorithm is guaranteed to extract a deterministic finite automaton (DFA) from any network and finite partitioning.

[4] The exact order of the exploration (i.e., selection of states from *New*) is not important, but if we want to be well defined we can assume that *New* is FIFO and that Σ has an order which the for loop over it follows. This would make the exploration a (sheared) BFS.

Algorithm 1 Pseudo-code of RNN R exploration with state space partitioning $p : S \to \mathbb{N}$. The functions of the network are marked R subscript.

```
Method map_transitions(R, p):
    $Q, F, \delta \leftarrow \emptyset$
    $New \leftarrow \{h_{0,R}\}$
    while $New \neq \emptyset$ do
        $h \leftarrow$ pop from New
        $q \leftarrow p(h)$
        if $q \notin Q$ then
            $Q \leftarrow Q \cup \{q\}$
            if $f_R(h) = Acc$ then  $F \leftarrow F \cup \{q\}$
            for $\sigma \in \Sigma$ do
                $h' \leftarrow g_R(h, \sigma)$
                $\delta \leftarrow \delta \cup \{((q, \sigma), p(h'))\}$
                $New \leftarrow New \cup \{h'\}$
            end
        end
    end
```

2.7 The L * algorithm

Angluin's L* algorithm (1987) is an exact learning algorithm for regular languages. The algorithm learns an unknown regular language L over an alphabet Σ from a teacher T, generating as output a DFA $\mathcal{A}$ that accepts L. In our work we implement such a teacher for L* around a given RNN, and apply L*to this teacher directly. Therefore it is sufficient here to limit our discussion to only the requirements of this interaction.

L* interacts with a teacher that must answer two types of queries: *membership queries*, in which the teacher must classify words presented by L*, and *equivalence queries*, in which the teacher must accept or reject automata proposed by L*based on whether or not they correctly represent the target language. If the teacher rejects an automaton $\mathcal{A}$, it must also provide a counterexample—a word that $\mathcal{A}$ misclassifies with respect to the target language. L* continues to present queries to the teacher until the teacher accepts a hypothesis $\mathcal{A}$, at which point it terminates and returns $\mathcal{A}$.

The L* algorithm is guaranteed to always present a minimal DFA consistent with all membership queries given so far, and we use this fact in our work. Additionally, provided the target language T is regular, L* is guaranteed to return a minimal DFA for T in polynomial time in ($|Q| + |w| + |\Sigma|$), where $|Q|$ is the number of states in that DFA, Σ is the input alphabet, and $|w|$ is the length of the longest counterexample given by the teacher (Angluin 1987; Berg et al. 2005).

3 Existing approaches and related work

Soon after the introduction of the RNN (Elman 1990), it was shown that, when learning a regular language, a simple (“Elman-”) RNN is able to cluster its reachable states in a manner that resembles a (not necessarily minimal) DFA for that language (Cleeremans et al. 1989[5]). Since then there has been a lot of research on extracting rules, and in particular finite automata, from RNNs. Partial surveys of these works are presented by Wang et al. (2017) and Jacobsson (2005).

Transition mapping In their 1996 paper, Omlin and Giles experimented on second-order RNNs, and found that their learned states also tend to cluster in small areas in the network state space. Through this, and an assumption of continuity in the network behavior (i.e., small changes in the current state lead only to small changes in the next state), they concluded that it was safe to cluster like-valued state vectors together as one state, and traverse these clustered states in order to recover a DFA from the RNN.

In particular, given a neural network R with state space S and alphabet Σ, and a partitioning function $p : S \rightarrow \mathbb{N}$, Omlin and Giles presented a method (Algorithm 1) for extracting a DFA abstraction of the network in which every abstracted state is an entire partition from p, and the transitions between abstracted states and their classifications are obtained by a single sample of the continuous values in each such partition.

In both their own work and more recent research by others (e.g. Wang et al. 2017), this extraction method has been shown to produce DFAs that are reasonably representative of given second-order RNNs—provided the given partitioning captures the differences between the network states well enough.

Quantisation For networks with bounded output values, Omlin and Giles suggested dividing each dimension of the network state space into $q \in \mathbb{N}$ (referred to as the *quantisation level*) equal intervals, yielding q^{d_s} subsets of the output space with d_s being the length of the state vectors.

However, because this technique applies a uniform quantisation over the entire output space, it suffers from inherent state explosion and does not scale to the networks used in practice today: the original paper demonstrates the technique on networks with 8 hidden values, whereas today’s can have hundreds to thousands.

Clustering Other state-partitioning approaches use *clustering* (Cechin et al. 2003; Wang et al. 2017; Cohen et al. 2017). In these approaches, an unsupervised classifier such as k-means is applied to a large sample set of reachable network states, creating a finite number of clusters. The sample states can be found by various methods, such as a BFS exploration of the network state space to a certain depth, or by recording all state vectors reached by the network when applied to its train set (if available). The partitioning of the state space defined by the clusters is then explored in a similar way to that described by Omlin&Giles

[5] This work references a slightly older version of (Elman 1990).

(1996). Clustering approaches yield automata that are much smaller than those given by the partitioning method originally proposed by Omlin and Giles, making them more applicable to networks of today's standards.

Weaknesses In both of these approaches the partitioning is set before the extraction begins, with no mechanism for recognizing and overcoming overly coarse behavior. Both approaches thus face the challenge of choosing the best parameter value for extraction, and are generally applied several times with different parameter values, after which the 'best' DFA is chosen according to a heuristic (e.g., accuracy against RNN on the test set). Additionally, both approaches can still have rather large state space, and—as the exploration of the extracted DFA is performed blindly—these states cannot be merged until the extraction is complete and the DFA can be minimised.

Note on architectures Many of these works use *second order RNNs* (Giles et al. 1990), which are shown to better map DFAs than simple RNNs (Goudreau et al. 1994; Wang et al. 2018). In this work however, we experiment on the popular GRU (Cho et al. 2014; Chung et al. 2014) and LSTM (Hochreiter&Schmidhuber 1997) architectures, as they are more widely used in practice.

3.1 Recent works and future directions

Since the initial publication of this method, several other approaches for extracting DFAs have been suggested, and still other works have begun grappling with more complicated targets such as weighted automata or context free languages.

DFAs Mayr&Yovine (2018) released an L*-based approach for learning DFAs from *any* neural network architecture, answering equivalence queries by drawing random samples over the input alphabet and checking if they are counterexamples to the proposed automaton. Their work analyses this approach from a PAC learning perspective and applies also to completely black box models, in contrast to our own work and other extraction works listed above (which rely on access to the RNN's hidden state from different prefixes). In Sect. 7.7, we compare our method to this approach, highlighting the advantage of the abstraction based approach to equivalence queries when the hidden state is available.

Wang&Niepert (2019) propose *state-regularised RNNs*, a variant of RNNs that is regularised towards transitioning between a finite number of learned internal states. Their work discusses both training these new RNNs and the recovery of DFAs from them once trained, presenting an extraction method tailored to their proposed architecture.

WFAs Ayache et al. (2018) use spectral learning (Balle et al. (2014)) to extract weighted, non-deterministic finite automata (WFAs) from any black box language model, evaluating on RNNs. Okudono et al. (2020) also apply spectral learning for WFA extraction, but this time to whitebox RNNs, using an adaptation of the equivalence query presented in this paper to refine the WFA beyond the initial spectral extraction. In a later work, we adapt L*to a weighted setting, extracting weighted deterministic finite automata (WDFAs) from any black box language model (Weiss et al. 2019). Finally, more recently, Zhang et al. (2021) expand on the partitioning and then transition-mapping approach of the classical

DFA extraction papers (Omlin&Giles 1996) to recover WFAs from RNNs without using exact or spectral learning.

CFGs With the understanding that some RNN architectures behave more like counter machines (Gers&Schmidhuber 2001; Weiss et al. 2018b; Suzgun et al. 2019), which are more expressive than DFAs, and indeed that an RNN in general might be trained on something more complicated than a regular language, it becomes interesting to consider extraction of context free languages (CFGs) from RNNs.

Recently, Yellin&Weiss (2021) use the DFA-extraction method presented in this paper as the initial step in an algorithm for extracting a subclass of CFGs from trained RNNs,[6] and Barbot et al. (2021) apply results on *visibly pushdown languages* and *tree automata* to extract a different subclass of CFGs, also from trained RNNs. Independently, there exist several works on learning (subclasses of) CFGs from queries, or from examples only, that have not yet been applied for extraction from RNNs (Sakakibara 1992; Yokomori 2003; Tellier 2006; Clark&Eyraud 2007; Clark 2010; D'ulizia et al. 2010; Shibata&Yoshinaka 2016; Clark&Yoshinaka 2016; Yoshinaka 2019).

4 Learning automata from RNNs using L*

In the following sections we show how to build a teacher for the L^* algorithm around a given RNN-acceptor R. The teacher must be able to answer membership and equivalence queries as required by L^*.

To implement **membership queries** we rely on the RNN classifier itself. To determine whether a given word w is in the unknown language L_R, we simply run the RNN on this word, and check whether it accepts or rejects w.

To implement **equivalence queries** we check the equivalence of the L^* hypothesised automaton $\mathcal{A}$ against an abstraction $A_{R,p}$ of the network, where p is a partitioning over the network's state space. If we find a disagreement $w \in \Sigma^*$ between $\mathcal{A}$ and the current abstraction $A_{R,p}$, we use R to determine whether this is because the L^* hypothesis is incorrect (i.e., $L_R(w) \neq \mathcal{A}(w)$), or a result of a poor abstraction (i.e., $L_R(w) \neq A_{R,p}(w)$). In the former case ($L_R(w) \neq \mathcal{A}(w)$), we end the equivalence query and return w as a counterexample to L^*. Otherwise, we refine p and restart the comparison of $\mathcal{A}$ and $A_{R,p}$. If no such disagreement w is found (i.e., $\mathcal{A}$ and $A_{R,p}$ are equivalent), we accept L^*'s hypothesis and the extraction ends.

p is maintained between equivalence queries, i.e., the partitioning p at the start of the $j{+}1^{\text{th}}$ equivalence query is the same partitioning p from the end of the j^{th} equivalence query.

In theory, the extraction continues until the automaton proposed by L^* is accepted, i.e., $\mathcal{A}$ and $A_{R,p}$ converge. In practice, for some RNNs this may take a long time and yield a large DFA (>30,000 states). To counter this, we place time or size limits on the interaction,

[6] By creating an algorithm for generalising CFGs from a sequence of DFAs, and using the hypotheses provided by L^*as that sequence.

after which the last L* hypothesis is returned.[7] We see that these DFAs still generalise well to their respective networks.

The partitioning p has to be coarse enough to facilitate feasible computation of $A_{R,p}$, but fine enough to capture the interesting observations made by the network. As we have an iterative setting, we can satisfy this by starting with a very coarse initial abstraction and refining it only sparingly, whenever it is proven incorrect.

The equivalence queries are described in Sect. 5, and the partitioning and its refinements in Sect. 6.

Note Convergence of $A_{R,p}$ and $\mathcal{A}$ does not guarantee that R and $\mathcal{A}$ are equivalent. Providing such a guarantee would be an interesting direction for future work.

5 Answering equivalence queries

Given a network R, a partitioning function p over its state space S, and a proposed minimal automaton $\mathcal{A}$, we wish to check whether the abstraction of the network $A_{R,p}$ is equivalent to $\mathcal{A}$, preferably while exploring as little of $A_{R,p}$ as necessary. If the two are not equivalent—meaning, necessarily, that at least one is not an accurate representation of the network R—we wish to find and resolve the cause of the inequivalence, either by returning a counterexample to L*(and so refining $\mathcal{A}$), or refining the partitioning function p (and so the abstraction $A_{R,p}$) in the necessary area. Hence our equivalence query must be able not only to return counterexamples when necessary, but also to specifically identify overly-coarse partitions in the partitioning p.

For clarity, from here onwards we refer to the continuous network states $h \in S$ as R-states, the abstracted states in $A_{R,p}$ as A-states, and the states of the L* DFAs $\mathcal{A}$ as L-states.

In this section we describe the details of an equivalence query assuming a given partitioning p and refinement operation `refine`. We present our initial partitioning p_0 and `refine` operation in Sect. 6.

5.1 Parallel exploration

The key intuition to our approach is the fact that $\mathcal{A}$ is minimal, and so each state in the DFA $A_{R,p}$ should—if the two automata are equivalent—be equivalent to exactly one state in the DFA $\mathcal{A}$. This is based on the fact that for automata $A = \langle \Sigma, Q, i, F, \delta \rangle$ and $A' = \langle \Sigma, Q', i', F', \delta' \rangle$ in which A' is minimal, A and A' are equivalent if and only if there exists a mapping $m : Q \to Q'$ satisfying that $m(i) = i'$, $f(q) = f'(m(q))$, and $m(\delta(q, \sigma)) = \delta'(m(q), \sigma)$ for every $q, \sigma \in Q \times \Sigma$.

To check the equivalence of $A_{R,p}$ and $\mathcal{A}$ without necessarily having to fully explore $A_{R,p}$, we build such a mapping between their states on-the-fly: we associate between states of

[7] We could also return the last abstraction, $A_{R,p}$, and focus on refining p over returning counterexamples. But we find that the abstractions are often less accurate (see Sect. 7.8). We suspect this is due to the lack of 'foresight' $A_{R,p}$ has, as opposed to L*'s many separating suffix strings (loosely, L* works by maintaining two growing lists of 'interesting' prefixes and suffixes, generating an equivalence query only when all the prefixes going into the each hypothesis state have the same classification on all of the suffixes).

the two automata during the extraction of $A_{R,p}$, by traversing $\mathcal{A}$ in parallel to the extraction of $A_{R,p}$ (which is extracted according to Algorithm 1). We update this association for all R-states visited during this extraction, i.e., including those at which the traversal is sheared.[8] Any inconsistencies (*conflicts*) in this association are definite indicators of inequivalence between $A_{R,p}$ and $\mathcal{A}$.

5.1.1 Conflict types

We refer to associations in which an accepting A-state is associated with a rejecting L-state or vice versa as *abstract classification conflicts*. We refer to multiple but disagreeing associations for a single A-state, i.e. situations in which one A-state is associated with two different (minimal) L-states, as *partitioning conflicts*. (The inverse, in which one minimal L-state is associated with several A-states, is not a problem: $A_{R,p}$ is not necessarily minimal and so these states may be equivalent.)

Recalling that the ulterior motive is to find inconsistencies between the proposed automaton $\mathcal{A}$ and the given network R, and that the exploration of $A_{R,p}$ runs atop an exploration of the actual R-states, we also check at each point during the exploration whether the current R-state $h \in S_R$ has identical classification to that of the current L-state reached in the parallel traversal of $\mathcal{A}$. As the classification of a newly discovered A-state is determined by the R-state with which it was first mapped, this also covers all abstract classification conflicts. We refer to failures of this test generally as *classification conflicts*, and check only for them and for partitioning conflicts.

5.2 Conflict resolution and counterexample generation

Classification conflicts are a sign that a path $w \in \Sigma^*$ satisfying $R(w) \neq \mathcal{A}(w)$ has been traversed in the exploration of $A_{R,p}$, and so necessarily that w is a counterexample to the equivalence of $\mathcal{A}$ and R. They are resolved by returning the path w as a counterexample to L*, so that it may refine its observations and provide a new automaton. All that is necessary for this is to maintain the current path w throughout the exploration.

Partitioning conflicts are a sign that an A-state $q \in Q_{R,p}$, that has already been reached with a path w_1 during the exploration of $A_{R,p}$, has been reached again with a new path w_2 for which the L-state is different from that of w_1. In other words, partitioning conflicts give us two sequences $w_1, w_2 \in \Sigma^*$ for which $\hat{\delta_{R,p}}(w_1) = \hat{\delta_{R,p}}(w_2)$ but $\hat{\delta_{\mathcal{A}}}(w_1) \neq \hat{\delta_{\mathcal{A}}}(w_2)$. We denote by $q_1, q_2 \in Q_{\mathcal{A}}$ the L-states reached in $\mathcal{A}$ by these sequences, $q_i = \hat{\delta_{\mathcal{A}}}(w_i)$. As $\mathcal{A}$ is a minimal automaton, q_1 and q_2 are necessarily inequivalent, meaning there exists a differentiating suffix $s \in \Sigma^*$ for which $f_{\mathcal{A}}(\hat{\delta_{\mathcal{A}}}(q_1, s)) \neq f_{\mathcal{A}}(\hat{\delta_{\mathcal{A}}}(q_2, s))$, i.e. for which $f_{\mathcal{A}}(w_1 \cdot s) \neq f_{\mathcal{A}}(w_2 \cdot s)$. Conversely, as $\hat{\delta_{R,p}}(w_1) = \hat{\delta_{R,p}}(w_2)$ then $\hat{\delta_{R,p}}(w_1 \cdot s) = \hat{\delta_{R,p}}(w_2 \cdot s)$, and so $f_{R,p}(w_1 \cdot s) = f_{R,p}(w_2 \cdot s)$.

Clearly in this case $\mathcal{A}$ and $A_{R,p}$ must disagree on the classification of either $w_1 \cdot s$ or $w_2 \cdot s$, and so at least one of them must be inconsistent with the network R. In order to determine the 'offending' automaton, we pass both $w_1 \cdot s$ and $w_2 \cdot s$ to R for their true classifications. If $\mathcal{A}$ is found to be inconsistent with the network, the word on which $\mathcal{A}$ and R disagree is returned to L* as a counterexample.

[8] These are important: they are the repeat visits to an A-state, from which a *partitioning conflict* may occur.

Else, $w_1{\cdot}s$ and $w_2{\cdot}s$ are necessarily classified differently by the network, and $A_{R,p}$ should not lead w_1 and w_2 to the same A-state. The R-states $h_1 = \hat{g}(w_1)$ and $h_2 = \hat{g}(w_2)$ are passed, along with the current partitioning p, to a refinement operation, which refines p such that the two are no longer mapped to the same A-state—preventing a reoccurrence of that particular conflict.

The previous reasoning can be applied to w_2 with *all* paths w_1 that have reached the conflicted A-state $q \in Q_{R,p}$ without conflict before w_2 was traversed. As such, the classifications of *all* the words $w_1{\cdot}s$ are tested against the network, prioritising returning a counterexample over refining the partitioning.[9] If eventually it is the partitioning that is refined, then the R-state that triggered the conflict, $h = \hat{g}(w_2)$, is split from all R-states $h_1 = \hat{g}(w_1)$ for w_1 that have already reached q in the exploration, in one single refinement.[10]

Every time the partitioning is refined, the guided exploration starts over, and the process repeats until either a counterexample is returned to L*, equivalence is reached (exploration completes without a counterexample), or some predetermined limit (such as time or partitioning size) is exceeded. We note that in practice—and very often so with the decision-tree based refinement operation that we present—there are cases in which starting over is equivalent to merely updating the associated A-state $p(h)$ of the R-state h that triggered the refinement and continuing the exploration from there, and we implement our equivalence query to take advantage of this.

In our implementation, whenever we find several potential counterexamples to the proposed DFA, we check them in order of increasing length and return the shortest counterexample we have found.

[9] As we will ultimately return the last L*hypothesis and not the abstraction if time runs out (see Sect. 7.8).

[10] At least, this is the ideal case. In practice, we allow a relaxed setting where it might only be split from some (non-empty) subset of them. In the worst case, this will trigger a further refinement when the query is attempted again.

5.3 Algorithm

Algorithm 2 Pseudo-code for equivalence checking of an RNN R and minimal DFA $\mathcal{A}$, with initial partitioning p_0. The main loop is in *check_equivalence*.

```
Method update_records($q, h, q_{\mathcal{A}}, w$):
    Visitors($q$) ← Visitors($q$) ∪ {$h$}
    Path($h$) ← $w$
    Association($q$) ← ($q_{\mathcal{A}}$)
    Push(Unexplored,$h$)

Method handle_partition_conf($q, h, q_{\mathcal{A}}, q'_{\mathcal{A}}$):
    find $s \in \Sigma^*$ s.t. $f_{\mathcal{A}}(q_{\mathcal{A}}, s) \neq f_{\mathcal{A}}(q'_{\mathcal{A}}, s)$
    for $h' \in$ Visitors($q$) do
        $w \leftarrow Path(h') \cdot s$
        if $f_R(w) \neq f_{\mathcal{A}}(w)$ then
            return Reject, $w$
        end
    end
    $p \leftarrow$ refine($p, h,$Visitors($q$)\{$h$})
    return Restart_Exploration, $\varepsilon$

Method parallel_explore($R, \mathcal{A}, p$):
    empty all of: $Q, F, \delta$, Unexplored, Visitors, Path, Association
    $q_0 \leftarrow p(h_{0,R})$
    update_records($q_0, h_{0,R}, q_{\mathcal{A},0}, \varepsilon$)

    while Unexplored $\neq \emptyset$ do
        $h \leftarrow$ Pop(Unexplored)
        $q \leftarrow p(h)$
        $q_{\mathcal{A}} \leftarrow$ Association($q$)
        if $f_R(h) \neq f_{\mathcal{A}}(q_{\mathcal{A}})$ then
            return Reject, Path(h)
        end
        if $q \in Q$ then
            continue
        end
        $Q \leftarrow Q \cup \{q\}$
        if $f_R(h) = Acc$ then
            $F \leftarrow F \cup \{q\}$
        end
        for $\sigma \in \Sigma$ do
            $h' \leftarrow g_R(h, \sigma)$
            $q' \leftarrow p(h')$
            $\delta(q, \sigma) \leftarrow q'$
            $q'_{\mathcal{A}} \leftarrow \delta_{\mathcal{A}}(q_{\mathcal{A}}, \sigma)$
            if $q' \in Q$ and Association($q'$) $\neq q'_{\mathcal{A}}$ then
                return handle_partition_conf($q', h'$, Association($q'$), $q'_{\mathcal{A}}$)
            end
            update_records($q', h', q'_{\mathcal{A}}$,Path($h$)·$\sigma$)
        end
    end
    return Accept, $\varepsilon$

Method check_equivalence($R, \mathcal{A}, p_0$):
    $p \leftarrow p_0$
    verdict ← Restart_Exploration
    while verdict = Restart_Exploration do
        verdict, $w \leftarrow$ parallel_explore($R, \mathcal{A}, p$)
    end
    return verdict,$w$
```

Pseudocode for this entire equivalence checking procedure (ignoring preference for shortest counterexamples) is presented in Algorithm 2.[11] The description here assumes the existence of a refinement operation `refine` separating in the partitioning an R-state h from a set of other R-states H, we present such a method in Sect. 6.

The overall iterative process, including the refinements to p, is desribed in `check_equivalence`, and the equivalence checking for a specific partitioning p is given in `parallel_explore`.

`parallel_explore` attempts to build $A_{R,p}$ in variables Q, F, q_0, δ, while also maintaining the associations of these states to R and $\mathcal{A}$ as follows:

- `Visitors` holds for every A-state q the set of all R-states h satisfying $p(h) = q$ that have been visited during the exploration. This is used for refinements triggered by partitioning conflicts.
- `Path` holds for every R-state h the sequence $w \in \Sigma^*$ with which h has been visited during the exploration.[12] This is used for generating potential counterexamples when handling a partitioning conflict.
- `Association` holds for every A-state q the L-state $q' \in Q_{\mathcal{A}}$ visited in the parallel exploration of $\mathcal{A}$ the first time that q was visited. If at any point q is visited while the parallel exploration is on a different state $q'' \neq q'$, a partitioning conflict is triggered.

Note that finding the separating suffix for two inequivalent states q_1, q_2 of a given automaton A can be done by a simple parallel BFS exploration of the states reachable from q_1 and q_2 in A, continuing until two states with opposite classifications are found.

6 Abstraction and refinement

Given a partitioning p, an R-state h, and a set of R-states $H \subseteq S \setminus \{h\}$, we must refine p to obtain a new partitioning p' satisfying:

1. for every $h_1 \in H$, $p'(h) \neq p'(h_1)$, and
2. for every $h_1, h_2 \in S$, if $p(h_1) \neq p(h_2)$ then $p'(h_1) \neq p'(h_2)$.

The first condition separates (in the partitioning) the R-states that caused the partitioning conflict leading to the refinement. The second condition maintains separations made by earlier refinements, i.e., it prevents previously created abstract states from being merged.

We want to generalise the information given by h and H well, so as not to invoke excessive refinements as new R-states are explored. Additionally, we would like to keep the partitioning as small as possible, so that $A_{R,p}$ can be explored and compared to $\mathcal{A}$ in reasonable time at every equivalence query.

To keep the partitioning small, we settle on a decision tree structure, in which each refinement only splits the partition in which the conflict was recognised. Additionally, seeing that in practice our equivalence checking method can overcome imperfect splits between H and h by generating further splits if necessary, we relax the first condition.

[11] And full code is available online at `www.github.com/tech-srl/lstar_extraction`.

[12] Technically this should be maintained as a *set* of sequences reaching h, but in practice, the probability of there being more than one such sequence per h is too low to consider.

Specifically, we allow the classifiers splitting between H and h in the conflicated partition to not do so perfectly, provided they separate at least some of H from h.

Our method is unaffected by the length of the R-states, and very conservative: each refinement increases the number of A-states by exactly one. Our experiments show that it is fast enough to quickly find counterexamples to proposed DFAs.

6.1 Initial partitioning

In addition to a refinement method, our algorithm needs an initial partitioning p_0 from which to start the first equivalence query. As we wish to keep the abstraction as small as possible, we begin with no state separation at all: $p_0 : h \mapsto 0$.

6.2 Decision-tree based partitioning, with support vector refinement

Let $h \in S, H \subset S$ be the R-states with which a refinement was invoked. We know the refinement is only applied to h, H satisfying $p(h) = p(h')$ for every $h' \in H$. To keep the partitioning small, we define a gentle refinement operation, in which for every call we only split the single partition $p(h)$. This approach avoids state explosion by adding only one A-state per refinement.

Decision Tree It is natural to maintain a partitioning p refined over time in this way as a decision tree, where each internal node tracks some single refinement made to p, and its leaves are the current A-states of the abstraction.

SVM classifiers At every refinement, for the split of $p(h)$, we would like to allocate a region around the R-state h that is large enough to contain other R-states that behave similarly, but separate from neighbouring R-states that do not. We achieve this by fitting an SVM (Boser et al. 1992) classifier with an RBF kernel[13] to separate h from H (splitting the partition $p(h)$ in exactly two). The max-margin property of the SVM ensures a large space around h, while the Gaussian RBF kernel allows for a non-linear partitioning of the space. We use this classifier to split the A-state $p(h)$, yielding a new partitioning p' with exactly one more A-state than p.

Whenever the SVM successfully separates h from H entirely, this approach satisfies the requirements of refinement operations. Otherwise, the method fails to satisfy condition 1 of the refinement operation. Nevertheless, the SVM classifier will always separate at least one of the R-states $h' \in H$ from h, and later explorations can invoke further refinements if necessary. In practice we see that this does not hinder the main goal of the abstraction, which is finding counterexamples to equivalence queries.

Unlike mathematically defined partitionings such as the quantisation proposed by Omlin&Giles (1996), our abstraction's storage is linear in the number of A-states it can map to; and computing an R-state's associated A-state may be linear in this number as well (e.g. if the decision tree is a chain). Luckily, as this number of A-states also grows very slowly (linearly in the number of refinements), this does not become a problem.

[13] While we see this as a natural choice, other kernels or classifiers may yield similar results. We do not explore such variations in this work.

6.3 Practical considerations

As the initial partitioning and the refinement operation are very coarse, our method runs the risk of accepting very small but wrong DFAs early in the extraction.

To counter this, two measures are taken:

1. At the beginning of extraction, one accepting and one rejecting sequence are provided to the teacher, and then checked as potential counterexamples at the beginning of every equivalence query.[14] Conversely, if these are not available, equivalence queries are extended with n random samples for some small n (e.g. $n = 100$) and range of lengths (e.g. 0-100): whenever $\mathcal{A}$ and $A_{R,p}$ are equivalent, n random samples are generated and checked as potential counterexamples ($\mathcal{A}(w) \neq R(w)$) before $\mathcal{A}$ can be accepted.
2. The first refinement is aggressive, generating a greater (but still manageable) number of A-states than made with the main single-partition split approach used for the rest of the extraction.

The first measure is taken specifically to prevent erroneous termination of the extraction on a single state automaton, and requires only two samples (if provided) or short additional time before accepting an equivalence query.

The second measure prevents the extraction from too readily terminating on small DFAs, by creating a (manageably) large $A_{R,p}$ that will hopefully capture a relatively rich representation of the RNN. Our method for it is presented in Sect. 6.3.1.

6.3.1 Aggressive difference-based refinement

At the first refinement, instead of splitting $p_0(h)$ to separate h from all or most of H using a single SVM, we split S in its entirety across multiple dimensions chosen according to h and H. Specifically, we calculate the mean h_m of H, find the d dimensions with the largest gap between h and h_m, and then split S along the middle of that gap for each of the d dimensions.

The resulting partitioning can be comfortably stored in a decision tree of depth d. It is intuitively similar to that of the quantisation suggested by Omlin and Giles, except that it focuses only on the dimensions with the greatest deviation of values between the states being split, and splits the 'active' range of values.

The value d may be set by the user, and increased if the extraction is suspected to have converged too soon. We found that $d = 10$ generally provides a strong enough initial partitioning of S, without making the abstraction too large for feasible exploration.

[14] When using these in our experiments, we used the shortest possible examples, e.g., the empty sequence and) for the balanced parentheses language.

7 Experimental results

We first demonstrate the effectiveness of our method on LSTM- and GRU-acceptors[15] trained on the Tomita grammars (1982), which have been used as benchmarks in previous automata-extraction work (Wang et al. 2017), and then on substantially more complicated languages. We show the effectiveness of our refinement-based equivalence query approach over that of plain random sampling and present cases in which our method extracts informative DFAs where other approaches fail. In addition, for some seemingly perfect networks, we find that our method quickly returns counterexamples representing deviations from the target language.

We clarify that when we refer to extraction time for any method, we consider the *entire* process: from the moment the extraction begins, to the moment a DFA is returned.[16]

Prototype Implementation and Settings

We implemented all methods in Python, using PyTorch (Paszke et al. 2019) and scikit-learn (Pedregosa et al. 2011). For the SVM classifiers, we used the SVC variant, with regularisation factor $C = 10^4$ to encourage perfect splits and otherwise default parameters—in particular, the RBF kernel with gamma value $1/(\text{num features})$.

All training and extraction was done on amazon instances of type `p3.2xlarge`, except for the BP and email classifier RNNs which were run on `p2.xlarge`.

7.1 Languages

We consider the Tomita Grammars (7.4.1), and more complicated regular languages defined by small, randomly sampled DFAs (7.4.2). We also consider the language of legal email addresses (defined precisely in 7.9.1), and the language of *balanced parentheses* (BP): the set of sequences over `()a-z` in which the parentheses are balanced, e.g. `a(a)ba` and `()(())`.

7.2 Sample sets and training

Tomita and Random Regular Languages We use train, validation, and test sets of sizes 5000, 1000 and 1000 containing samples of lengths 1-100 (uniformly distributed). To get ‘representative’ sample sets, we define a distribution over each DFA’s state transitions favouring transitions which do not reduce the number of reachable states,[17] sample from that distribution, and train the RNN to provide correct output for all prefixes of every sample (as opposed to only the full samples).[18] We train these RNNs with the Adam optimiser,

[15] While many previous automata-extraction works evaluate on second-order RNNs (Giles et al. 1990), we evaluate on the more popular LSTM and GRU architectures. We note that with the exception of quantisation-based partitioning (Omlin&Giles 1996), which requires minor adaptation for unbounded RNN state space, all of these methods—including our own—can be applied to any RNN architecture.

[16] Covering among others: abstraction exploration, abstraction refinements (including training SVM classifiers), and L* refinements (for our method), and total time for all created DFAs (for k-means clustering). Unless otherwise stated, this time is measured using the `process_time` method in python’s time module.

[17] (E.g., a transition into a sink reject state—unless it also comes from the sink reject state—reduces the number of reachable states.)

[18] The intuition behind this choice is that every ‘irreversible’ transition in the DFA (e.g., the first `0` in a sample for Tomita 1, the language of sequences containing only `1`) is delated, increasing the time spent in the states before them, which might otherwise be underrepresented in the samples.

using initial learning rate 0.0003, an exponential learning rate scheduler with gamma 0.9, and dropout 0.1. Each RNN was trained for up to 100 epochs on its train set, or until the validation set had 100% accuracy for 3 epochs in a row, whichever came sooner.

Balanced Parentheses and Email Addresses We generated positive samples using tailored functions,[19] and negative samples as a mix of both random sequences and mutations of the positive samples.[20] Here we train the RNN only on the full samples (as opposed to classifying every prefix). We trained all networks to 100% accuracy on their train sets, and considered only those that reached 99.9+% accuracy on a test set consisting of up to 1000 uniformly sampled words of each of the lengths $n \in 1, 4, 7, ..., 28$. The positive to negative sample ratios in the test sets were not controlled. The BP and email train sets were randomly generated during training. The BP train set created ≈44600 samples, of which ≈60% were positive for each RNN, and reached balanced parentheses up to depth 11. The email addresses train set created 40000 samples.

7.3 Details on our extraction (practical considerations)

We apply the measures discussed in Sect. 6.3 as follows: First, for all networks, we apply our method with aggressive initial refinement depth $d = 10$ (Sect. 6.3.1). Second, we use additional counterexamples:

Additional Counterexamples For the Tomita and random DFA languages, during extraction, we used random samples as additional potential counterexamples. Specifically, whenever an equivalence query was going to accept, we considered an additional 100 potential counterexamples, each generated as follows: first, we choose a length from $0 - 10$ (uniformly), and then uniformly sample a sequence of that length over the RNN input alphabet.

For BP and email addresses, during extraction, we presented each RNN along with one positive and one negative sample to check for counterexamples at each equivalence query. These were chosen as the shortest positive and shortest negative word in the train set of the RNN, in particular: for BP, the initial samples were the empty sequence (positive) and `)` (negative), and for emails, the initial samples were `0@m.com` (positive) and the empty sequence (negative). For BP, these samples are covered anyway by L^*'s initial membership queries, but for email addresses the positive sample helps 'kick off' the extraction, preventing the method from accepting an automaton with a single (rejecting) state.

No further parameter tuning was required to achieve our results.

7.4 Small regular languages

7.4.1 The Tomita grammars

The Tomita grammars (1982) are the following 7 languages over $\Sigma = \{0, 1\}$:

1. `1*`
2. `(10)*`

[19] For instance, a function that creates emails by uniformly sampling 2 sequences of length 2–8, choosing uniformly from the options `.com`, `.net`, and all `.co.XY` for `X,Y` lowercase characters, and then concatenating the three with an additional `@`.

[20] With mutations obtained by adding, removing, changing, or moving up to 9 characters.

3. The complement of `((0|1)*0)*1(11)*(0(0|1)*1)*0(00)*(1(0|1)*)*`, i.e.: all sequences w which do not contain an odd series of 1s followed later by an odd series of 0s
4. All words w not containing `000`,
5. All w for which $\#_0(w)$ and $\#_1(w)$ are even (where $\#_a(w)$ is the number of a's in w),
6. All w for which $(\#_0(w) - \#_1(w)) \equiv_3 0$, and
7. `0*1*0*1*`.

They are the languages classically used to evaluate DFA extraction from RNNs.

We trained one 1-layer GRU network with hidden size 50 for each Tomita grammar (7 GRUs in total), in the manner described in Sect. 7.2. In training, all but one of the RNNs reached 3 consecutive epochs with 100% validation set accuracy within 10 epochs, and reached 100% test set accuracy. The 6th Tomita grammar was harder to train, with the RNN reaching only 78% validation accuracy after 100 epochs. As our focus is on extraction rather than training, we repeated training on this language, eventually obtaining an RNN with perfect train and validation accuracy for this language as well (this time with initial learning rate 0.0004 and gamma 0.95). We then applied our method to extract from the perfectly trained RNNs.

For each one, our method correctly extracted and accepted the target grammar in under 1 second.

7.4.2 Random small regular languages

Though the Tomita grammars are a popular language set for evaluating DFA extraction from RNNs, they are quite simple: the largest Tomita grammars are still only 5-state DFAs over a 2-letter alphabet. As our method performed so well on these grammars, we expand to more challenging languages.

We considered randomly-generated minimal DFAs of varying complexity, specifically, DFAs with alphabet size and number of states $(|\Sigma|, |Q|) = (3, 5), (5, 5)$ and $(3, 10)$. For each combination we randomly generated 10 minimal DFAs, making 30 DFAs overall. For each DFA we trained 6 2-layer RNNs: 3 GRUs and 3 LSTMs, each with hidden state sizes $d_s = 50, 100$ and 500, this makes 180 RNNs overall. The training method is described in Sect. 7.2. We applied our extraction method to each of these RNNs, with a time limit of 30 seconds (after which the last L^* hypothesis is returned) and initial split depth and counterexamples as described in Sect. 7.3. The results of these experiments are shown in Table 1. Each row in the table represents the average of 10 extractions.

Most extractions completed before the time limit, having reached equivalence.[21] We compared the extracted automata against the networks on their training sets and on 1000 randomly generated word samples for each of the word-lengths 10,50,100 and 1000. In all settings (hidden size, alphabet size, and DFA size) where the RNNs achieved 100% test set accuracy, our extraction obtained DFAs with perfect accuracy against their RNNs. For two RNNs which reached 99% accuracy, our extraction achieved 99% accuracy against the RNNs, and for the two RNNs with less than 99% accuracy our extraction achieved on average $\geq$ 88% accuracy for all evaluation sets.

[21] Though this is not necessarily a guarantee of true equivalence, it does generally indicate strong similarity.

7.5 Comparison with a-priori quantisation

In their 1996 paper, Omlin and Giles suggested partitioning the network state space by dividing each state dimension into q equal intervals, with q being the *quantisation level*. We tested this method on each of our small regular language RNNs (Sect. 7.4.2), with $q = 2$ and a time limit of 500 seconds to avoid excessive memory consumption.[22]

In many cases, we found that 500 seconds was not enough time for this method to extract a complete DFA from our RNNs.[23] To enable some comparison, we allow the method to return incomplete DFAs, i.e. DFAs in which some transitions are missing, and we move from evaluating just the accuracy of a DFA to evaluating both its accuracy and its *coverage*, with *coverage* being the fraction of samples for which it has a full transition path.

We provide the results of extracting with this method in Table 2, which uses the exact same RNNs as in Table 1.

The extracted DFAs are very large—with some even having 100, 000 states–and yet their coverage of sequences of length 1, 000 and even 100 tends to zero as the RNN complexity (state size d_s, or RNN target language complexity) increases. For the covered sequences, the extracted DFA's accuracy was often very high (99+%), suggesting that quantisation—while impractical—is sufficiently expressive to describe a network's state space. However, it is also possible that the sheer size of the quantisation (2^{50} for our smallest RNNs, and more for others) simply allowed each explored R-state its own A-state, giving high accuracy just by observation bias (only covered sequences could have their accuracy checked).

This is in contrast to our method, which always returns complete DFAs,[24] and which consistently extracted accurate DFA from the same networks in a fraction of the time and memory used by the plain quantisation approach. This is because our method maintains from a very early point in extraction a complete DFA $\mathcal{A}$ that constitutes a constantly improving approximation of the considered RNN.

7.6 Comparison with k-Means clustering

Next, we implemented a simple k-means clustering and extraction approach and applied it to the same networks from Sect. 7.4.2 with varying k.

Specifically, for each RNN, we sampled $N = 5000$ unique prefixes from its train set, computed the states reached from them in the RNN, and used k-means clustering to partition the state space according to those states for each of $k = 1, 6, 11, ..., 31$.[25] We then

[22] LSTMs have unbounded state space, which makes quantisation challenging. Specifically for $q = 2$ however, we just split each dimension along 0.

[23] This is because the quantisation method, even for the smallest possible q ($q = 2$), generates far more partitions than can be traversed within the time limit (q^{d_s} where d_s is the RNN state size, and $d_s \geq 50$ in our case). Note that this is in contrast to our method: our method only applies this quantisation method on d initial dimensions for user-defined d (typically ≤ 10), before continuing with only very gentle refinements as needed.

[24] Provided L* manages to generate at least one equivalence query before the time limit, which we observe to always happen in practice (usually taking $\leq$1 second).

[25] Using `sklearn.cluster.KMeans`.

mapped the transitions of each partitioning to create 7 potential DFAs, and evaluated each one against the RNN on its 1000-sample test set to choose the best.

k-means has a well defined and ‘reasonably quick’ stopping condition: the number of RNN states visited, and the number of clusters to be created and traversed from them, is given as input to the extraction.[26] Hence for this extraction we do not use a time limit, allowing the method to extract all of its potential DFAs in full, evaluate them, and return the best DFA. As done for the other methods, we measure for k-means the total time from beginning the extraction until a single final DFA is returned. In particular, this covers sampling once all 5000 RNN states (generally <10 seconds), making a k-state DFA from these RNN states by applying k-means clustering to them (taking from <1 to ~50 seconds for each k, depending on the states and on k), and finally choosing the best DFA by evaluating on the test set (generally <10 seconds). We note that the bulk of the extraction time is spent in clustering the sampled states into different numbers of clusters k.

In Table 3 we report the results of these extractions. In particular, we report the time (in seconds) spent on each full extraction, the number of clusters k used for each best DFA, each DFA’s size $|Q_{\mathcal{A}}|$ after minimisation, and of course each extracted DFA’s accuracy against the same sample sets as before (i.e., as in 1).

For the GRU networks trained on smaller DFAs (which reached 100% test-set accuracy), k-means clustering is as successful as our method, often returning a DFA with perfect or near-perfect accuracy against the target RNN. For the LSTMs and the larger DFAs however, our method obtains far higher accuracy, and often in less time. The difference in success on the LSTMs and GRUs is curious, we leave this question open in this work.

7.7 Comparison with random sampling for counterexample generation

For 3 of the Tomita grammars (specifically, Tomitas 3,4, and 7), the first counterexample returned in our extraction (Sect. 7.4.1) was actually created by the initial random sampling. Moreover, for all of the Tomita grammars, answering all equivalence queries using a random sampler alone (with up to 1, 000 samples per query) was successful at extracting the grammars from the RNNs, and this was also true for many of the languages considered in Sect. 7.4.2. The termination is slightly slower than our own, to allow for sampling many potential counterexamples before accepting the L* hypothesis, but still fast enough to make random sampling seem appealing (the method spent ≈ 10 seconds on each Tomita grammar). Indeed, Mayr&Yovine (2018) even suggest such a method in their recent work, analysing it from a PAC perspective.

Given this, the question may arise whether there is at all merit to the exploration and refinement of abstractions of the network, as opposed to a simple random sampling approach to counterexample generation for L* equivalence queries.

In this section we show the advantage of our method for counterexample generation, through the example of balanced parentheses (BP): the language of sequences with correctly balanced parentheses over the alphabet `()a-z`. BP is not a regular language, but the attempt to approximate it with DFAs, and in particular the search for counterexamples to proposed DFAs, proves informative. In particular, when sampling the tokens with uniform

[26] This is in contrast to our method, which may continue to refine its hypothesis indefinitely without ever reaching equivalence (consider for example an RNN that has learned a non-regular language), or quantisation, which creates so many partitions in modern-sized architectures (2^{50} even for our smallest networks and quantisation level) that it cannot be used without adding some time or size limit.

distribution, the probability of randomly generating a sequence with nested and correctly balanced parentheses over the BP alphabet is very low. This prevents the random sampler from finding counterexamples to L*'s proposed automata, each of which accept balanced parentheses to a bounded depth (see Examples in Fig. 1), highlighting the advantage of our approach.

We train one GRU and one LSTM network on BP, each with 2 layers and hidden dimension 50. We extract DFAs from these networks using L*, generating counterexamples once with our method and once with a random counterexample generator. The random counterexample generator works as follows: for each equivalence query, it randomly samples sequences over the input alphabet Σ until a counterexample (sample on which $\mathcal{A}$ and the RNN disagree) is found. In particular, for each length $l = 1, 2, 3, ...$ and increasing until a counterexample is found, it generates and compares up to 1000 random samples of length l, with uniform distribution.

We allowed each method 400 seconds[27] to extract an automaton from networks trained to 100% train set accuracy. The accuracy of these extracted automata against the original networks on their training sets is recorded in Table 4, as well as the maximum parentheses nesting depth the L* proposed automata reached during extraction.

We list the counterexamples and counterexample generation times for each of the BP network extractions in Table 5. Note the succinctness and the generation speed of the counterexamples generated by our method: excluding two samples at the end of the GRU extraction, they are clear of the 'neutral' tokens `a-z` and of repeating parentheses (e.g., `()()()`), as these were not necessary to advance the automata learned by L* (Fig. 1). In contrast, the random sampling method has difficulty finding legally balanced sequences, taking a long time to find counterexamples at all, and including many 'uninformative' neutral tokens in its results.

The extracted DFAs themselves were also pleasing: each subsequent DFA proposed by L* for this language was capable of accepting all words with balanced parentheses of increasing nesting depth, as pushed by the counterexamples provided by our method (Fig. 1). In addition, for the GRU network trained on BP, our extraction method managed to push past the limits of the network's 'understanding'—finding the point at which the network begins to overfit to the particularly deeply-nested examples in its training set, and extracting the slightly more complicated automaton seen in Fig. 2.

7.8 Additional variations on our method

We show the necessity of the initial split and counterexamples for our method, the effect of running extraction for a longer time (if it has not completed), and support the decision to return the final L* hypothesis $\mathcal{A}$ as opposed to the final abstraction $A_{R,p}$ whenever the extraction has not reached equivalence in time.

Removing the Initial Split Heuristics

We run the extraction again on the same RNNs as in Table 1, but this time setting the initial split depth to 1 and the number of random samples before accepting a hypothesis to 0. We report the results in Table 6. The average number of counterexamples ("#c-exs") per extraction drops to almost 0 for most settings, meaning the majority L* initial hypotheses are accepted immediately by the method (without counterexamples). The number of

[27] Timed using the `clock()` method from python's `time` module.

Table 1 Results for DFA extracted using our method from 2-layer GRU and LSTM networks with various state sizes, trained on random regular languages of varying sizes and alphabets. Each row in each table represents 10 experiments with the same parameters (network hidden-state size d_s, alphabet size $|\Sigma|$, and minimal target DFA size $|Q_T|$). In each experiment, a random DFA is generated and an RNN is trained on it, after which a DFA is extracted from and compared to the RNN. The column $|Q_A|$ represents the size of the final returned DFA, *#c-exs* describes how many counterexamples were used during extraction, *max |c-ex|* describes their maximum length, and *RNN Acc.* is the accuracy of the trained RNN on its test set. Each column represents the average of the 10 experiments, except for *max |c-ex|* which gives the overall maximum counterexample used across all RNNs in that row. Each extraction was run with a time limit of 30 seconds, and whenever an extraction timed out the last automaton proposed by L* was taken as the extracted automaton. For the accuracies on the different lengths, 1000 random words of each length were sampled and evaluated, and for the accuracy on the training set all of the RNN's training set was evaluated (i.e., comparing DFA against RNN)

Extraction from LSTM networks — Our method												
						max	RNN	Average extracted DFA accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_A\|$	#c-exs	|c-ex|	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	6.82	10.8	2.5	12	1.0	1.0	1.0	1.0	1.0	1.0
100	3	5	4.09	5.0	2.0	10	1.0	1.0	1.0	1.0	1.0	1.0
500	3	5	10.03	5.0	1.8	10	1.0	1.0	1.0	1.0	1.0	1.0
50	5	5	16.66	19.9	3.0	8	0.99	0.99	0.99	0.99	0.99	0.99
100	5	5	12.53	6.6	2.4	12	1.0	1.0	1.0	1.0	1.0	1.0
500	5	5	18.34	5.0	2.3	8	1.0	1.0	1.0	1.0	1.0	1.0
50	3	10	30.97	67.8	5.2	9	0.91	0.92	0.88	0.88	0.88	0.89
100	3	10	21.15	23.4	4.6	18	0.99	1.0	0.99	0.99	0.99	0.99
500	3	10	16.27	10.0	4.0	9	1.0	1.0	1.0	1.0	1.0	1.0
Extraction from GRU networks — Our method												
50	3	5	4.85	5.0	2.0	13	1.0	1.0	1.0	1.0	1.0	1.0
100	3	5	3.22	5.0	2.0	10	1.0	1.0	1.0	1.0	1.0	1.0
500	3	5	6.29	5.0	1.8	10	1.0	1.0	1.0	1.0	1.0	1.0
50	5	5	15.98	11.8	2.8	16	1.0	1.0	1.0	1.0	1.0	1.0
100	5	5	7.22	5.0	2.4	18	1.0	1.0	1.0	1.0	1.0	1.0
500	5	5	12.3	4.9	2.1	8	1.0	1.0	1.0	1.0	1.0	1.0
50	3	10	29.09	76.2	5.6	11	0.94	0.97	0.92	0.92	0.92	0.93
100	3	10	13.88	23.3	4.7	20	1.0	1.0	1.0	1.0	1.0	1.0
500	3	10	12.01	10.0	3.8	9	1.0	1.0	1.0	1.0	1.0	1.0

states in the returned automata is often smaller than in the target, and their accuracy drops significantly.

This shows that indeed our method must be coupled with some heuristics to prevent acceptance in the early stages, during which both the abstraction and the L* hypothesis only reflect the RNN's classification on very short sequences, and have not yet diverged.

Timing out: Using the Abstraction, and Increasing the Time Limit

When we increase $|\Sigma|$ and $|Q|$ of our randomly generated target DFAs to 10, the training routine used in this work is not sufficient for the RNNs with dimensions $d_s = 50$ and $d_s = 100$ to train perfectly, and they reach on average $< 0.8\%$ test set accuracy on their target languages. For these RNNs, we observe that our extraction method does not reach

Table 2 Results for DFA extracted using a simple partitioning of the RNN state space, in which each state dimension is split into $q = 2$ equal segments (positive and negative). The extractions were applied to the same RNNs as in Table 1, with each row representing 10 experiments as before. $|Q_A|$ again reports the (average) number of states in the extracted DFAs, though this time it is rounded for clearer presentation. The extractions were run with a time limit of 500 seconds. This time, instead of reporting only the accuracy of the extracted DFAs against their RNNs on different samples sets, we also report their coverage: the fraction of samples for which the DFAs have a classification at all (i.e., do not have missing transitions). The accuracy is computed only on covered sequences, and we write report the accuracy as −1 when all extractions in the row have 0 coverage for that set. For example: 1.0×0.12 tells us that only 12% of samples have full transitions in the extracted DFA, but that for those 12%, the DFA accuracy against the RNN is perfect

Extraction from LSTM Networks — Quantisation

					RNN	Extracted DFA Accuracy × Coverage				
d_s	$\|Q_T\|$	$\|\Sigma\|$	Time (s)	$\|Q_A\|$	Acc.	l=10	l=50	l=100	l=1000	Train
50	5	3	100.17	16868	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
100	5	3	237.06	40088	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
500	5	3	469.64	60295	1.0	1.0×1.0	1.0×0.53	1.0×0.42	1.0×0.21	1.0×0.59
50	5	5	283.83	29472	0.99	0.99×1.0	0.99×1.0	0.99×1.0	0.99×1.0	0.99×1.0
100	5	5	469.88	47873	1.0	1.0×1.0	1.0×0.94	1.0×0.91	1.0×0.79	1.0×0.95
500	5	5	503.68	39508	1.0	1.0×0.03	− 1×0.0	− 1×0.0	− 1×0.0	1.0×0.08
50	10	3	434.75	77538	0.91	0.98×1.0	0.94×0.58	0.97×0.44	0.94×0.31	0.97×0.65
100	10	3	500.62	83402	0.99	1.0×1.0	1.0×0.46	1.0×0.32	1.0×0.02	1.0×0.55
500	10	3	502.84	64720	1.0	1.0×1.0	− 1×0.0	− 1×0.0	− 1×0.0	1.0×0.12
Extraction from GRU Networks — Quantisation										
50	5	3	102.48	21359	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
100	5	3	239.93	49203	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
500	5	3	501.37	82933	1.0	1.0×1.0	1.0×0.22	1.0×0.13	1.0×0.0	1.0×0.35
50	5	5	335.37	42008	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
100	5	5	500.34	60089	1.0	1.0×0.98	1.0×0.77	1.0×0.67	1.0×0.41	1.0×0.8
500	5	5	502.31	49206	1.0	1.0×0.02	− 1×0.0	− 1×0.0	− 1×0.0	1.0×0.08
50	10	3	500.42	100417	0.94	1.0×1.0	0.99×0.4	0.99×0.27	0.98×0.14	0.99×0.51
100	10	3	500.42	103488	1.0	1.0×1.0	1.0×0.51	1.0×0.34	1.0×0.06	1.0×0.58
500	10	3	501.93	82378	1.0	1.0×1.0	− 1×0.0	− 1×0.0	− 1×0.0	1.0×0.12

equivalence in the provided time. In particular, the L* hypotheses grow very large, and the extraction often times out while increasing the *observation table*: the internal table of sequence labels maintained by L* between equivalence queries (i.e., the majority time is spent on refining $\mathcal{A}$ after each new counterexample).

In all of our experiments, whenever we run out of time, we return the last L* hypothesis $\mathcal{A}$ as the extracted automaton. In this section, we check how much this hypothesis improves as we increase the time limit, and evaluate the option of returning the last abstraction $A_{R,p}$ used by our method instead.

Table 7 shows a set of extractions from imperfectly trained RNNs, trained with the same training routine and number of repetitions as before. We make 10 DFAs all with $|Q| = |\Sigma| = 10$ and on each DFA train 4 2-layer RNNs: 2 GRUs and 2 LSTMs, each with hidden state sizes $d_s = 50$ and $d_s = 100$. We then extract from each RNN with 5 different time limits ranging from 50 to 1000 seconds. This means that overall Table 7

Table 3 Results for DFA extracted using k-means clustering from the same 2-layer GRU and LSTM networks considered in Table 1, i.e., each row represents the average results of 10 experiments as before, and considers the exact same trained RNNs. The extractions did not have a time limit, instead, the number of states sampled was set to 5000 and the k values considered were $k = 1, 6, 11, ..., 31$. The accuracies were evaluated on the same sample sets as in Table 1

Extraction from LSTM networks — k -means clustering											
						RNN	Average extracted DFA accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_{\mathcal{A}}\|$	k	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	48.93	4.5	25.5	1.0	0.85	0.85	0.85	0.85	0.85
100	3	5	69.85	3.9	20.0	1.0	0.82	0.81	0.81	0.81	0.81
500	3	5	274.96	5.0	18.5	1.0	0.87	0.85	0.86	0.85	0.86
50	5	5	53.13	3.3	18.5	0.99	0.8	0.8	0.8	0.79	0.8
100	5	5	84.48	4.3	18.0	1.0	0.83	0.83	0.83	0.82	0.83
500	5	5	289.63	5.6	24.0	1.0	0.98	0.97	0.98	0.97	0.98
50	3	10	52.74	6.4	19.5	0.91	0.67	0.66	0.65	0.67	0.67
100	3	10	63.06	12.0	27.5	0.99	0.8	0.75	0.75	0.74	0.76
500	3	10	250.99	11.6	28.0	1.0	0.93	0.89	0.89	0.89	0.9
Extraction from GRU Networks — k -means Clustering											
50	3	5	21.95	5.0	14.5	1.0	0.99	1.0	1.0	1.0	1.0
100	3	5	24.89	4.9	12.0	1.0	1.0	1.0	1.0	1.0	1.0
500	3	5	85.46	5.0	13.5	1.0	0.99	1.0	1.0	1.0	1.0
50	5	5	22.74	5.4	20.0	1.0	1.0	1.0	1.0	1.0	1.0
100	5	5	29.13	5.0	18.5	1.0	1.0	1.0	1.0	1.0	1.0
500	5	5	91.68	5.1	19.0	1.0	1.0	1.0	1.0	1.0	1.0
50	3	10	27.98	12.4	28.5	0.94	0.87	0.84	0.84	0.83	0.85
100	3	10	27.15	10.5	31.0	1.0	0.96	0.94	0.94	0.94	0.94
500	3	10	92.63	10.0	28.5	1.0	0.99	0.98	0.99	0.98	0.99

shows results for 10 DFAs, 40 RNNs, and 200 extractions (each row represents 10 extractions).[28]

Alongside the details of the last L* hypothesis $\mathcal{A}$, we also report the size of our final partitioning p (i.e., number of partitions it divides the state space into), the size (after minimisation) of the abstraction $A_{R,p}$ it defines, and the accuracy of $A_{R,p}$ against its target RNN.

The results show clearly that the L* hypothesis is the preferable choice when the extraction does not complete. Effectively, the partitioning p and abstraction $A_{R,p}$ it defines act as a tool for refining the L* hypotheses, and not so much the other way around.[29]

[28] We ran each extraction in itself, for example each RNN's 1000 second extraction was not merely a continuation of its 50 second extraction but a full extraction in its own.

[29] This may be because, whenever the equivalence checking finds a disagreement, it first checks for the possibility of a counterexample to L* before checking whether the abstraction needs to be refined. However, the difference may also come through the more long-sighted nature of L*: internally, L*maintains a growing list of prefixes and suffixes, all combinations of which its hypotheses have to classify correctly. In contrast, traversing a partitioning of the state space only looks as far as the immediate classification and transitions of each visited partition. L*'s advantage here is also its curse: learning a DFA with L* has polynomial time complexity in the size of the DFA, whereas traversing a partitioning is linear in the number of partitions.

The results also show that, for these non-terminating extractions, it is ‘difficult’ to improve beyond the automata reached in the early stages: increasing the extraction time to 100, 200, and even 1000 seconds gives only a small increase in accuracy each time. We also see that the number of counterexamples used per extraction grows very slowly with the increase in time, i.e., more time does not significantly increase the number of hypotheses presented by L*.

Analysing the time spent by the extraction reveals that L* gets ‘stuck’ refining the large hypotheses it creates, generating many membership queries without reaching new equivalence queries. The average equivalence query time across all experiments is <1.5s, whereas the maximum hypothesis refinement time in each experiment grew to over 10, 48, 60, 170 and 314 seconds for each of the time limits respectively.[30] A more efficient implementation of L*, or possibly an approximation of it, would be an important step towards scaling this method.

7.9 Discussion

7.9.1 Adversarial inputs

Balanced Parentheses Excitingly, the penultimate counterexample returned by our method during the extraction of balanced parentheses (BP) in Sect. 7.7 is an *adversarial input*: a sequence with unbalanced parentheses that the network accepts (despite its target language accepting only sequences with balanced parentheses). This input is found in spite of the network's seemingly perfect behavior on its set of 44000+ training samples. Note that the random sampler did not manage to find such samples.

Inspecting the extracted automata indeed reveals an almost-but-not-quite correct DFA for the BP language (Fig. 2). The RNN overfit to random peculiarities in the training data and did not learn the intended language, and our extraction method managed to discover and highlight an example of this ‘incorrect’ behaviour.

Email Addresses For a seemingly perfect LSTM-acceptor trained on the regular expression

$$[a-z][a-z0-9]*@[a-z0-9]+.(\mathrm{com}|\mathrm{net}|\mathrm{co}.[a-z][a-z])\$$$

(simple email addresses over the 38 letter alphabet {a-z, 0-9, @, .}) to 100% accuracy on a 40,000 sample train set and a 2,000 sample test set, our method quickly returned the counterexamples seen in Table 8, showing clearly words that the network misclassified (e.g., `25.net`). We ran extraction on this network for 400 seconds, and while we could not extract a representative DFA in this time,[31] our method did show that the network learned a far more elaborate (and incorrect) function than needed. In contrast, given a 400 second overall time limit, the random sampler did not find any counterexample beyond the provided one.

[30] I.e., for example, each one of the 1000 second extractions spent at least 314 seconds on at least one hypothesis refinement.

[31] A 134-state DFA $\mathcal{A}$ was proposed by L* after 178 seconds, and the next refinement to $\mathcal{A}$ (initiated 4.43 seconds later) timed out. The accuracy of the 134-state DFA on the train set was nearly random. We suspect that the network learned such a complicated behavior that it simply could not be represented by any small DFA.

We note that our implementation of k-means clustering and extraction had no success with this network, returning a completely rejecting automaton (representing the empty language), despite trying k values of up to 100 and using all of the network states reached using a train set with a 50:50 ratio between positive and negative samples.

Beyond demonstrating the capabilities of our method, these results also highlight the brittleness in generalisation of trained RNNs, and suggest that evidence based on test-set performance should be interpreted with extreme caution. This reverberates the results of Gorman and Sproat (2016), who trained a neural architecture based on a multi-layer LSTM to mimic a finite state transducer (FST) for number normalisation. They showed that the RNN-based network, trained on 22M samples and validated on a 2.2M sample development set to 0% error on both, still had occasional errors (though with error rate < 0.0001) when applied to a 240,000 sample blind test set.

7.9.2 Limitations and discussion

L* ***Optimisation*** One limitation of the method shown in this work is the polynomial time complexity of L*, which becomes a significant issue as the extracted DFA grows (see Sect. 7.8, *Timing out*). Applying our method with more efficient variants of L*, such as the TTT algorithm presented by Isberner et al. (2014), may yield better results.

L* ***and Noise*** Whenever applied to an RNN that has failed to generalise properly to its target language, our method soon finds several adversarial inputs, and begins to build very large DFAs. As noted above, to L*'s polynomial complexity and intolerance to noise, this quickly becomes extremely slow.[32]

Of course by the nature of L*, any complexity in the final returned automaton is only a result of the inherent complexity of the RNN's learned behaviour, and so we may say that this result is not necessarily incorrect. Nevertheless, it limits us, and seeking a way to recognise and overcome 'noise' in the given network's behaviour is an interesting avenue for future work.

Adversarial Inputs On the bright side, this same limitation does demonstrate the ease with which our method identifies imperfectly trained networks. These cases are annoyingly frequent: for many RNN-acceptors with 100% train and test accuracy on large test sets, our method was able to find many simple misclassified examples (Sect. 7.9.1).

Note on Heuristics In Sect. 3, we note that existing works consider multiple RNNs, and then must choose the best according to a heuristic. Our method can also be seen as considering multiple DFAs and abstractions, with the equivalence query being the 'heuristic' deciding whether to terminate or consider more DFAs/abstractions. We highlight here our differences. First, in our method, the DFAs considered are always minimal (thanks to L*), and the abstractions used can be much smaller than in other methods. In particular the abstractions can be small because they are dynamically refined by the method on an as-needed basis, and so can afford to be very coarse: 'missed partitions' are discovered and fixed automatically by the method. Secondly, even when the refinement eventually creates a very large abstraction, the equivalence query is applied 'on-the-fly', meaning it can cut off and return counterexamples/refine the abstraction even before $A_{R,p}$ has been fully mapped.

[32] This happened for example to our balanced-parentheses LSTM network, which timed out during L* refinement after the last counterexample.

Table 4 Accuracy of extracted automata against their networks, which were trained to 100% training accuracy on the balanced parentheses (BP) language. The comparisons were done on the training sets of the networks. The maximum nesting depth the extracted automata reached while still behaving as BP is recorded (the GRU network ultimately returned a more complex automaton than the one extracted from the LSTM network, but this automaton no longer behaved as BP and so we have no reasonable measure for its 'depth'). The hidden size d_s and the number of layers in each network is also noted. (For the LSTM network, this is the size of both the memory and the cell vectors, meaning the total hidden size of a single cell in this network is twice as big as the value listed.)

Network	Accuracy on train set		Max nesting depth		d_s	#Layers
	Our method	Random	Our method	Random		
GRU	99.98	87.12	8	2	50	2
LSTM	99.98	94.19	8	3	50	2

8 Learning from only positive samples

Thus far, the method presented here can be used to learn a DFA from a set of positive and negative samples: we train an RNN-acceptor to generalise from them, and then extract a DFA from it.

However, we can also use our method to learn a DFA from positive samples only, by training an RNN using a language-modeling objective, and then extracting from an RNN-acceptor interpretation of it. Such RNNs are trained only on positive samples, attempting to model their distribution rather than classify what is or isn't in the language:

A *language-model RNN (LM-RNN)* over an alphabet Σ and end-of-sequence symbol $\$ \notin \Sigma$ is an RNN with classification component $f_R : S_R \to [0,1]^{\Sigma \cup \{\$\}}$ defining for every RNN-state a distribution over $\Sigma \cup \{\$\}$. An LM-RNN effectively defines for every sequence $w \in \Sigma^*$ and token $\sigma \in \Sigma \cup \{\$\}$ the probability of sampling σ after seeing w: $P(\sigma|w) = f_R(\hat{g_R}(w))(\sigma)$.

LM-RNNs can be interpreted as classifiers by taking a threshold t and defining that they accept exactly the set of sequences $w = w_1 w_2 ... w_n \in \Sigma^*$ which satisfy: 1. $P(\$|w) \geq t$, and 2. for every strict prefix $w' = w_1 w_2 ... w_i$, $i < n$ of w, $P(w_{i+1}|w') \geq t$. This interpretation recently appears as *locally ϵ-truncated support* in the work of Hewitt et al. (2020), with $\epsilon = t$.

LM-RNNs can therefore be adapted for extraction as classifiers by defining each of their states as accepting or rejecting according to the probability they assign to \$, and introducing an artificial sink-reject state v[33] that is entered whenever a sequence transitions through a token with too low probability. Formally:

Making an RNN acceptor Let R be an LM-RNN with reachable state space $S \subsetneq \mathbb{R}^{d_s}$, initial state $h_{0,R} \in S$, update function g_R, and classification function f_R. Let $t \in [0,1]$ be a threshold and let $v \in \mathbb{R}^{d_s} \setminus S$ be a vector that cannot be reached in R from any input sequence.[34] To create an RNN-acceptor R' from R, we build the components $h'_{0,R} = h_{0,R}$,

$$f'_R(s) = \begin{cases} Acc & : f_R(s)(\$) \geq t \\ Rej & : \text{else} \end{cases}, \text{ and } g'_R(s,\sigma) = \begin{cases} v & : f_R(s)(\sigma) < t \text{ or } s{=}v \\ g_R(s,\sigma) & : \text{else} \end{cases}.$$

[33] I.e., an externally maintained state $v \notin S_R$.

[34] For most RNN architectures, finding such a vector v is easy from the architecture definition. For instance, for LSTMs and GRUs, $v = \bar{2}$ is sufficient: both have at least some of their state dimensions bound to the range $[-1, 1]$.

Table 5 Extraction of automata from GRU and LSTM networks trained to 100% accuracy on the training set for the language of balanced parentheses over the 28-letter alphabet `a-z`, `(`, `)`. Each table shows the counterexamples and the counterexample generation times for each of the successive equivalence queries posed by L*during extraction, for both our method and a brute force approach. Generally, each successive equivalence query from L*for either network was an automaton classifying the language of all words with balanced parentheses up to nesting depth n, with increasing n. The exception to this comes after the penultimate counterexample in the extraction from the GRU network, in which a word with unbalanced parentheses was returned as a counterexample to L*(whose automaton currently rejects it)

Refinement-based vs. brute-force counterexample generation on the balanced parentheses language			
Refinement based		Brute Force	
Counterexample	Time (seconds)	Counterexample	Time (seconds)
GRU			
))	1.1	))	0.4
(())	1.2	(()i)ma	32.6
((()))	2.1		
(((())))	3.1		
((((()))))	3.8		
(((((())))))	4.4		
((((((()))))))	6.6		
(((((((())))))))	9.2		
((((((((v())))))))	10.7		
((((((((a()z)))))))))	8.3		
LSTM			
))	1.4	))	1.5
(())	1.6	tg(gu()uh)	57.5
((()))	3.1	((wviw(iac)r)mrsnqqb)iew	231.5
(((())))	3.1		
((((()))))	3.4		
(((((())))))	4.7		
((((((()))))))	6.3		
(((((((())))))))	9.2		
((((((((()))))))))	14.0		

The new RNN-acceptor R' can now be passed directly to our algorithm for extraction.

When the language is 'small'—in the sense that uniformly sampled sequences are likely to be rejected—sampling sequences according to the RNN's distribution is likely to hit a sample that has not yet been considered by L*. Hence here random sampling according to the RNN's distribution can be a useful augmentation to the equivalence query—though this can also create overly long counterexamples (Sect. 8.1.3).

This approach—training an LM-RNN, adapting it as a classifier, and then extracting from it with the method presented in this work—has been recently applied by Yellin&Weiss (2021) to elicit a sequence of DFAs from trained LM-RNNs, as part of a process for learning context free grammars from trained RNNs.

Note Extracting from LM-RNNs requires some hyperparameter tuning, as changing the threshold t changes the set of sequences accepted by R'.

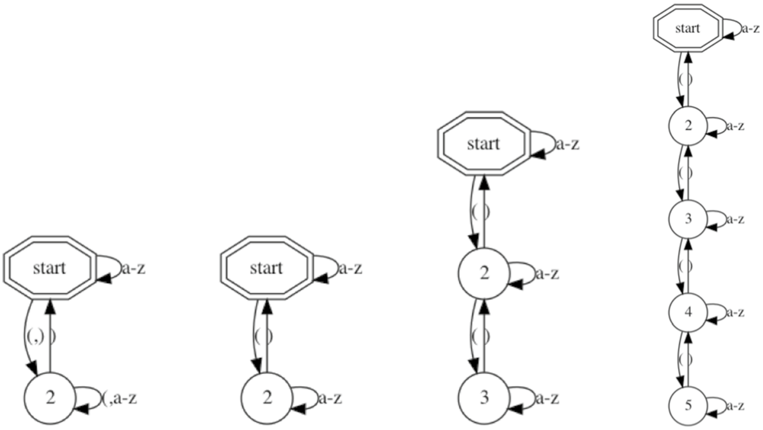

Fig. 1 Select automata of increasing size for recognising balanced parentheses over the 28 letter alphabet `a-z`, (,), up to nesting depths 1 (flawed), 1 (correct), 2, and 4, respectively. In this and in all following automata figures, the initial state is an octogon, accepting states have a double border, and sink reject states (rejecting states whose transitions all lead back to themselves) are not drawn

8.1 Proof of concept

We provide a small number of example extractions from LM-RNNs trained on non-regular languages, observing the ability of the method to generate increasingly 'complex' DFA approximations of the targets. More examples are also present in Yellin&Weiss (2021).

8.1.1 $a^n b^n$

We train a 2-layer LSTM-based LM-RNN with hidden dimension $d_s = 50$ on positive samples from the language $a^n b^n = \{a^i b^i \mid i \in \mathbb{N}\}$.[35] We then interpret it as an RNN-acceptor as described above, and extract from it using our extraction method, with $t = 0.1$ and a time limit of 400 seconds.

As expected, the extraction generates a series of DFA approximations of the non-regular target language, we present some of these in Fig. 3. The extraction ultimately reached DFAs approximating $a^n b^n$ up to $n \leq 20$ before timing out, with the majority of time spent on refining the L* hypotheses, which grew slower as the DFA grew: the final hypotheses returned by L* took 46, 54, and 63 seconds each to generate after their 'prompting' counterexamples, and the next L* refinement after them also timed out after 53 seconds (meanwhile, each of the counterexamples took < 5 seconds to generate). This result suggests that this method may benefit from applying a more efficient implementation of L*, such as the TTT algorithm of Isberner et al. (2014).

[35] 20 epochs on 5000 non-unique samples of average length 50.

Fig. 2 Automaton with vague resemblance to the BP automata of Fig. 1, but no longer representing a language of balanced parentheses up to a certain depth. (Showing how a trained network may be overfitted past a certain sample complexity.)

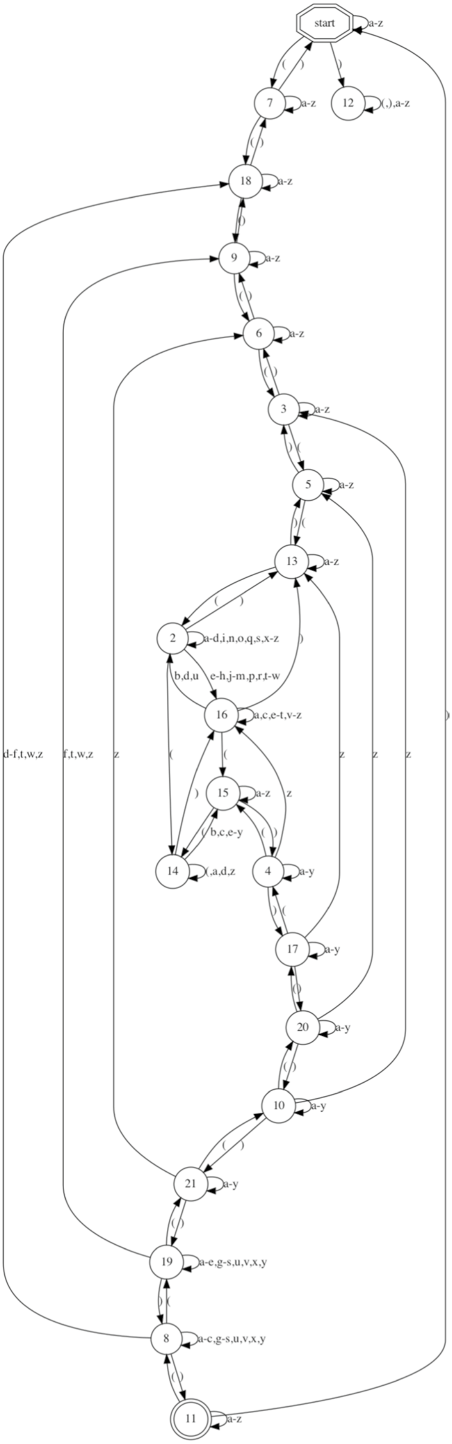

Table 6 Extracting with our method from the same RNNs as in Table 1, but this time without the initial heuristics as described in Sect. 7.3. The extraction time is reduced significantly, along with the accuracy: L*'s first hypotheses are frequently very small, and without the aggressive initial state-splitting and random samples, the abstraction is too coarse to find counterexamples

Extraction from LSTM networks — Our method (No Heuristics)												
						max	RNN	Average extracted DFA accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_A\|$	#c-exs	|c-ex|	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	0.26	2.2	0.2	3	1.0	0.63	0.64	0.63	0.63	0.64
100	3	5	0.21	2.1	0.1	3	1.0	0.64	0.65	0.63	0.64	0.64
500	3	5	0.23	2.1	0.1	3	1.0	0.64	0.65	0.63	0.64	0.64
50	5	5	0.25	1.8	0.0	–	0.99	0.74	0.74	0.74	0.74	0.75
100	5	5	0.2	1.8	0.0	–	1.0	0.74	0.74	0.74	0.74	0.75
500	5	5	0.29	1.8	0.0	–	1.0	0.74	0.74	0.74	0.74	0.75
50	3	10	9.52	15.8	1.6	8	0.91	0.66	0.65	0.65	0.65	0.66
100	3	10	0.65	2.4	0.3	5	0.99	0.58	0.56	0.57	0.57	0.58
500	3	10	0.16	1.7	0.0	–	1.0	0.55	0.54	0.54	0.55	0.55
Extraction from GRU Networks — Our Method (No Heuristics)												
50	3	5	0.16	2.1	0.1	3	1.0	0.64	0.65	0.63	0.64	0.64
100	3	5	0.16	2.0	0.0	–	1.0	0.66	0.67	0.66	0.66	0.67
500	3	5	0.2	2.0	0.0	–	1.0	0.66	0.67	0.66	0.66	0.67
50	5	5	0.19	1.8	0.0	–	1.0	0.74	0.74	0.74	0.74	0.75
100	5	5	0.18	1.8	0.0	–	1.0	0.74	0.74	0.74	0.74	0.75
500	5	5	0.21	1.8	0.0	–	1.0	0.74	0.74	0.74	0.74	0.75
50	3	10	0.13	1.7	0.0	–	0.94	0.55	0.55	0.54	0.55	0.56
100	3	10	0.16	1.7	0.0	–	1.0	0.55	0.54	0.54	0.55	0.55
500	3	10	0.31	2.5	0.3	7	1.0	0.59	0.59	0.59	0.59	0.6

8.1.2 Dyck-3

We consider the language Dyck-3 with 3 additional neutral tokens, i.e.: correctly balanced sequences over the alphabet `{}()[]abc`. For example, `{}a(b[])c` is in the language, but `([)]` and `())` are not.

We use a 2-layer GRU with dimension 50, and train it as a language model on 50000 non-unique samples of lengths 1-100 from Dyck-3 for 20 epochs, reaching a train, test, and validation cross-entropy loss of ≈1.7. We interpret the GRU as a classifier using rejection threshold $t = 0.01$, and extract from it using our method with a time limit of 400 seconds and initial split depth $d = 10$.[36]

[36] We also augmented the equivalence queries with random counterexample generation using LM-sampling, to be considered before accepting any DFA. However, this was never used: our abstraction-based method rejected every hypothesis before reaching this stage.

Table 7 Extracting with our method from 2-layer GRUs and LSTMs trained imperfectly on DFAs with size $|\Sigma| = |Q| = 10$, varying RNN hidden size (d_s) and extraction time limit. Each row represents the average of 10 experiments, with average DFA ($|Q_{\mathcal{A}}|$, $|Q_{A_{R,p}}|$) and final partitioning ($|p|$) sizes rounded for space. We report both the accuracy (against the RNN) of the final L* hypothesis, $\mathcal{A}$, and the abstraction $A_{R,p}$ used by the method to find counterexamples to each $\mathcal{A}$. We see that the final L* hypothesis is clearly the superior option when extraction has not terminated. Unfortunately, we also see that the accuracy does not increase well with more time, this is because the hypothesis generation (time from counterexample to new hypothesis) grows slower with each iteration

	Time					RNN	$\mathcal{A}$ Accuracy			$A_{R,p}$ Accuracy		
d_s	Limit	$\lvert Q_{\mathcal{A}}\rvert$	$\lvert Q_{R,p}\rvert$	$\lvert p\rvert$	#c-exs	Acc.	l=10	l=1000	Train	l=10	l=1000	Train
Extraction from LSTM networks — Our method (Time limits)												
50	50	116	157	1029	5.9	0.74	0.84	0.85	0.85	0.66	0.66	0.66
	100	178	163	1031	6.8	0.74	0.86	0.86	0.87	0.66	0.67	0.66
	200	300	160	1031	7.5	0.74	0.86	0.86	0.87	0.66	0.67	0.67
	500	466	162	1031	8.0	0.74	0.88	0.88	0.88	0.65	0.66	0.66
	1000	810	162	1032	9.1	0.74	0.89	0.9	0.9	0.65	0.66	0.66
100	50	113	304	1029	5.0	0.78	0.77	0.77	0.77	0.62	0.62	0.62
	100	200	307	1031	6.0	0.78	0.79	0.79	0.79	0.61	0.62	0.62
	200	313	305	1029	6.6	0.78	0.8	0.8	0.81	0.61	0.62	0.62
	500	532	310	1032	7.4	0.78	0.81	0.81	0.81	0.61	0.62	0.62
	1000	728	309	1032	7.6	0.78	0.81	0.82	0.83	0.61	0.62	0.62
Extraction from GRU Networks — Our Method (Time Limits)												
50	50	132	349	1031	6.3	0.75	0.86	0.86	0.86	0.68	0.69	0.7
	100	210	348	1031	6.9	0.75	0.86	0.86	0.86	0.68	0.69	0.69
	200	352	348	1030	7.6	0.75	0.87	0.87	0.87	0.68	0.69	0.69
	500	540	349	1032	8.8	0.75	0.89	0.89	0.89	0.68	0.7	0.69
	1000	830	353	1034	9.7	0.75	0.89	0.89	0.9	0.68	0.7	0.7
100	50	141	508	1031	5.0	0.79	0.76	0.76	0.77	0.61	0.63	0.62
	100	174	506	1031	5.6	0.79	0.77	0.78	0.78	0.62	0.63	0.63
	200	306	508	1030	6.5	0.79	0.78	0.78	0.79	0.61	0.62	0.63
	500	567	522	1035	7.4	0.79	0.81	0.81	0.82	0.62	0.63	0.63
	1000	780	517	1033	7.5	0.79	0.81	0.81	0.82	0.61	0.62	0.62

The abstraction-based equivalence query provides L* with counterexamples teaching it new ‘parantheses nestings’ one at a time,[37] creating in 128 seconds the Dyck-3 approximation $\mathcal{A}_{24}$ shown in Figu. 4 (the 24th hypothesis created during the extraction). Each of the counterexamples, including those after $\mathcal{A}_{24}$, takes under 3 seconds to find.

[37] The counterexamples are: 1. ( ) 2. { } 3. [} 4. (( )) 5. ({ }) 6. ([ ]) 7. { ( ) } 8. { { } } 9. { [ ] } 10. [{ }] 11. ((( ))) 12. (({ })) 13. ({ [ ] }) 14. ([{ }]) 15. { ([ ]) } 16. { { [ ] } } 17. { [[ ]] } 18. [( ) { }] 19. ([( )]) 20. [(( ))] 21. [[ ] ( )] 22. (([ ])) 23. ([[ ]]) 24. [([ ])] 25. [[( )]] 26. [[{ }]] 27. [[[ ]]] 28. (((( )))) 29. ((([ ]))) 30. (([{ }])) 31. (([[ ])]] 32. (( ) [ ]) Excluding the third counterexample [}, which teaches L*of an incorrectly balanced pair that must be rejected, each counterexample describes a new way to nest parentheses pairs in each other, and is accepted by the RNN. Unfortunately towards the end errors show up, and we see in the 30th counterexample an incorrectly balanced sequence that the RNN accepts.

Table 8 Counterexamples generated during extraction from an LSTM email-address network with 100% train and test accuracy. Examples of the network deviating from its target language are shown in bold

Counter-example	Time (s)	Network Classification	Target Classification
0@m.com	provided	√	√
@@y.net	2.93	×	×
25.net	1.60	√	×
5x.nem	2.34	√	×
0ch.nom	8.01	×	×
9s.not	3.29	×	×
2hs.net	3.56	√	×
@cp.net	4.43	×	×

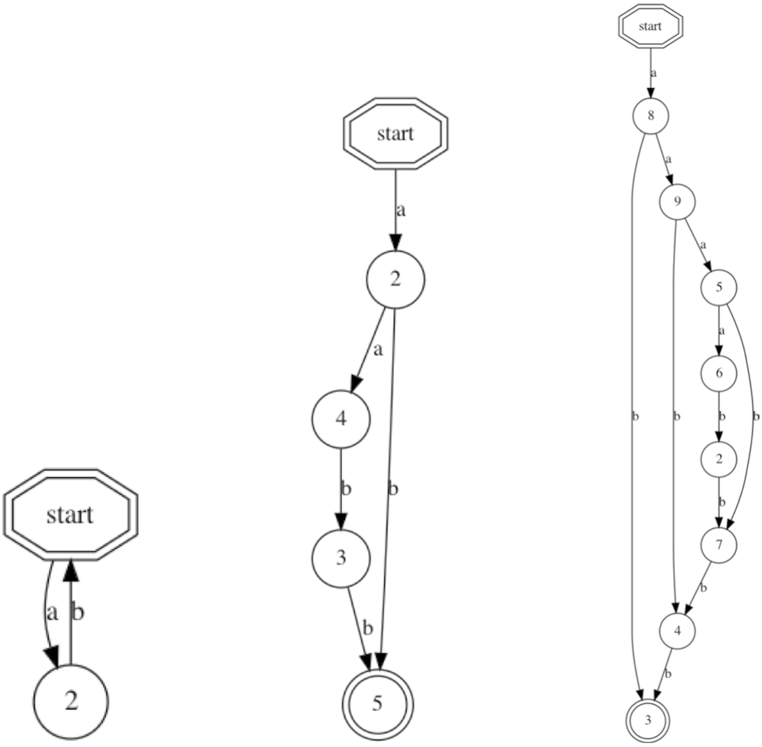

Fig. 3 Automata approximating the language $a^n b^n$ up to different lengths, extracted from an RNN trained on only positive examples. The extraction created 'correct' approximations up to $n = 20$ before reaching the time limit

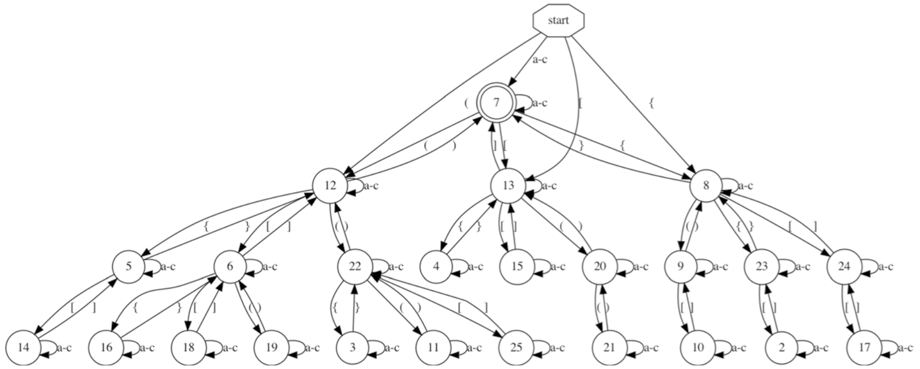

Fig. 4 An automaton approximating the language Dyck-3 with neutral tokens `a-c`, obtained in 128 seconds as the 24th hypothesis during extraction from a GRU trained on only positive samples from the language. The automaton correctly recognises many (but not all) correct parenthesis nestings up to depth $n = 3$, for example, it accepts the sequence `{([])}()` but not the sequence `({()})`. It rejects the empty sequence, this is an artefact of the RNN's behavior

After the counterexample `[([])]` returned for $\mathcal{A}_{24}$ however, L* begins to find irregularities in the LSTM's behavior, and jumps from the 26 state DFA shown in Fig. 4 to the 47 state DFA shown in Fig. 5. The new hypothesis shows us how the GRU has overfitted to the training data. For example, one of the shortest sequences reaching the 'new' accepting state 41 is `[([a]])`, and indeed checking the GRU shows that it accepts this sequence despite it being incorrectly balanced. Following the transitions for this sequence, the GRU's 'first mistake' appears to be on the neutral tokens of state 9, which instead of sitting on a self-loop now go to the different state 22.

Up until $\mathcal{A}_{24}$, the L* refinement time (time from counterexample to next equivalence query) was < 10 seconds per hypothesis. The next refinement, creating $\mathcal{A}_{25}$, takes 68 seconds however, and from there all remaining refinements take $15 - 35$ seconds each.

8.1.3 Sampling the LM-RNN for equivalence queries

Long Samples We take the same Dyck-3 RNN as above and again use L* to extract from it for 400 seconds, but this time with the equivalence query based only on comparison of samples generated from the RNN's distribution. Specifically, for each equivalence query, we sample sequences up to length 100 indefinitely (as the focus here is finding counterexamples, not reaching equivalence quickly) with tokens chosen according to the GRU's next-token distribution.

Sampling the GRU is effective for creating well balanced nested parentheses, and the method rejects the initial hypotheses of L*(in which the parentheses are not yet nested), in under one second. The counterexample has 57 tokens and is:

`{c{}{(b[]){()}c}[]()}({{{}c}ccca}cc[]){b}bbb[]abc[]a[c]()`

which reaches a maximum nesting depth of 4 and shows multiple parentheses nesting combinations. Unfortunately, a second equivalence query is never made before reaching the time limit. The length of the counterexample slows L* down (it has polynomial time

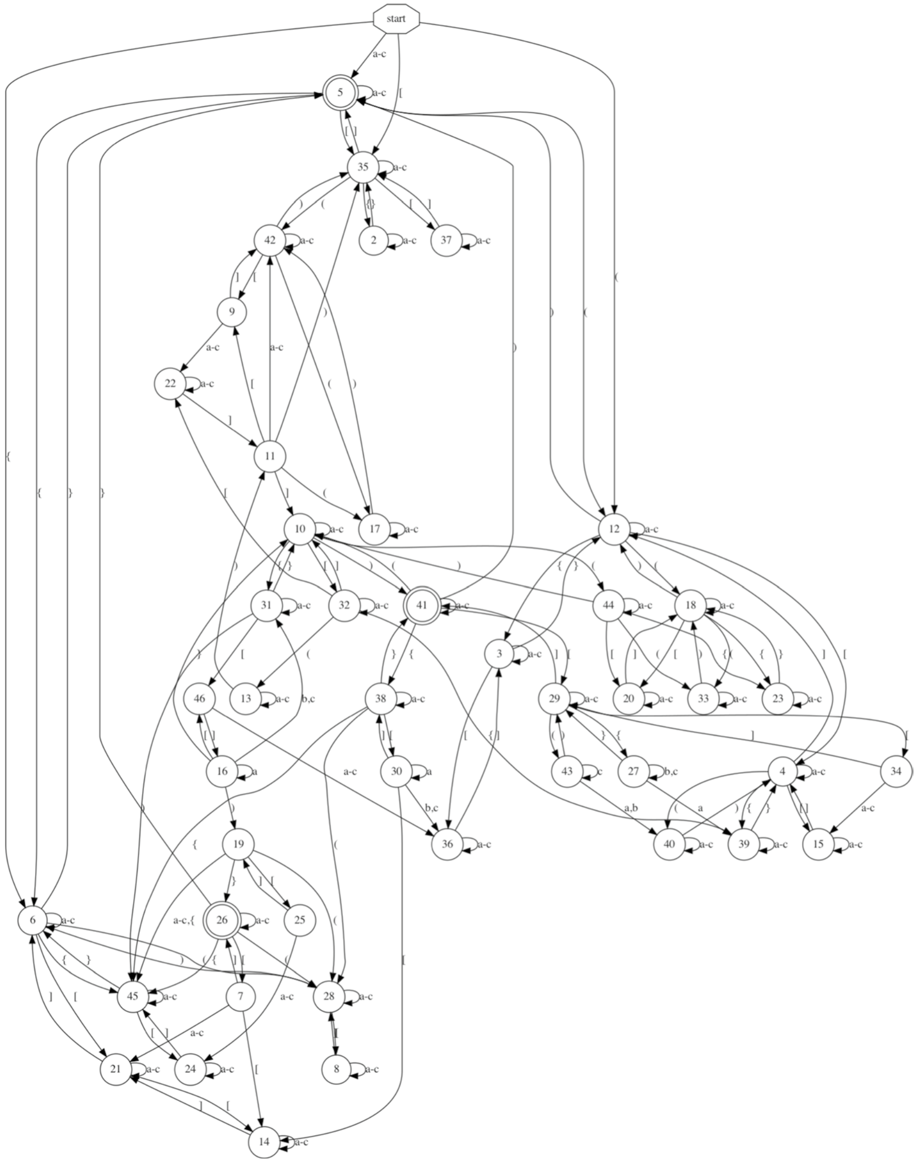

Fig. 5 The next hypothesis presented by L^* after receiving the counterexample `[([])]` to the DFA shown in 4, while extracting from our LM-GRU trained on Dyck-3. While the previous hypotheses reflected clear (regular) subsets of Dyck-3 with bounded depth, now L^* has found several ‘irregularities’ in the RNN, and encoded them into a new hypothesis which is much larger and more complicated than those before it

complexity in, among other things, the length of its counterexamples), and—possibly more significantly—it is possible that this counterexample has led L^* to many ‘incorrect’ behaviours in the RNN, forcing it to begin working on a large DFA covering all of them at a very early stage in the extraction.

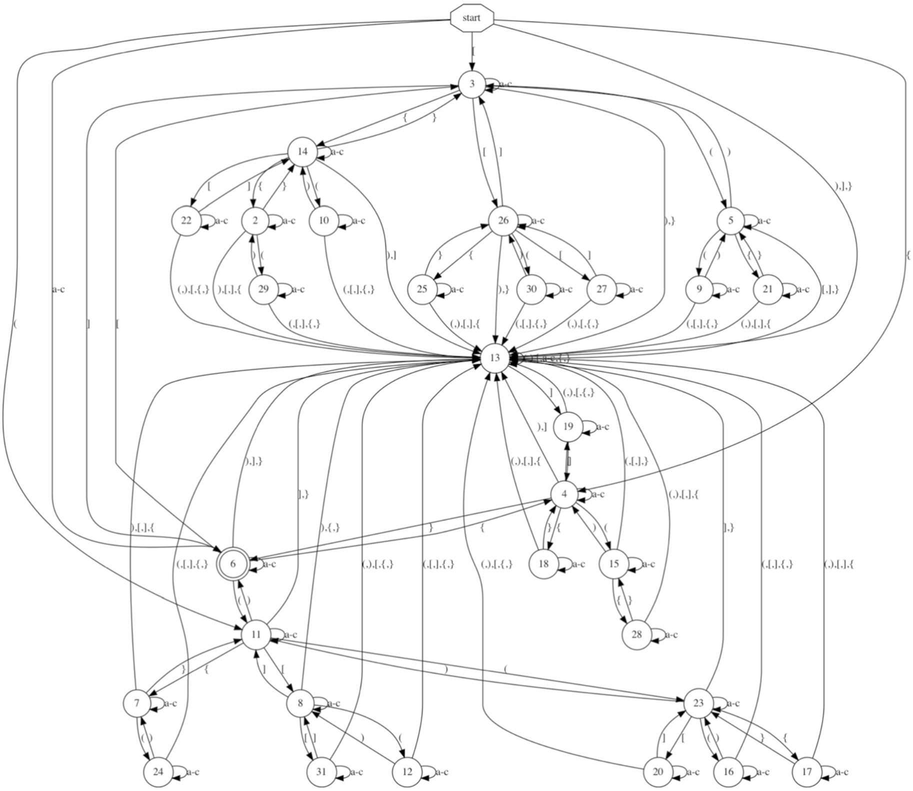

Fig. 6 The last DFA extracted from the LM-GRU trained on Dyck-3 with neutral tokens `a-c`, when extracting with L* for 400 seconds and only using LM-sampling with maximum length 10 for the equivalence queries. It is not a subset of Dyck-3, for example, it accepts the sequence `]]]}`. This seems to be an oversight in the extraction: the RNN does not accept this sequence, and an appropriate counterexample would fix this

LM Sampling: Short Samples A second attempt at extraction with RNN-sampled counterexamples,[38] this time with maximum sample length 10, creates 23 DFAs. The last of these is shown in Fig. 6.

The equivalence queries are fast (the first ten take <1 second each, and all take <6 seconds),[39] though the extraction does not as clearly resemble Dyck-3: the DFAs have irregularities relative to those obtained with the abstraction-based L* extraction method. We do not know whether this is due to the random sampling missing key counterexamples (such as the `[}` counterexample in Sect. 8.1.2) or a reflection of unwanted behaviours in the RNN, but initial checks of misclassified sequences in the last DFA of this extraction show

[38] Again with reject threshold $t = 0.01$ and time limit 400s.

[39] The counterexamples are (examples rejected by the RNN are marked with **R**): 1. **R**: `ab(){c}[(c` 2. `{()}[ac[]]` 3. `[{}]` 4. `a()b([][])` 5. `{a}({})ba` 6. `c{{}c}b(b)` 7. `b[({})b()]` 8. **R**: `b[[][[]{}]` 9. `a([a[]]())` 10. `{()[][]}a` 11. `[acaa{()}]` 12. `b[{{()}}]a` 13. `((()))c{}` 14. **R**: `(c)ca[{[]}` 15. `{[[]]}[]ca` 16. `(({}))` 17. `([()])` 18. `(({}b[])c)` 19. `({}(){()})` 20. `[]{bb({})}` 21. `[(())c]a` 22. `cbb[[()c]]` 23. `a[(a[])]ab` . The last counterexample is given only 5 seconds before the time limit, and so the 24th equivalence query is not reached.

that the RNN actually classifies them correctly, suggesting that at least some key counterexamples could help ‘clean’ these DFAs.

9 Conclusions

We present a novel technique for extracting deterministic finite automata from recurrent neural networks, with roots in exact learning. As our method makes no assumptions as to the internal configuration of the network, it is easily applicable to any RNN architecture, and we evaluate it on the popular LSTM and GRU models. We show also how to apply it to RNNs trained as language models rather than acceptors.

In contrast to previous methods, our method is not affected by hidden state size, and successfully extracts representative DFAs from trained RNNs of arbitrary size—provided of course that the language learned by these RNNs can be approximated by DFAs. Our technique works with little to no parameter tuning, and requires very little prior information to get started (the input alphabet, and optionally 2 labeled samples).

By the nature of L^*, which always returns the minimal automaton consistent with all of its observations, our method is guaranteed to never extract a DFA more complicated than the language of the RNN being considered. Moreover, the counterexamples returned during our extraction can point us to ‘incorrect’ (with respect to the target language) patterns that the network has learned without our awareness.

Beyond scalability and ease of use, our method obtains reasonable approximations for RNNs even if extraction is cut short: for the poorly trained RNNs (RNNs with <80% accuracy on their own test sets) considered in Table 7, our method obtains ≥77% train set accuracy in each of the extractions. Moreover, for networks that accurately represent small automata, we have shown that our method gets very good results: in these cases our method often obtains small, succinct DFAs, with accuracies of over 99% against their networks, in seconds or tens of seconds of extraction (Table 1). This is in contrast to existing methods, which require orders of magnitude more time to complete, and often return cumbersome or inaccurate DFAs (Tables 2 and 3).

Acknowledgements The authors thank Rémi Eyraud, Xiaokun Luan, and the anonymous reviewers for their constructive comments. The research leading to the results presented in this paper is supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 802774 (iEXTRACT).

Author contributions All authors worked on the algorithm design and the paper manuscript. Gail did all the implementations and experiments, under the supervision of Yoav and Eran.

Data availability All experiments were on synthetic data (generated by code below:)

Code availability Code for this paper is available at: https://github.com/tech-srl/lstar_extraction

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Adi, Y., Kermany, E., Belinkov, Y., Lavi, O., & Goldberg, Y. (2016). Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. http://arxiv.org/abs/1608.04207

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information Computer, 75*(2), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6.

Arras, L., Montavon, G., Müller, K., & Samek, W. (2017). Explaining recurrent neural network predictions in sentiment analysis. http://arxiv.org/abs/1706.07206

Ayache, S., Eyraud, R., & Goudian, N. (2018). Explaining black boxes on sequential data using weighted automata. In: Unold O, Dyrka W, Wieczorek W (eds) Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018, PMLR, Proceedings of Machine Learning Research, vol 93, pp 81–103, http://proceedings.mlr.press/v93/ayache19a.html

Balle, B., Carreras, X., Luque, F. M., & Quattoni, A. (2014). Spectral learning of weighted automata - A forward-backward perspective. *Machine Learning, 96*(1–2), 33–63. https://doi.org/10.1007/s10994-013-5416-x.

Barbot, B., Bollig, B., Finkel, A., Haddad, S., Khmelnitsky, I., Leucker, M., Neider, D., Roy, R., & Ye, L. (2021). Extracting context-free grammars from recurrent neural networks using tree-automata learning and a* search. In: Chandlee J, Eyraud R, Heinz J, Jardine A, van Zaanen M (eds) Proceedings of the Fifteenth International Conference on Grammatical Inference, PMLR, Proceedings of Machine Learning Research, vol 153, pp 113–129, https://proceedings.mlr.press/v153/barbot21a.html

Berg, T., Jonsson, B., Leucker, M., & Saksena, M. (2005). Insights to angluin's learning. *Electronic Notes in Theoretical Computational Science, 118,* 3–18. https://doi.org/10.1016/j.entcs.2004.12.015.

Boser, B.E., Guyon, I.M., & Vapnik, V.N. (1992). A training algorithm for optimal margin classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, ACM, New York, NY, USA, COLT '92, pp 144–152, https://doi.org/10.1145/130385.130401, http://doi.acm.org/10.1145/130385.130401

Casey, M. (1998). Correction to proof that recurrent neural networks can robustly recognize only regular languages. *Neural Computation, 10*(5), 1067–1069. https://doi.org/10.1162/089976698300017340

Cechin, A.L., Simon, D.R.P., & Stertz, K. (2003). State automata extraction from recurrent neural nets using k-means and fuzzy clustering. In: Proceedings of the XXIII International Conference of the Chilean Computer Science Society, IEEE Computer Society, Washington, DC, USA, SCCC '03, pp 73–78, http://dl.acm.org/citation.cfm?id=950790.951318

Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. http://arxiv.org/abs/1409.1259

Chung, J., Gülçehre, Ç., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. http://arxiv.org/abs/1412.3555

Clark, A. (2010). Distributional learning of some context-free languages with a minimally adequate teacher. In: Sempere JM, García P (eds) Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings, Springer, Lecture Notes in Computer Science, vol 6339, pp 24–37, https://doi.org/10.1007/978-3-642-15488-1_4

Clark, A., & Eyraud, R. (2007). Polynomial identification in the limit of substitutable context-free languages. J Mach Learn Res 8:1725–1745, http://dl.acm.org/citation.cfm?id=1314556

Clark, A., & Yoshinaka, R. (2016). Distributional Learning of Context-Free and Multiple Context-Free Grammars, Springer Berlin Heidelberg, pp 143–172. https://doi.org/10.1007/978-3-662-48395-4_6

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation, 1*(3), 372–381. https://doi.org/10.1162/neco.1989.1.3.372.

Cohen, M., Caciularu, A., Rejwan, I., & Berant, J. (2017). Inducing Regular Grammars Using Recurrent Neural Networks. http://arxiv.org/abs/1710.10453

Dulizia, A., Ferri, F., & Grifoni, P. (2010). A survey of grammatical inference methods for natural language learning. *Artificial Intelligence Review, 36,* 1–27.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science, 14*(2), 179–211.

Gers, F., & Schmidhuber, E. (2001). Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks, 12*(6), 1333–1340. https://doi.org/10.1109/72.963769

Giles, C. L., Sun, G. Z., Chen, H. H., Lee, Y. C., & Chen, D. (1990). Higher order recurrent networks and grammatical inference. In D. S. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2* (pp. 380–387). Morgan-Kaufmann.

Goldberg, Y. (2016). A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Resoure, 57,* 345–420. https://doi.org/10.1613/jair.4992.

Goldberg, Y. (2017). Neural Network Methods for Natural Language Processing. *Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers,*. https://doi.org/10.2200/S00762ED1V01Y201703HLT037

Goldman, S. A., & Kearns, M. J. (1995). On the complexity of teaching. *Journal of Computational System of Science, 50*(1), 20–31. https://doi.org/10.1006/jcss.1995.1003.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge: The MIT Press.

Gorman, K., & Sproat, R. (2016) Minimally supervised number normalization. Transactions of the Association for Computational Linguistics 4:507–519, https://www.transacl.org/ojs/index.php/tacl/article/view/897

Goudreau, M. W., Giles, C. L., Chakradhar, S. T., & Chen, D. (1994). First-order versus second-order single-layer recurrent neural networks. *IEEE Transactions Neural Networks, 5*(3), 511–513. https://doi.org/10.1109/72.286928.

Hewitt, J., Hahn, M., Ganguli, S., Liang, P., & Manning, C.D. (2020). RNNs can generate bounded hierarchical languages with optimal memory. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Online, pp 1978–2010, 10.18653/v1/2020.emnlp-main.156, https://www.aclweb.org/anthology/2020.emnlp-main.156

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Isberner, M., Howar, F., & Steffen, B. (2014). The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour B, Smolka SA (eds) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, Springer, Lecture Notes in Computer Science, vol 8734, pp 307–322, https://doi.org/10.1007/978-3-319-11164-3_26

Jacobsson, H. (2005). Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation, 17*(6), 1223–1263. https://doi.org/10.1162/0899766053630350.

Kádár, Á., Chrupala, G., & Alishahi, A. (2016). Representation of linguistic form and function in recurrent neural networks. http://arxiv.org/abs/1602.08952

Karpathy, A., Johnson, J., & Li, F. (2015). Visualizing and understanding recurrent networks. http://arxiv.org/abs/1506.02078

Lei, T., Barzilay, R., & Jaakkola, T.S. (2016). Rationalizing neural predictions. http://arxiv.org/abs/1606.04155

Li, J., Chen, X., Hovy, E.H., & Jurafsky. D. (2015). Visualizing and understanding neural models in NLP. http://arxiv.org/abs/1506.01066

Linzen T, Dupoux E, Goldberg Y (2016) Assessing the ability of LSTMs to learn syntax-sensitive dependencies. Transactions of the Association for Computational Linguistics 4:521–535, https://transacl.org/ojs/index.php/tacl/article/view/972

Mayr, F., Yovine, S. (2018). Regular Inference on Artificial Neural Networks. In: Holzinger A, Kieseberg P, Tjoa AM, Weippl E (eds) 2nd International Cross-Domain Conference for Machine Learning and Knowledge Extraction (CD-MAKE), Springer International Publishing, Hamburg, Germany, Machine Learning and Knowledge Extraction, vol LNCS-11015, pp 350–369, 10.1007/978-3-319-99740-7_25, https://hal.inria.fr/hal-02060043, part 5: MAKE Explainable AI

Murdoch, W.J., & Szlam, A. (2017). Automatic rule extraction from long short term memory networks. http://arxiv.org/abs/1702.02540

Okudono, T., Waga, M., Sekiyama, T., & Hasuo, I. (2020). Weighted automata extraction from recurrent neural networks via regression on state spaces. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, AAAI Press, pp 5306–5314, https://aaai.org/ojs/index.php/AAAI/article/view/5977

Omlin, C. W., & Giles, C. L. (1996). Extraction of rules from discrete-time recurrent neural networks. *Neural Networks, 9*(1), 41–52. https://doi.org/10.1016/0893-6080(95)00086-0

Omlin CW, Giles CL (2000) Symbolic knowledge representation in recurrent neural networks: Insights from theoretical models of computation. In: Cloete I, Zurada JM (eds) Knowledge-based Neurocomputing, MIT Press, Cambridge, MA, USA, pp 63–116, http://dl.acm.org/citation.cfm?id=337224.337236

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: An imperative style, high-performance deep learning library. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché Buc F, Fox E, Garnett R (eds) Advances in

Neural Information Processing Systems 32, Curran Associates, Inc., pp 8024–8035, http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.

Sakakibara, Y. (1992). Efficient learning of context-free grammars from positive structural examples. *Information and Computation, 97*(1), 23–60. https://doi.org/10.1016/0890-5401(92)90003-Xhttps://www.sciencedirect.com/science/article/pii/089054019290003X.

Shi, X., Padhi, I., & Knight, K. (2016). Does string-based neural mt learn source syntax? In: EMNLP, pp 1526–1534

Shibata, C., & Yoshinaka, R. (2016). Probabilistic learnability of context-free grammars with basic distributional properties from positive examples. *Theoretical Computer Science, 620*, 46–72. https://doi.org/10.1016/j.tcs.2015.10.037https://www.sciencedirect.com/science/article/pii/S0304397515009433, algorithmic Learning Theory.

Strobelt, H., Gehrmann, S., Huber, B., Pfister, H., Rush, A.M. (2016). Visual analysis of hidden state dynamics in recurrent neural networks. http://arxiv.org/abs/1606.07461

Suzgun, M., Gehrmann, S., Belinkov, Y., & Shieber, S.M. (2019). LSTM networks can perform dynamic counting. http://arxiv.org/abs/1906.03648

Tellier, I. (2006). Learning recursive automata from positive examples. *Revieq d'Intelligence Artificial, 20*(6), 775–804. https://doi.org/10.3166/ria.20.775-804.

Tomita, M. (1982). Dynamic construction of finite automata from examples using hill-climbing. In: Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, Michigan, pp 105–108

Wang, C., & Niepert, M. (2019). State-regularized recurrent neural networks. In: Chaudhuri K, Salakhutdinov R (eds) Proceedings of the 36th International Conference on Machine Learning, PMLR, Proceedings of Machine Learning Research, vol 97, pp 6596–6606, http://proceedings.mlr.press/v97/wang19j.html

Wang, Q., Zhang, K., Ororbia II, A.G., Xing, X., Liu, X., Giles, C.L. (2017). An empirical evaluation of recurrent neural network rule extraction. http://arxiv.org/abs/1709.10380

Wang, Q., Zhang, K., Ororbia II, A.G., Xing, X., Liu, X., & Giles, C.L. (2018). A comparison of rule extraction for different recurrent neural network models and grammatical complexity. http://arxiv.org/abs/1801.05420

Weiss, G., Goldberg, Y., & Yahav, E. (2018a). Extracting automata from recurrent neural networks using queries and counterexamples. In: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, pp 5244–5253, http://proceedings.mlr.press/v80/weiss18a.html

Weiss, G., Goldberg, Y., & Yahav, E. (2018b). On the practical computational power of finite precision RNNs for language recognition. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), Association for Computational Linguistics, Melbourne, Australia, pp 740–745, https://doi.org/10.18653/v1/P18-2117, https://www.aclweb.org/anthology/P18-2117

Weiss, G., Goldberg, Y., & Yahav, E. (2019). Learning deterministic weighted automata with queries and counterexamples. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox E, Garnett R (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 32

Yellin, D. M., & Weiss, G. (2021). Synthesizing context-free grammars from recurrent neural networks. In J. F. Groote & K. G. Larsen (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 351–369). Cham: Springer International Publishing.

Yokomori, T. (2003). Polynomial-time identification of very simple grammars from positive data. *Theoretical Computational Science, 298*(1), 179–206. https://doi.org/10.1016/S0304-3975(02)00423-1.

Yoshinaka, R. (2019). Distributional learning of conjunctive grammars and contextual binary feature grammars. *Journal of Computation Systematic of Science, 104,* 359–374. https://doi.org/10.1016/j.jcss.2017.07.004.

Zeng, Z., Goodman, R. M., & Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation, 5*(6), 976–990. https://doi.org/10.1162/neco.1993.5.6.976

Zhang, X., Du, X., Xie, X., Ma, L., Liu, Y., & Sun, M. (2021). Decision-guided weighted automata extraction from recurrent neural networks. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual

Event, February 2-9, 2021, AAAI Press, pp 11699–11707, https://ojs.aaai.org/index.php/AAAI/article/view/17391