Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples

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

Abstract

We present a novel algorithm that uses exact learning and abstraction to extract a deterministic finite automaton describing the state dynamics of a given trained RNN. We do this using Angluin's L* algorithm as a learner and the trained RNN as an oracle. Our technique efficiently extracts accurate automata from trained RNNs, even when the state vectors are large and require fine differentiation.

1. Introduction

Recurrent Neural Networks (RNNs) are a class of neural networks used to process sequences of arbitrary lengths. An RNN receives an input sequence timestep by timestep, returning a new *state vector* after each step. For classification tasks, this is followed by passing the state vectors to a multi-class classification component, which is trained alongside the RNN and returns a classification for the sequence. We call a combination of an RNN and a binary classification component an *RNN-acceptor*.

RNNs are central to deep learning, and natural language processing in particular. However, while they have been shown to reasonably approximate a variety of languages, what they eventually learn is unclear. Indeed, several lines of work attempt to extract clear rules for their decisions (Jacobsson, 2005; Omlin & Giles, 1996; Cechin et al., 2003).

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.)

We approach this task using *exact learning*.

[1]Technion, Haifa, Israel [2]Bar Ilan University, Ramat Gan, Israel. Correspondence to: Gail Weiss <sgailw@cs.technion.ac.il>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

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*: label a given instance
- *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 for some regular language L. In this context, the concept is L, and the instances are words over its alphabet.

We designate a trained RNN[1] as teacher for the L* algorithm, in order to extract a DFA representing its behavior. The RNN is used trivially to answer membership queries: input sequences are fed to the network for classification. The main challenge in this setting is answering equivalence queries.

Problem Definition: Equivalence Query Given an RNN-acceptor R trained over a finite alphabet Σ, and a DFA $\mathcal{A}$ over Σ, determine whether R and $\mathcal{A}$ are equivalent, and return a counterexample $w \in \Sigma^*$ if not.

As this problem is likely to be intractable, we use an approximation. One approach would be random sampling; however, should R and $\mathcal{A}$ be similar, this may take time.

Our Approach We use *finite abstraction* of the RNN R to answer equivalence queries. The finite abstraction and the L* DFA $\mathcal{A}$ act as two hypotheses for the RNN ground truth, and must at least be equivalent to each other in order to be equivalent to R. 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.

Our approach is guaranteed never to return an incorrect counterexample nor invoke an unnecessary refinement; i.e., *it yields no false negatives*. As far as we know, this is the first attempt to apply exact learning to a given RNN.

[1]In what follows, when understood from context, we use the term RNN to mean RNN-acceptor.

Main Contributions

- We present a novel and general framework for extracting automata from trained RNNs. We use the RNNs as teachers in an exact learning setting.
- We implement the technique and show its ability to extract descriptive automata in settings where previous approaches fail. We demonstrate its effectiveness on modern RNN architectures.
- 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 generalized 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.

2. Related Work

DFA extraction from RNNs was extensively explored by Giles and colleagues; see Wang et al. (2017) and Jacobsson (2005) for a partial survey.

Broadly, the approaches work by defining a finite partitioning of the real-valued RNN state space and then exploring the network transitions in the partitioned space, using techniques such as BFS exploration (Omlin & Giles, 1996) and other transition-sampling approaches. The approaches differ mainly in their choice and definition of partitioning.

These works generally use second order RNNs (Giles et al., 1990), which are shown to better map DFAs than first-order Elman RNNs (Elman, 1990; Goudreau et al., 1994; Wang et al., 2018). In this work, however, we will focus on GRUs (Cho et al., 2014; Chung et al., 2014) and LSTMs (Hochreiter & Schmidhuber, 1997), as they are more widely used in practice.

One approach to state space partitioning is to divide each dimension into q equal intervals, with q being the *quantization level* (Omlin & Giles, 1996). This approach suffers from inherent state space 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).

Another approach is to fit an unsupervised classifier such as k-means to a large sample set of reachable network states (Cechin et al., 2003; Zeng et al., 1993). The number of clusters k generated with these classifiers is a parameter that might greatly affect extraction results, especially if it is too small. The sample states can be found by a simple 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).

An inherent weakness of both these approaches is that the partitioning is set before the extraction begins, with no mechanism for recognizing and overcoming overly coarse behavior. Both methods thus face the challenge of choosing the best parameter value for extraction. They are generally applied several times with different parameter values, after which the 'best' DFA is chosen according to a heuristic.

Current techniques treat all the dimensions of an RNN as a single state. In future work, it may be interesting to make the distinction between 'fast' and 'slow' internal states as introduced in the differential state framework unifying GRUs and LSTMs (Ororbia II et al., 2017).

3. Background

Recurrent Neural Networks and RNN acceptors An RNN is a parameterized 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. To use a set of discrete symbols as an input alphabet, each symbol is deterministically mapped to an input vector using either a one-hot encoding or an embedding matrix. As we are only interested in the internal network transitions, we use one-hot encoding in this work. For convenience, we refer to input symbols and their corresponding input vectors interchangeably. We denote the state space of a network R by $S_R = \mathbb{R}^{d_s}$. For multi-layered RNNs, where several layers each have their own state vector, we consider the concatenation of these vectors as the state vector of the entire network. In a binary *RNN-acceptor*, there is an additional function $f_R : S_R \to \{Acc, Rej\}$ that classifies the RNN's state vectors. An RNN-acceptor R is defined by the pair of functions g_R, f_R.

Network Abstraction Given a neural network R with state space S and alphabet Σ, and a partitioning function $p\colon S \to \mathbb{N}$, 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. The method is effectively a BFS exploration of the partitions defined by p, beginning with $p(h_0)$, where h_0 is the network's initial state, and continuing according to the network's transition function g_R.

We denote by $A^{R,p}$ the DFA extracted by this method from a network R and partitioning p, and denote all its related sets and functions by subscript R, p.

The L* Algorithm The L* algorithm is an exact learning algorithm for extracting a DFA from any teacher that can answer *membership queries* (label a given word) and *equivalence queries* (accept or reject a given DFA, with a counterexample if rejecting). We know that L* always proposes a minimal DFA in equivalence queries and utilize this in our work. Beyond this, we treat the algorithm as a black box. A short review is provided in the supplementary material.

4. Learning Automata from RNNs using L*

We build an RNN-based teacher for L* as follows:

For membership queries, we use the RNN classifier directly, checking whether it accepts or rejects the given word.

For equivalence queries: Given a proposed DFA $\mathcal{A}$, we compare it to abstractions $A^{R,p}$ of the network R, beginning with some initial partitioning p of S_R. If we find a disagreement between $\mathcal{A}$ and an abstraction $A^{R,p}$, we use R to determine whether to return it as a counterexample or to refine p and restart the comparison.

In theory this continues until $\mathcal{A}$ and $A^{R,p}$ converge, i.e., are equivalent. 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, after which the last L* DFA, $\mathcal{A}$, is returned. We see that these DFAs still generalize well to their respective networks.[2]

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. Notations

Automaton and Classification Function For a deterministic automaton $A = \langle \Sigma, Q, q_0, F, \delta \rangle$, Σ is its alphabet, Q the set of automaton states, $F \subseteq Q$ the set of accepting states, $q_0 \in Q$ the initial state, and $\delta : Q \times \Sigma \to Q$ its transition function. We denote by $\hat{\delta} : Q \times \Sigma^* \to Q$ the recursive application of δ to a sequence, i.e., for every $q \in Q$, $\hat{\delta}(q, \varepsilon) = q$, and for every $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\hat{\delta}(q, w \cdot \sigma) = \delta(\hat{\delta}(q, w), \sigma)$. For convenience, we add the notation $f : Q \to \{Acc, Rej\}$ as the function giving the classification of each state, i.e., $f(q) = Acc \iff q \in F$.

Binary RNN-acceptor For a binary RNN-acceptor, we denote by $h_{0,R}$ the initial state of the network, and by $\hat{g_R} : S_R \times \Sigma^* \to S_R$ the recursive application of g_R to a sequence, i.e., for every $h \in S_R$, $\hat{g_R}(h, \varepsilon) = h$, and for every $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\hat{g_R}(h, w \cdot \sigma) = g_R(\hat{g_R}(h, w), \sigma)$. We drop the subscript R when it is clear from context.

We note that a given RNN-acceptor can be interpreted as a deterministic, though possibly infinite, state machine.

[2]We could also return the last abstraction, $A^{R,p}$, and focus on refining p over returning counterexamples. But the abstractions are often less accurate. We suspect this is due to the lack of 'foresight' $A^{R,p}$ has in comparison to L*'s many separating suffix strings.

Shorthand As an abuse of notation, for any DFA or RNN classifier C with state transition function t_C, state classification function f_C, and initial state $q_{C,0}$, we use $\hat{t_C}(w)$ to denote $\hat{t_C}(q_{C,0}, w)$, $f_C(q, w)$ to denote $f_C(\hat{t_C}(q, w))$, and $f_C(w)$ to denote $f_C(\hat{t_C}(q_{C,0}, w))$. Within this notation, the classifications of a word $w \in \Sigma^*$ by an automaton A and a binary RNN-acceptor R with respective classification functions f_A and f_R are given by $f_A(w)$ and $f_R(w)$.

6. Answering Equivalence Queries

6.1. Overview

Given a network R, a partitioning function $p : S \to \mathbb{N}$ over its state space S, and a proposed minimal automaton $\mathcal{A}$, we wish to check whether R is equivalent to $\mathcal{A}$, preferably exploring as little of R's behavior as necessary to respond.

We search for a disagreeing example w between $\mathcal{A}$ and the abstraction $A^{R,p}$, by parallel traversal of the two. If one is found, we check its true classification in R. If this disagrees with $\mathcal{A}$, w is returned as a counterexample; otherwise, p is refined (Section 7) and the traversal begins again.[3]

Every counterexample w returned by our method is inherently true, i.e., satisfies $f_{\mathcal{A}}(w) \neq f_R(w)$. From this and the minimality of L* equivalence queries, we obtain:

Property 1 *Every separate state in the final extracted automaton $\mathcal{A}$ is justified by concrete input to the network.*

In other words, all complexity in a DFA extracted from a given RNN R is a result of the inherent complexity of R. This is in contrast to other methods, in which incorrect partitioning of the network state space may lead to unnecessary complexity in the extracted DFA, even after minimization. Moreover, our method refines the partitioning only when it is proven too coarse to correctly represent the network:

Property 2 *Every refinement to the partitioning function $p : S \to \mathbb{N}$ is justified by concrete input to the network.*

This is important, as the search for counterexamples runs atop an extraction of the abstraction $A^{R,p}$, and so unnecessary refinements—which may lead to state space explosion—can make the search so slow as to be impractical.

For clarity, we henceforth 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 as L-states.

6.2. Parallel Exploration

The key intuition to our approach is that $\mathcal{A}$ is minimal, and so each A-state should—if the two DFAs are equivalent—be

[3]If the refinement does not affect any states traversed so far, this is equivalent to fixing the current state's abstraction and continuing.

equivalent to exactly one L-state, w.r.t. classification and projection of transition functions. The extraction of $A^{R,p}$ is effectively a BFS traversal of $A^{R,p}$, allowing us to associate between states in the two DFAs during its extraction.

We refer to bad associations, in which an accepting A-state is associated with a rejecting L-state or vice versa, as *abstract classification conflicts*, and to multiple but disagreeing associations, in which one A-state is associated with two different L-states, as *clustering conflicts*. (The inverse case, in which one L-state is associated with several A-states, is not necessarily a problem, as $A^{R,p}$ is not necessarily minimal.)

We may also assert that the classification of each R-state h encountered while extracting $A^{R,p}$ is identical to that of the L-state $q_{\mathcal{A}} \in Q_{\mathcal{A}}$ that the parallel traversal of $\mathcal{A}$ reaches during the exploration. As the classification of an A-state is determined by the R-state with which it was first reached, this also covers all abstract classification conflicts. We refer to failures of this assertion as *classification conflicts*, and check only for them and for clustering conflicts.

6.3. Conflict Resolution and Counterexample Generation

We assume an initial partitioning $p : S \to \mathbb{N}$ of the R-state space and a refinement operation $ref : p, h, H \mapsto p'$ which receives a partitioning p, an R-state h, and a set of R-states $H \subseteq S \setminus \{h\}$, and returns a new partitioning p' satisfying:

1. $\forall h_1 \in H, p'(h) \neq p'(h_1)$, and
2. $\forall h_1, h_2 \in S, p(h_1) \neq p(h_2) \Rightarrow p'(h_1) \neq p'(h_2)$.

(In practice, condition 1 may be relaxed to separating at least one of the vectors in H from h, and our method can and has overcome imperfect splits.)

Classification conflicts occur when some $w \in \Sigma^*$ for which $f_R(w) \neq f_{\mathcal{A}}(w)$ has been traversed during $A^{R,p}$'s extraction. We resolve them by returning w as a counterexample.

Clustering conflicts occur when the parallel exploration associates an A-state $q \in Q_{R,p}$ with an L-state q_2, after q has already been associated with an L-state $q_1 \neq q_2$. As $\mathcal{A}$ is minimal, q_1 and q_2 cannot be equivalent. It follows that if $w_1, w_2 \in \Sigma^*$ are the BFS traversal paths through which q was associated with $q_1, q_2 \in Q_{\mathcal{A}}$, then there exists some differentiating sequence $s \in \Sigma^*$ for which $f_{\mathcal{A}}(q_1, s) \neq f_{\mathcal{A}}(q_2, s)$, i.e., for which $f_{\mathcal{A}}(w_1 \cdot s) \neq f_{\mathcal{A}}(w_2 \cdot s)$. Conversely, the arrival of w_1 and w_2 at the same A-state $q \in A^{R,p}$ gives $f_{R,p}(w_1 \cdot s) = f_{R,p}(q, s) = f_{R,p}(w_2 \cdot s)$.

It follows that $\mathcal{A}$ and $A^{R,p}$ disagree on the classification of either $w_1 \cdot s$ or $w_2 \cdot s$, and so necessarily at least one is not equivalent to R. We pass $w_1 \cdot s$ and $w_2 \cdot s$ through R for their true classifications. If $\mathcal{A}$ is at fault, the sequence on which $\mathcal{A}$ and R disagree is returned as a counterexample. Otherwise, necessarily, $f_R(w_1 \cdot s) \neq f_R(w_2 \cdot s)$, and so $A^{R,p}$ should satisfy $\hat{\delta_{R,p}}(w_1) \neq \hat{\delta_{R,p}}(w_2)$. The R-states $h_1 = \hat{g}(w_1)$ and $h_2 = \hat{g}(w_2)$ are passed, along with p, to ref, to yield a new, finer, partitioning p' for which $\hat{\delta_{R,p'}}(w_1) \neq \hat{\delta_{R,p'}}(w_2)$.

This reasoning applies to w_2 with *all* paths w' that have reached q without conflict before w_2. As such, the classifications of *all* words $w' \cdot s$ are tested against R, prioritizing returning a counterexample over refining p. If a refinement is triggered, then $h = \hat{g}(w_2)$ is split from the set of R-states $h' = \hat{g}(w')$.

Algorithm 1 shows pseudocode for this equivalence checking. In it, all mappings except one are unique and defined before they are accessed. The exception is `Paths`, as we might reach the same R-state $h \in S_R$ more than once, by different paths. This can be remedied by maintaining in `Paths` not single paths but lists of paths.

Our experiments showed that long counterexamples often caused $\mathcal{A}$ to blow up, without generalizing well. Thus, we always return the shortest available counterexample.

7. Abstraction and Refinement

Given a partitioning p, an R-state h, and a set of R-states $H \subseteq S \setminus \{h\}$, we refine p in accordance with the requirements described in Section 6.3. We want to generalize the information given by h and H well, so as not to invoke excessive refinements. We also need an initial partitioning p_0 from which to start.

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.

7.1. Initial Partitioning

As we wish to keep the abstraction as small as possible, we begin with no state separation at all: $p_0 : h \mapsto 0$.

7.2. Support-Vector based Refinement

In this section we assume $p(h') = p(h)$ for every $h' \in H$, which is true for our case. The method generalizes trivially to cases where this is not true.[4]

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 neighboring R-states that do not. We achieve this by fitting an SVM (Boser et al., 1992) classifier with an RBF kernel[5] to separate h from H. The

[4]By removing from H any vectors h' for which $p(h') \neq p(h)$.

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

Algorithm 1 Pseudo-code for equivalence checking of an RNN R and minimal DFA $\mathcal{A}$, with initial partitioning p_0.

```
method update_records($q, h, q_{\mathcal{A}}, w$):
Visitors($q$) $\leftarrow$ Visitors($q$) $\cup \{h\}$,
Paths($h$) $\leftarrow w$
Association($q$) $\leftarrow (q_{\mathcal{A}})$
Push(New,$\{h\}$)
end method

method handle_cluster_conf($q, q_{\mathcal{A}}, q'_{\mathcal{A}}$):
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
    $w \leftarrow Paths(h) \cdot s$
    if $f_R(w) \neq f_{\mathcal{A}}(w)$ then
        return Reject, $w$
end for
$p \leftarrow ref(p, h'$,Visitors$(q') \setminus \{h'\})$
return Restart_Exploration
end method

method parallel_explore($R, \mathcal{A}, p$):
empty all of: $Q, F, \delta$, New, Visitors, Paths, Association
$q_0 \leftarrow p(h_0)$
update_records($q_0, h_0, q_{\mathcal{A},0}, \varepsilon$)
while New $\neq \emptyset$ do
    $h \leftarrow$ Pop(New)
    $q \leftarrow p(h)$
    $q_{\mathcal{A}} \leftarrow$ Association($q$)
    if $f_R(h) \neq f_{\mathcal{A}}(q_{\mathcal{A}})$ then
        return Reject, (Paths($h$))
    if $q \in Q$ then continue
    $Q \leftarrow Q \cup \{q\}$
    if $f_R(h) = Acc$ then $F \leftarrow F \cup \{q\}$
    for $\sigma \in \Sigma$ do
        $h' \leftarrow g_R(h, \sigma)$
        $q' \leftarrow p(h')$
        $\delta(q, \sigma) \leftarrow q'$
        if $q' \in Q$ and Association($q'$) $\neq \delta_{\mathcal{A}}(q_{\mathcal{A}}, \sigma)$ then
            return handle_cluster_conf($q, q_{\mathcal{A}}, \delta_{\mathcal{A}}(q_{\mathcal{A}}, \sigma)$)
        update_records($q', h', \delta_{\mathcal{A}}(q_{\mathcal{A}}, \sigma)$,Paths($h$) $\cdot \sigma$)
    end for
end while
return Accept
end method

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

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. We track the refinements by arranging the obtained SVMs in a decision tree, where each node's decision is the corresponding SVM, and the leaves represent the current A-states.

Barring failure of the SVM, this approach satisfies the requirements of refinement operations, and avoids state explosion by adding only one A-state per refinement. Otherwise, the method fails to satisfy requirement 1. Nevertheless, at least one of the R-states $h' \in H$ is separated from h, and later explorations can invoke further refinements if necessary. In practice this does not hinder the goal of the abstraction: finding counterexamples to equivalence queries.

The 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. However, as this number of A-states also grows very slowly (linearly in the number of refinements), this does not become a problem.

7.3. Practical Considerations

As the initial partitioning and the refinement operation are very coarse, the method may accept very small but wrong DFAs. To counter this, two measures are taken:

1. One accepting and one rejecting sequence are provided to the teacher as potential counterexamples to be considered at every equivalence query.
2. The first refinement uses an aggressive approach that generates a great (but manageable) number of A-states.

The first measure, necessary to prevent termination on a single state automaton, requires only two samples. These can be found by random sampling, or taken from the training set.[6] In keeping with the observation made in Section 6.3, we take the shortest available samples. The second measure prevents the extraction from too readily terminating on small DFAs. Our method for it is presented in Section 7.3.1.

7.3.1. Aggressive Difference-based Refinement

We split h from at least one of the vectors $h' \in H$ by splitting S along the d dimensions with the largest gap between h and the mean h_m of H, down the middle of that gap. This refinement can be comfortably maintained in a decision tree, generating at the split point a tree of depth d for which, on each layer $i = 1, 2, ..., d$, each node is split along the dimension with the i-th largest gap. This refinement follows intuitively from the quantization suggested by Omlin and

in this work.

[6]If no such samples exist, a single state DFA may be correct.

Giles, but 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 values of around 7-10 generally provide a strong enough initial partitioning of S, without making the abstraction too large for feasible exploration.

8. Experimental Results

We demonstrate the effectiveness of our method on networks trained on the Tomita grammars (1982),[7] used as benchmarks in previous automata-extraction work (Wang et al., 2017), and on substantially more complicated languages. We show the effectiveness of our equivalence query approach over simple random sampling and present cases in which our method extracts informative DFAs whereas other approaches fail. In addition, for some seemingly perfect networks, we find that our method quickly returns counterexamples representing deviations from the target language.

On all networks, we applied our method with initial refinement depth 10. Unlike other extraction methods, where parameters must be tuned to find the best DFA, no parameter tuning was required to achieve our results.

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.[8]

Prototype Implementation and Settings We implemented all methods in Python, using PyTorch (Paszke et al., 2017) and scikit-learn (Pedregosa et al., 2011). For the SVM classifiers, we used the SVC variant, with regularization factor $C = 10^4$ to encourage perfect splits and otherwise default parameters—in particular, the RBF kernel with gamma value $1/(\text{num features})$.

Training Setup As our focus was extraction, we trained all networks to 100% accuracy on their train sets, and of these we considered only those that reached $99.9+\%$ accuracy on a dev 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 dev sets were not controlled.

The train sets contained samples of various lengths, with a 1:1 ratio between the positive and negative samples from each length where possible. To achieve this, a large number of words were uniformly sampled for each length. When not enough samples of one class were found, we limited the ratio to 50:1, or took at most 50 samples if all were classified identically. The train set sizes, and the lengths of the samples in them, are listed for every language in this paper in the supplementary material.

Table 1. Accuracy of DFAs extracted from GRU networks representing small regular languages. Single values represent the average of 3 experiments, multiple values list the result for each experiment. Extraction time of 30 seconds is a timeout.

Hidden		DFA	Average Accuracy on Length				
Size	Time (s)	Size	10	50	100	1000	Train
50	30, 30, 30	11,11,155	99.9	99.8	99.9	99.9	99.9
100	11.0	11,10,11	100	99.9	99.9	99.9	100
500	30, 30, 30	10,10,10	100	99.9	100	99.9	100.0

For languages where the positive class was unlikely to be found by random sampling—e.g. balanced parentheses or emails—we generated positive samples using tailored functions.[9] In these cases we also generated negative samples by mutating the positive examples.[10] Wherever a test set is mentioned, it was taken as a 1:1 sample set from the same distribution generating the positive and negative samples.

Effectiveness on Random Regular Languages We first evaluated our method on the 7 Tomita grammars. We trained one 2-layer GRU network with hidden size 100 for each grammar (7 RNNs in total). All but one RNN reached 100% dev accuracy; the one trained on the 6th Tomita grammar reached 99.94%. For each RNN, our method correctly extracted and accepted the target grammar in under 2 seconds.

The largest Tomita grammars have 5-state DFAs over a 2-letter alphabet. We also explored substantially more complex grammars: we trained 2-layer GRU networks with varying hidden-state sizes on 10-state minimal DFAs generated randomly over a 3-letter alphabet. We applied our method to these networks with a 30 second time limit—though most reached equivalence sooner. Extracted DFAs were compared against their networks on their train sets and on 1000 random samples for each of several word-lengths.

Table 1 shows the results. Each row represents 3 experiments: 9 random DFAs were generated, trained on, and extracted. The extracted DFAs are small, and highly accurate even on long sequences (length 1000). Additional results showing similar trends, including experiments on LSTM networks, are available in the supplementary material.

[7]The Tomita grammars are the following 7 languages over the alphabet $\{0, 1\}$: [1] 1^*, [2] $(10)^*$, [3] the complement of $((0|1)^*0)^*1(11)^*(0(0|1)^*1)^*0(00)^*(1(0|1)^*)^*$, [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] $0^*1^*0^*1^*$.

[8]Measured using `clock()`, of Python's `time` module, and covering among others: abstraction exploration, abstraction refinements (including training SVM classifiers), and L* refinements.

[9]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 `@`.

[10]By adding, removing, changing, or moving up to 9 time characters.

Comparison with a-priori Quantization In their 1996 paper, Omlin and Giles suggest partitioning the network state space by dividing each state dimension into q equal intervals, with q being the *quantization level*. We tested this method on each of our networks, with $q = 2$ and a time limit of 1000 seconds to avoid excessive memory consumption.

In contrast to our method, which extracted on these same networks small and accurate DFAs within 30 seconds, we found that for this method this was not enough time to extract a complete DFA. The extracted DFAs were also very large—often with over 60,000 states—and their coverage of sequences of length 1000 tended to zero. For the covered sequences however, the extracted DFA's accuracy was often very high (99+%), suggesting that quantization—while impractical—is sufficiently expressive to describe a network's state space. However, it is also possible that the sheer size of the quantization (2^{50} for our smallest RNNs) simply allowed each explored R-state its own A-state, giving high accuracy by observation bias.

This highlights the key strength of our method: in contrast to other methods, our method is able to find small and accurate DFAs representing a given RNN, when such DFAs are available. It does this in a fraction of the time required by other methods to complete their extractions. This is because, unlike other methods, it maintains from a very early point in extraction a complete DFA that constitutes a continuously improving approximation of R.

Comparison with Random Sampling For Counterexample Generation We show that there is merit to our approach to equivalence queries over simple random sampling.

Networks R for which the ratio between accepting and rejecting sequences is very uneven may be closely approximated by simple DFAs—making it hard to differentiate between them and their L* proposed automata by random sampling. We trained two networks on one such language: balanced parentheses (BP) over the 28-letter alphabet {`a`,`b`,...,`z`,`(`,`)`} (the language of all sequences w over `a-z()` in which every opening parenthesis is eventually followed by a single corresponding closing parenthesis, and vice versa). The networks were trained to 100% accuracy on train sets of size $\sim$44600, containing samples with balanced parentheses up to depth 11. The two train sets had 36% and 43% negative samples, which were created by slightly mutating the positive samples. The networks were a 2-layer GRU and a 2-layer LSTM, both with hidden size 50 per cell.

We extracted from these networks using L*, approaching equivalence queries either with our method or by random sampling. We implemented the random sampling teacher to sample up to 1000 words of each length in increasing order. For fairness, we also provided it with the same two initial samples our teacher was given, allowing it to check and possibly return them at every equivalence query.

Table 2. Accuracy and maximum nesting depth of extracted automata for networks trained on BP, using either abstractions ("Abstr") or random sampling ("RS") for equivalence queries. Accuracy is measured with respect to the trained RNN.

	Train Set Accuracy		Max Nest. Depth	
Network	Abstr	RS	Abstr	RS
GRU	99.98	87.12	8	2
LSTM	99.98	94.19	8	3

Table 3. Counterexamples generated during extraction of automata from a GRU network trained on BP.

Refinement Based		Brute Force	
example	Time (s)	example	Time (s)
))	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		

We ran each extraction with a time limit of 400 seconds and found a nice pattern: every DFA proposed by L* represented BP to some bounded nesting depth, and every counterexample taught it to increase that depth by 1.

The accuracy of the extracted DFAs on the network train sets is shown in Table 2, along with the maximum depth the L* DFAs reached while still mimicking BP. For the GRU extractions, the counterexamples and their generation times are listed in Table 3. Note the speed and succinctness of those generated by our method as opposed to those generated by random sampling.

Adversarial Inputs Excitingly, the penultimate counterexample returned by our method is an adversarial input: a sequence with unbalanced parentheses that the network (incorrectly) accepts. This input is found in spite of the network's seemingly perfect behavior on its 44000+ sample train set. We stress 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 (the automata as well as the counterexamples are available in the supplementary material). The RNN overfit to random peculiarities in the training data and did not learn the intended language.

k-Means Clustering We also implemented a simple k-means clustering and extraction approach and applied it to the BP networks with a variety of k values, allowing it to divide the state space into up to 100 clusters based on the states observed with the networks' train sets. This failed to learn any BP to any depth for either network: for both networks, it only managed to extract DFAs almost resembling BP to nesting depth 3 (accepting also some unbalanced sequences).

Limitations Due to L*'s polynomial complexity and intolerance to noise, for networks with complicated behavior, extraction becomes extremely slow and returns large DFAs. Whenever applied to an RNN that has failed to generalize properly to its target language, our method soon finds several adversarial inputs, builds a large DFA, and times out while refining it.[11]

This does however demonstrate the ease with which the method identifies incorrectly 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.

For instance, for a seemingly perfect LSTM network trained on the regular expression

[a-z][a-z0-9]*@[a-z0-9]+.(com|net|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 4, 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,[12] 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.

We note that our implementation of kmeans 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 50/50 ratio between positive and negative samples.

Beyond demonstrating the capabilities of our method, these results also highlight the brittleness in generalization of trained RNNs, and suggest that evidence based on test-set performance should be interpreted with extreme caution.

[11]This happened also to our BP LSTM network, which timed out during L* refinement after the last counterexample.

[12]A 134-state DFA $\mathcal{A}$ was proposed by L* after 178 seconds, and the next refinement to $\mathcal{A}$ (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.

Table 4. Counterexamples generated during extraction from an LSTM email 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	×	×

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 normalization. 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.

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, including the popular LSTM and GRU models.

In contrast to previous methods, our method is not affected by hidden state-size, and successfully extracts representative DFAs for any networks that can indeed be represented as such. Unlike other extraction approaches, our technique works with little to no parameter tuning, and requires very little prior information to get started (the input alphabet, and 2 labeled examples).

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 patterns the network has learned without our awareness.

Beyond scalability and ease of use, our method can return reasonably accurate DFAs even if extraction is cut short. Moreover, we have shown that for networks that do correspond to succinct automata, our method gets very good results—generally extracting small, succinct DFAs with accuracies of over 99% with respect to their networks, in seconds or tens of seconds. This is in contrast to existing methods, which require orders of magnitude more time to complete, and often return large and cumbersome DFAs (with tens of thousands of states).

Supplementary Material

This supplementary material contains a description of the L* algorithm (Appendix A), and additional experimental results and details (Appendix B).

A. Angluin's L* Algorithm

Algorithm 2 L* Algorithm with explicit membership and equivalence queries.

```
$S \leftarrow \{\epsilon\}, E \leftarrow \{\epsilon\}$
**for** $(s \in S)$, $(a \in \Sigma)$, and $(e \in E)$ **do**
  $T[s, e] \leftarrow$ **Member**$(s \cdot e)$
  $T[s \cdot a, e] \leftarrow$ **Member**$(s \cdot a \cdot e)$
**end for**
**while** True **do**
  **while** $(s_{new} \leftarrow Closed(S, E, T) \neq \perp)$ **do**
    $Add(S, s_{new})$
    **for** $(a \in \Sigma, e \in E)$ **do**
      $T[s_{new} \cdot a, e] \leftarrow$ **Member**$(s_{new} \cdot a \cdot e)$
    **end for**
  **end while**
  $\mathcal{A} \leftarrow MakeHypothesis(S, E, T)$
  $cex \leftarrow$ **Equivalence**$(\mathcal{A})$
  **if** $cex = \perp$ **then**
    **return** $\mathcal{A}$
  **else**
    $e_{new} \leftarrow FindSuffix(cex)$
    $Add(E, e_{new})$
    **for** $(s \in S, a \in \Sigma)$ **do**
      $T[s, e_{new}] \leftarrow$ **Member**$(s \cdot e_{new})$
      $T[s \cdot a, e_{new}] \leftarrow$ **Member**$(s \cdot a \cdot e_{new})$
    **end for**
  **end if**
**end while**
```

Angluin's L* algorithm (1987) is an exact learning algorithm for regular languages. The algorithm learns an unknown regular language U over an alphabet Σ, generating a DFA that accepts U as output. We only provide a brief and informal description of the algorithm; for further details see (Angluin, 1987; Berg et al., 2005).

Algorithm 2 shows the L* algorithm. This version is adapted from Alur et al. (2005), where the membership and equivalence queries have been made more explicit than they appear in Angluin (1987).

The algorithm maintains an *observation table* (S, E, T) that records whether strings belong to U. In Algorithm 2, this table is represented by the two-dimensional array T, with dimensions $|S| \times |E|$, where, informally, we can view S as a set of words that lead from the initial state to states of the hypothesized automaton, and E as a set of words serving as experiments to separate states. The table T itself maps a word $w \in (S \cup S \cdot \Sigma) \cdot E$ to `True` if $w \in U$ and `False` otherwise.

The table is updated by invoking membership queries to the teacher. When the algorithm reaches a consistent and closed observation table (meaning that all states have outgoing transitions for all letters, without contradictions), the algorithm constructs a hypothesized automaton $\mathcal{A}$, and invokes an *equivalence query* to check whether $\mathcal{A}$ is equivalent to the automaton known to the teacher. If the hypothesized automaton accepts exactly U, then the algorithm terminates. If it is not equivalent, then the teacher produces a counterexample showing a difference between U and the language accepted by $\mathcal{A}$.

A simplified run through of the algorithm is as follows: the learner starts with an automaton with one state—the initial state—which is accepting or rejecting according to the classification of the empty word. Then, for every state in the automaton, for every letter in the alphabet, the learner verifies by way of membership queries that for every shortest sequence reaching that state, the continuation from that prefix with that letter is correctly classified. As long as an inconsistency exists, the automaton is refined. Every time the automaton reaches a consistent state (a complete transition function, with no inconsistencies by single-letter extensions), that automaton is presented to the teacher as an equivalence query. If it is accepted, the algorithm completes; otherwise, it uses the teacher-provided counterexample to further refine the automaton.

B. Additional Results

B.1. Random Regular Languages

We show results for extraction from GRU and LSTM networks with varying hidden sizes, trained on small regular languages of varying alphabet size and DFA size. Each extraction was limited to 30 seconds, and had initial refinement depth 10. For each of the combinations of state size and target language complexity, 3 networks of each type were trained, each on a different (randomly generated) language. The full results of these experiments are shown in Table 5. Note that each row in each of the tables represents 3 experiments, i.e. in total $9 \times 2 \times 3 = 54$ random DFAs were generated, trained on, and re-extracted.

We note with satisfaction that in 36 of the 54 experiments, the extraction process reached equivalence on a regular language identical to the target language the network had been trained on. We also note that on one occasion in the LSTM experiments the extraction reached equivalence too easily, accepting an automaton of size 2 that ultimately was not a great match for the network. Such a problem could be countered by increasing the initial split depth, for instance when sampling shows that a too-simple automaton has been

Table 5. 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 3 experiments with the same parameters (network hidden-state size, alphabet size, and minimal target DFA size). Single values represent the average of the 3 experiments, multiple values list the result for each experiment. An extraction time of 30 seconds signals a timed out extraction (for which the last automaton proposed by L* is taken as the extracted automaton).

Extraction from LSTM Networks — Our Method

Hidden Size	Alphabet Size	Language / Target DFA Size	Extraction Time (s)	Extracted DFA Size	Average Extracted DFA Accuracy l=10	l=50	l=100	l=1000	Training
50	3	5	2.9	5,5,6	100.0	99.96	99.86	99.90	100.0
100	3	5	2.9	5,5,2	92.96	92.96	93.73	93.46	91.06
500	3	5	11.8	5,5,5	100.0	100.0	100.0	99.96	100.0
50	5	5	30, 30, 30	68, 59, 115	99.96	99.93	99.76	99.93	99.99
100	5	5	30, 7.7, 30	57, 5, 38	99.96	99.96	99.96	99.90	100.0
500	5	5	30, 20.7, 19.0	5, 5, 5	100.0	100.0	99.93	99.90	100.0
50	3	10	30, 30, 11.1	10, 10, 10	99.96	99.96	99.90	99.90	100.0
100	3	10	7.6, 30, 7.7	10, 10, 11	99.96	99.93	99.96	99.96	100.0
500	3	10	30, 30, 30	10, 9, 10	92.30	92.80	93.70	93.43	92.30

Extraction from GRU Networks — Our Method

Hidden Size	Alphabet Size	Language / Target DFA Size	Extraction Time (s)	Extracted DFA Size	Average Extracted DFA Accuracy l=10	l=50	l=100	l=1000	Training
50	3	5	1.7	5,5,6	100.0	100.0	99.86	99.96	100.0
100	3	5	4.1	5,5,5	100.0	100.0	100.0	99.96	100.0
500	3	5	7.0	5,5,5	100.0	100.0	100.0	100.0	100.0
50	5	5	30, 30, 8.2	150,93,5	100.0	99.90	99.93	99.86	100.0
100	5	5	9.0, 8.0, 30	5,5,16	100.0	100.0	99.96	99.96	99.99
500	5	5	15.5, 30, 25.6	5,5,5	100.0	100.0	99.96	100.0	100.0
50	3	10	30, 30, 30	11,11,155	99.96	99.83	99.93	99.93	99.99
100	3	10	11.0	11,10,11	100.0	99.93	99.96	99.93	100.0
500	3	10	30, 30, 30	10,10,10	100.0	99.93	100.0	99.90	100.0

Table 6. Results for automata extracted using Omlin & Giles' a-priori quantization as described in their 1996 paper, with quantization level 2, from the same networks used in Table 5 (3 networks for each set of parameters and network type).

Extraction from LSTM Networks — O&G Quantization

Hidden	Alphabet	Language/	Extracted			Coverage / Accuracy (%)											
Size	Size	Target DFA Size	DFA Sizes			l=1		l=5		l=10		l=15		l=50		Training	
50	3	5	3109	3107	3107	100	100	100	100	30.66	83.65	3.87	81.53	0.0	NA	27.44	88.0
100	3	5	2225	2252	2275	100	100	100	100	7.57	80.50	0.07	50.0	0.0	NA	19.31	84.57
500	3	5	585	601	584	100	100	100	100	0.0	NA	0.0	NA	0.0	NA	8.80	71.71
50	5	5	1956	1973	1962	100	100	100	73.93	0.03	100	0.0	NA	0.0	NA	12.39	78.34
100	5	5	1392	1400	1400	100	100	100	64.3	0.0	NA	0.0	NA	0.0	NA	11.19	74.80
500	5	5	359	366	366	100	100	33.43	70.60	0.0	NA	0.0	NA	0.0	NA	6.24	73.92
50	3	10	3135	3238	3228	100	100	100	100	29.43	83.72	4.57	94.19	0.0	NA	27.70	88.80
100	3	10	2294	2282	2272	100	100	100	100	0.90	91.30	0.0	NA	0.0	NA	16.83	81.07
500	3	10	586	589	589	100	100	100	100	0.0	NA	0.0	NA	0.0	NA	8.48	74.77

Extraction from GRU Networks — O&G Quantization

Hidden	Alphabet	Language/	Extracted			Coverage / Accuracy (%)											
Size	Size	Target DFA Size	DFA Sizes			l=1		l=5		l=10		l=15		l=50		Training	
50	3	5	4497	4558	4485	100	100	100	100	50.73	90.52	25.26	91.95	2.66	96.25	42.28	91.08
100	3	5	3188	3184	3197	100	100	100	100	3.50	66.66	0.07	50.0	0.0	NA	19.14	83.63
500	3	5	1200	1221	1225	100	100	100	100	0.0	NA	0.0	NA	0.0	NA	8.98	74.45
50	5	5	2810	2796	2802	100	100	100	87.97	0.10	100	0.0	NA	0.0	NA	14.56	80.61
100	5	5	1935	1941	1936	100	100	100	73.17	0.0	NA	0.0	NA	0.0	NA	12.05	76.39
500	5	5	721	706	749	100	100	91.03	55.94	0.0	NA	0.0	NA	0.0	NA	9.52	71.53
50	3	10	4598	4582	4586	100	100	100	100	15.73	79.76	1.23	70.71	0.0	NA	24.32	86.93
100	3	10	3203	3192	3194	100	100	100	100	0.3	81.25	0.0	NA	0.0	NA	19.18	83.84
500	3	10	1226	1209	1209	100	100	100	100	0.0	NA	0.0	NA	0.0	NA	13.39	76.64

extracted.

We also ran extraction with Omlin and Giles' a-priori quantization on each of these networks, with quantization level 2 and a time limit of 50 seconds. The extractions did not complete in this time. For completeness, we present the size, coverage, and accuracies of these partially extracted DFAs in Table 6. The smaller size of the extracted DFAs for networks with larger hidden size is a result of the transition function computation taking longer: this slows the BFS exploration and thus the extraction visits less states in the allotted time.

B.2. Balanced Parentheses

We trained a GRU and an LSTM network on the irregular language of balanced parentheses, and then attempted to extract automata from these networks. For both, L* at first proposed a series of automata each representing the language of balanced parentheses to increasing nesting depth. Some of these are shown in Fig. 1. For the GRU network, after a certain point in the extraction, we even found a counterexample which showed the network had not generalized correctly to the language of balanced parentheses, and the next automaton returned resembled—but was not quite—an automaton for balanced parentheses nested to some depth. We show this automaton in Fig. 2.

In our main submission, we show the counterexamples returned during a 400-second extraction from the GRU network, as generated either by random sampling or by our method. For completeness, we present now in Table 7 the counterexamples for the LSTM extraction.

B.3. Other Interesting Examples

B.3.1. Counting

We trained an LSTM network with 2 layers and hidden size 100 (giving overall state size $d_s = 2 \times 2 \times 100 = 400$) on the regular language

[a-z]*1[a-z1]*2[a-z2]*3[a-z3]*4[a-z4]*5[a-z5]*$

over the 31-letter alphabet {a,b,...,z,1,2,...,5}, i.e. the regular language of all sequences 1+2+3+4+5+ with lowercase letters a-z scattered inside them. We trained this network on a train set of size 20000 and tested it on a test set of size 2000 (both evenly split on positive and negative examples), and saw that it reached 100% accuracy on both.

We extracted from this network using our method. Within 2 counterexamples (the provided counterexample 12345, and another generated by our method), and after a total of 9.5 seconds, L* proposed the automaton representative of the network's target language, shown in Fig. 3. However, our method did not accept this DFA as the correct DFA for the network. Instead, after a further 85.4 seconds of exploration

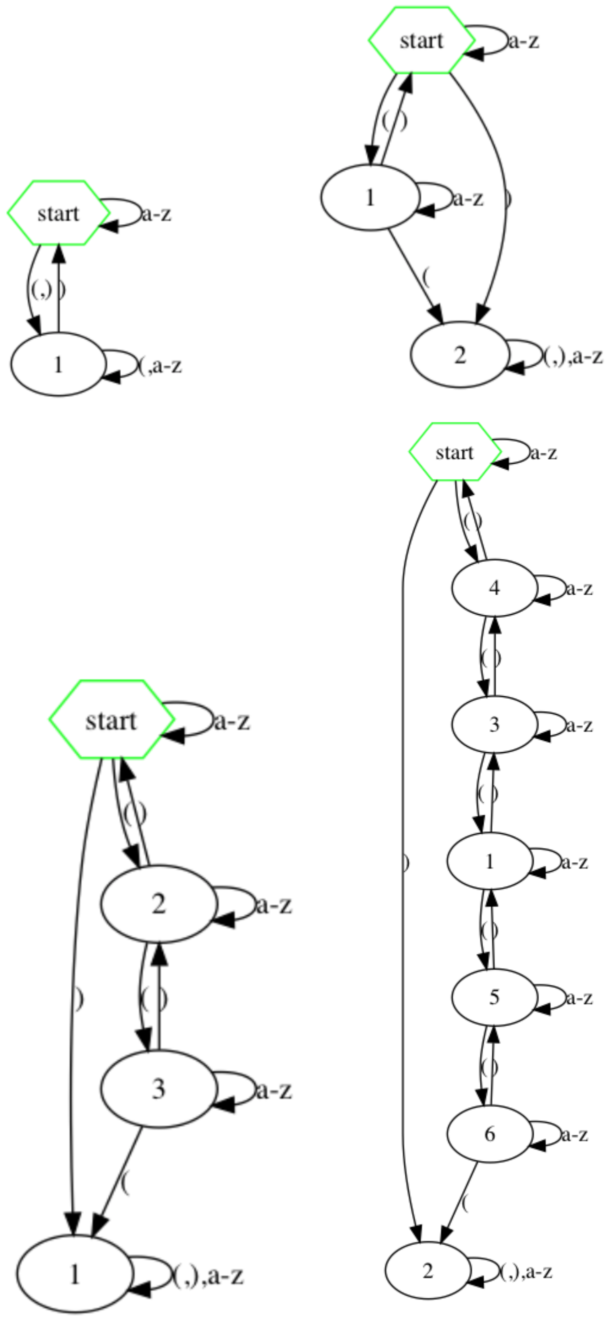


Figure 1. Select automata of increasing size for recognizing balanced parentheses over the 28 letter alphabet a-z,(,), up to nesting depths 1 (flawed), 1 (correct), 2, and 5, respectively.

Table 7. Extraction of automata from an LSTM network trained to 100% accuracy on the training set for the language of balanced parentheses over the 28-letter alphabet `a-z,(,)`. The 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. Each successive equivalence query from L^* was an automaton classifying the language of all words with balanced parentheses up to nesting depth n, with increasing n.

Refinement Based		Brute Force	
Counterexample	Time (seconds)	Counterexample	Time (seconds)
))	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		

Table 8. Counterexamples returned to the equivalence queries made by L^* during extraction of a DFA from a network trained to 100% accuracy on both train and test sets on the regular language [a-z]*1[a-z1]*2[a-z2]*3[a-z3]*4[a-z4]*5[a-z5]*$ over the 31-letter alphabet {`a`,`b`, ...,`d`,`1`,`2`, ...,`5`}. Counterexamples highlighting the discrepancies between the network behavior and the target behavior are shown in bold.

Counterexample Generation for the Counting Language

Counterexample	Generation Time (seconds)	Network Classification	Target Classification
12345	provided	True	True
512345	8.18	False	False
aca11	85.41	True	False
blw11	0.50	True	False
dnm11	0.96	False	False
bzm11	0.90	False	False
drxr11	0.911	True	False
brdb11	0.90	False	False
elrs11	1.16	True	False
hu11	1.93	False	False
ku11	2.59	False	False
ebj11	2.77	False	False
pgl11	3.77	True	False
reeg11	4.16	False	False
eipn11	5.66	False	False

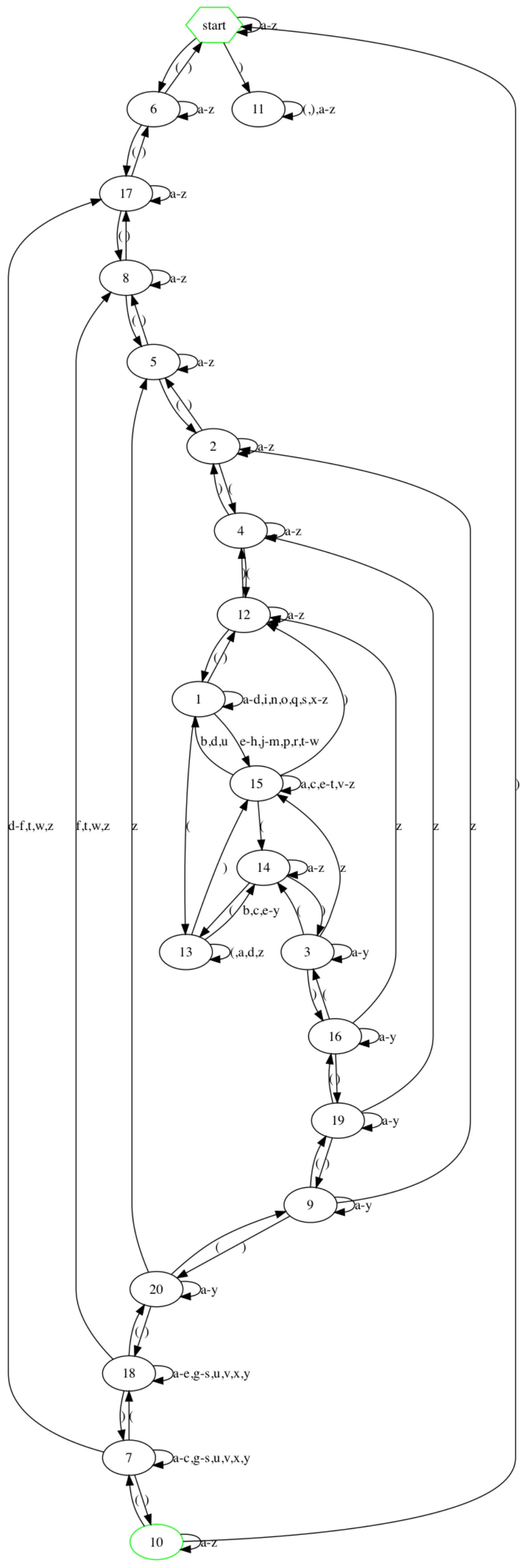


Figure 2. Automaton no longer representing a language of balanced parentheses up to a certain depth. (Showing how a trained network may be overfitted past a certain sample complexity.)

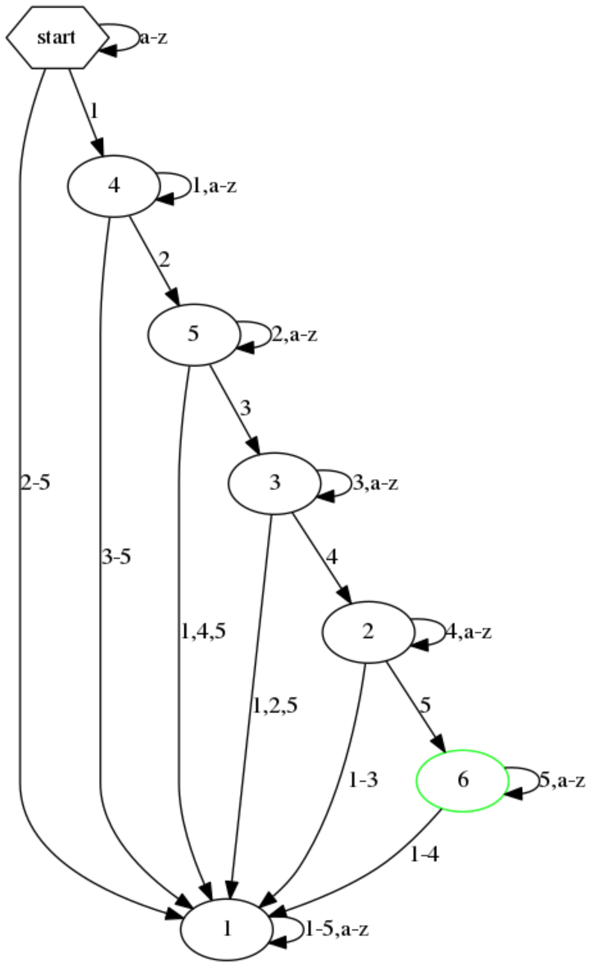


Figure 3. DFA representing the regular language [a-z]*1[a-z1]*2[a-z2]*3[a-z3]*4[a-z4]*5[a-z5]*$ over the alphabet {a,b, ...,z,1,2, ...,5}

.

and refinement, the counterexample `aca11` was found and returned to L*, meaning: our method found that the network accepted the word `aca11` — despite this word not being in the target language of the network and the network having 100% accuracy on both its train and test set.

Ultimately, after 400 seconds our method extracted from the network (but did not reach equivalence on) a DFA with 118 states, returning the counterexamples listed in Table 8 and achieving 100% accuracy against the network on its train set, and 99.9+% accuracy on all sampled sequence lengths. We note that by the nature of our method, the complexity of this DFA is necessarily an indicator of the inherent complexity of the concept to which the trained network has generalized.

B.3.2. TOKENIZED JSON LISTS

We trained a GRU network with 2 layers and hidden size 100 on the regular language representing a simple tokenized JSON list with no nesting,

(\[\])|(\[[S0NTF](,[S0NTF])*\])$

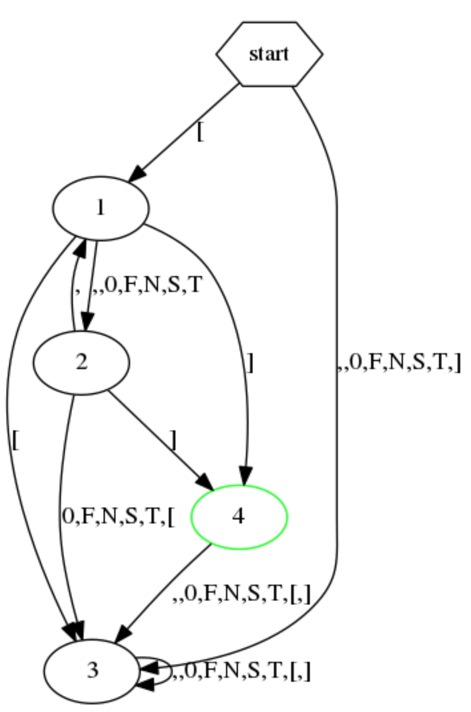
over the 8-letter alphabet {[,],S,0,N,T,F,,}, to accuracy 100% on a training set of size 20000 and a test set of size 2000, both evenly split between positive and negative examples. As before, we extracted from this network using our method.

Within 2 counterexamples (1 provided and 1 generated) and a total of 3.8 seconds, our method extracted the automaton shown in Fig. 4a, which is almost but not quite representative of the target language. 7.12 seconds later it returned a counterexample to this DFA which pushed L* to refine further and return the DFA shown in Fig. 4b, which is also almost but not quite representative of zero-nesting tokenized JSON lists.

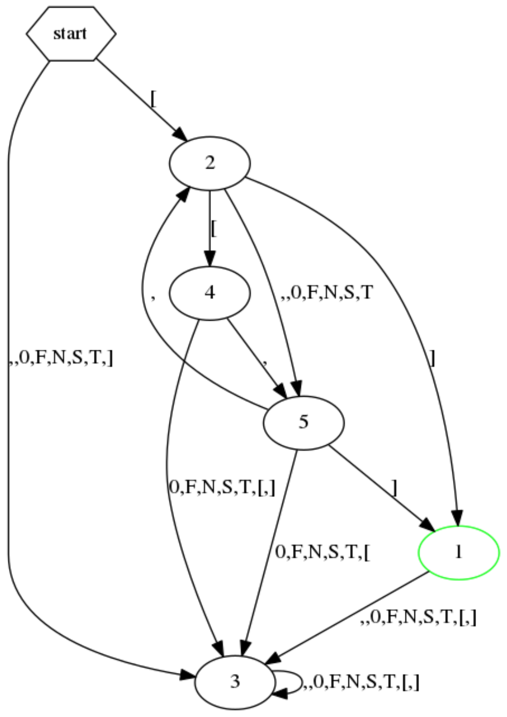
Ultimately, after 400 seconds, our method extracted (but did not reach equivalence on) an automaton of size 441, returning the counterexamples listed in Table 9 and achieving 100% accuracy against the network on both its train set and all sampled sequence lengths. As before, we note that each state split by the method is justified by concrete inputs to the network, and so the extraction of a large DFA is a sign of the inherent complexity of the learned network behavior.

B.4. Train Set Details

The sizes, sample lengths, and positive to negative ratios of the samples in the train sets are listed here (Table 10) for each of the networks used in our main experiments, as well as for the JSON and Counting languages. Note that for some languages (such as the first Tomita grammar, `1*`), there are very few positive/negative samples. For these languages, the test sets are less balanced between positive and negative samples.



(a)



(b)

Figure 4. Two DFAs resembling, but not perfectly, the correct DFA for the regular language of tokenized JSON lists, (\[\])|(\[[S0NTF](,[S0NTF])*\])$. DFA 4a is almost correct, but accepts also list-like sequences in which the last item is missing, i.e., there is a comma followed by a closing bracket. DFA 4b is returned by L* after the teacher (network) rejects 4a, but is also not a correct representation of the target language — treating the sequence `[,` as a legitimate list item equivalent to the characters `S`,`0`,`N`,`T`,`F`

.

Table 9. Counterexamples returned to the equivalence queries made by L* during extraction of a DFA from a network trained to 100% accuracy on both train and test sets on the regular language (\[\])|(\[[S0NTF](,[S0NTF])*\])$ over the 8-letter alphabet { `[`,`]`,`S`,`0`,`N`,`T`,`F`,`,` }. Counterexamples highlighting the discrepancies between the network behaviour and the target behaviour are shown in bold.

Counterexample Generation for the Non-Nested Tokenized JSON-lists Language

Counterexample	Generation Time (seconds)	Network Classification	Target Classification
[]	provided	True	True
[SS]	3.49	False	False
[[,]	7.12	True	False
[S,,	8.61	True	False
[0,F	8.38	True	False
[N,0,	8.07	False	False
[S,N,0,	9.43	True	False
[T,S,	9.56	False	False
[S,S,T,[]	15.15	False	False
[F,T,[	3.23	False	False
[N,F,S,0	10.04	True	False
[S,N,[,,,,	27.79	True	False
[T,0,T,	28.06	True	False
[S,T,0,],	26.63	True	False

Table 10. Train set statistics for networks used in this work. The random regular language networks used in the main work were based on minimal DFAs of size 10 over alphabets of size 3, with 3 languages per hidden state size. We list statistics for their train sets in grouped by the hidden size. The train set sizes and lengths were the same for each of these random languages, but the number of positive/negative samples found each time varied slightly.

Train Set Stats

Language	Architecture	Hidden Size	Train Set Size	Of Which Positive Samples	Lengths in Train Set
Tomita 1	GRU	100	613	14	0-13,16,19,22
Tomita 2	GRU	100	613	8	0-13,16,19,22
Tomita 3	GRU	100	2911	1418	0-13,16,19,22
Tomita 4	GRU	100	2911	1525	0-13,16,19,22
Tomita 5	GRU	100	1833	771	0-13,16,19,22
Tomita 6	GRU	100	3511	1671	0-13,15-20
Tomita 7	GRU	100	2583	1176	0-13,16,19,22
Random 1-3	GRU	50	16092	8038, 7768, 8050	1-15,16,18,...,26
Random 4-6	GRU	100	16092	7783, 7842, 8167	1-15,16,18,...,26
Random 7-9	GRU	500	16092	8080, 8143, 7943	1-15,16,18,...,26
Balanced Parentheses	GRU	50	44697	25243	0-73
Balanced Parentheses	LSTM	50	44816	28781	0-81
emails	LSTM	100	40000	20000	0-34
JSON Lists	GRU	100	20000	10000	0-74
Counting	LSTM	100	20000	10000	0-43

We reiterate that all networks used in this work were trained to 100% train set accuracy and reached at least 99.9% on a set of 1000 samples from each of the lengths $4, 7, 10, ..., 28$.

An explicit description of the Tomita grammars can be found in (Tomita, 1982).

Acknowledgments

The research leading to the results presented in this paper is supported by the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 (PRIME), The Israeli Science Foundation (grant number 1555/15), The Allen Institute for Artificial Intelligence, and The Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI)

References

Alur, R., Černý, P., Madhusudan, P., and Nam, W. Synthesis of interface specifications for java classes. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pp. 98–109, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040314. URL http://doi.acm.org/10.1145/1040305.1040314.

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

Berg, T., Jonsson, B., Leucker, M., and Saksena, M. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci.*, 118:3–18, 2005. doi: 10.1016/j.entcs.2004.12.015. URL https://doi.org/10.1016/j.entcs.2004.12.015.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pp. 144–152, New York, NY, USA, 1992. ACM. ISBN 0-89791-497-X. doi: 10.1145/130385.130401. URL http://doi.acm.org/10.1145/130385.130401.

Cechin, A. L., Simon, D. R. P., and Stertz, K. 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*, SCCC '03, pp. 73–78, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2008-1. URL http://dl.acm.org/citation.cfm?id=950790.951318.

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

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

Elman, J. L. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. ISSN 1551-6709. doi: 10.1207/s15516709cog1402_1. URL `http://dx.doi.org/10.1207/s15516709cog1402_1`.

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

Goldman, S. A. and Kearns, M. J. On the complexity of teaching. *J. Comput. Syst. Sci.*, 50(1):20–31, 1995. doi: 10.1006/jcss.1995.1003. URL `https://doi.org/10.1006/jcss.1995.1003`.

Gorman, K. and Sproat, R. Minimally supervised number normalization. *Transactions of the Association for Computational Linguistics*, 4:507–519, 2016. ISSN 2307-387X. URL `https://www.transacl.org/ojs/index.php/tacl/article/view/897`.

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

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

Jacobsson, H. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Comput.*, 17 (6):1223–1263, June 2005. ISSN 0899-7667. doi: 10.1162/0899766053630350. URL `http://dx.doi.org/10.1162/0899766053630350`.

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

Ororbia II, A. G., Mikolov, T., and Reitter, D. Learning simpler language models with the delta recurrent neural network framework. *CoRR*, abs/1703.08864, 2017. URL `http://arxiv.org/abs/1703.08864`.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

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., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

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

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

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