Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples

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

[1] Technion, Israel
sgailw@cs.technion.ac.il
yahave@cs.technion.ac.il
[2] Bar-Ilan University, Israel
yogo@cs.biu.ac.il

Abstract. We address the problem of extracting an automaton 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 automaton describing the state dynamics of a given RNN. We use Angluin's L* algorithm as a learner and the given RNN as an oracle, employing abstract interpretation of the RNN for answering equivalence queries. Our technique allows automaton-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 with automata extraction from multi-layer GRU and LSTM based RNNs, with state-vector dimensions and underlying automata and alphabet sizes which are significantly larger than in previous automata-extraction work. In some cases, the underlying target language can be described with a succinct automata, yet the extracted automata is large and complex. These are cases in which the RNN failed to learn the intended generalization, and our extraction procedure highlights words which are misclassified by the seemingly "perfect" RNN.

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. While neural networks can reasonably approximate a variety of languages, and even precisely represent a regular language [6], they are in practice unlikely to generalize exactly to the concept being trained, and what they eventually learn in actuality is unclear [26]. Indeed, several lines of work attempt to glimpse into the RNN black-box [25,24,32,33,20,23,22,30,4,7,21,19,28,1,18].

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 [3].

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. A state vector and an input symbol are combined for producing the next state vector. The RNN is essentially a parameterized mathematical function that takes as input a state vector and an input vector, and produces a new state vector. The state vectors can be passed to a classification component that is used to produce a binary or multi-class classification decision. The RNN is trainable, and, when trained together with the 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. We call a combination of an RNN and a classification component an *RNN-acceptor*.

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.
RNN types: The parameterized RNN function can take many forms, from a simple Elman RNN [11] to the more recent *gated RNN* functions such as the Long Short-Term Memory (LSTM) [17] and the gated recurrent unit (GRU) [8,9]. While all these functions are theoretically equally expressive (the Elman RNN can theoretically represent any turing-computable function [29]), the gated architectures were shown empirically to be much easier to train, and capable of learning complex concept classes.

RNNs can be stacked in a *multi-layer* configuration, in which the sequence of output vectors of one RNN are fed as the input vectors to another RNN. Adding layers in this manner was demonstrated empirically to perform better (produce more accurate classifications) in many occasions. In this work we use multi-layer gated RNNs.

RNNs play a central role in deep learning, and in particular in natural language processing. For more in-depth overview, see [15,12,13].

Problem Definition Given an RNN-acceptor R trained to accept or reject sequences over an alphabet Σ, our goal is to extract a deterministic finite-state automaton A that mimics the behavior of R. That is, our goal is to extract a DFA A such that the language $L \subseteq \Sigma^*$ of sequences accepted by A is observably equivalent to that accepted by R. Intuitively, we would like to obtain DFA that accepts *exactly* the same language as the R, but this is generally practically impossible as it requires enumerating all words in Σ^n for some n which is not known a priori.

Existing Techniques There has been a lot of interest in extracting rules from recurrent neural networks. Back in the 90's, a series of work by Omlin, Giles and colleagues [25] presented a technique for extracting a DFA from an RNN[3] by splitting each coordinate of the output space into equal intervals and treating each hypercube as a state. Their technique also prescribed a method for mapping the transitions between the abstracted states. Because this technique applies a uniform (and a priori) quantization over the

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

entire output space, it suffers from inherent state explosion—even a small network explored at coarse granularity leads to a prohibitively large state space.

Another state-quantisation approach is based on *clustering* [33,32,10]. In clustering approaches, all states visited while applying the network to the sequences in its training set are recorded and then fed to a clustering algorithm. The partitioning of the state space defined by the clusters is then explored in the in a similar way to that described in [25], and is much smaller than the quantisation proposed by Omlin and Giles, making it more applicable to networks of today's standards. For reviews on RNN extraction techniques see [32,18].

We note however that both techniques apply an a-priori quantisation, with no method for refinement if it is found to be too coarse. In addition, both may have rather large state space, and as the exploration of the extracted DFA is performed blindly, the explored states cannot be merged until the extraction is complete and the DFA can be minimized.

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 any of two kinds of queries [14]:

- *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 element on which the hypothesis and the concept disagree (a counterexample).

Angluin's L^* algorithm is an exact learning algorithm for learning a DFA from a minimally adequate teacher with knowledge of a regular language [3]. In this context, the questions posed by Angluin's algorithm (the student) to the teacher are:

- *membership queries*: state whether a given word (sequence) is in the language or not
- *equivalence queries*: state whether a given DFA is representative of the regular language held by the teacher or not, and if not, present a word classified differently by the given DFA and the language held by the teacher.

The algorithm maintains a *hypothesized DFA* and refines or accepts it according to the teacher's responses to its queries.

Our Approach We treat DFA extraction from RNNs as an exact learning problem. We use Angluin's L^* exact learning algorithm to *elicit* an automaton from *any type* of recurrent neural network, using the network as the *teacher*.
RNN as a teacher: The teacher in the L^* setup must support the two kinds of queries (membership and equivalence) the algorithm may make while learning.

Answering membership queries is trivially done by running the given word through the RNN-acceptor, and checking whether it accepts the word or rejects it.

Answering equivalence queries, however, is not that simple. The main challenge is that no finite interpretation of the network's states and transitions is given upfront: the state of an RNN are high-dimensional real-valued vectors, resulting in an infinite state space which cannot be exhaustively enumerated.

To address this challenge, we use *an abstract representation* of the RNN. Our abstract representation is guaranteed to be bounded. A unique aspect of our setting, compared to previous L^* works, is that we can 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.

Main Contributions The main contributions of this paper are:

- We present a novel framework for extracting automata from RNN-acceptors. Our technique uses exact learning with abstraction to learn an automata, using the given RNN-acceptor as the teacher.
- A unique aspect of our approach is that it performs exact learning with abstraction of the teacher. In this setting, a disagreement between the teacher and the learner means that either the learner has to refine its representation (as typical in exact learning), or that the abstraction of the teacher is not precise enough and must be refined.
- We implemented the technique and show its ability to extract descriptive automata in settings where other approaches fail. Furthermore, we demonstrate the effectiveness of the method on modern and commonly used gated RNN types—multi-layer LSTM and multi-layer GRU—in contrast to previous work that were applied to the simpler Elman RNN or second-order RNNs.
- In some cases, the extracted DFAs are much more complex than is needed to capture the concept class the RNN was trained on. This happens when the RNN failed to generalize the intended concept, despite having perfect accuracy on large train and test sets. Our method revealed these deficiencies in the trained models, and produced *adversarial inputs*—words that are misclassified by the trained RNN but are not present in the test set.

2 Preliminaries

In this paper we use the following notations and terminology.

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

Recurrent Neural Network Classifier / RNN-acceptor 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 $x_t \in \mathbb{R}^{d_i}$ and returns a state-vector $h_{t+1} \in \mathbb{R}^{d_s}$. For an initial state-vector h_0 and a sequence of input vectors $x_1, ..., x_m$, the function g_R is applied recursively, resulting in the state vectors $h_1, ..., h_m$, where each state h_t corresponds to an input prefix $x_1, ..., x_t$. When the input is sequence of discrete symbols from a given alphabet Σ, each symbol is deterministically mapped to a vector using either a one-hot encoding or an embedding

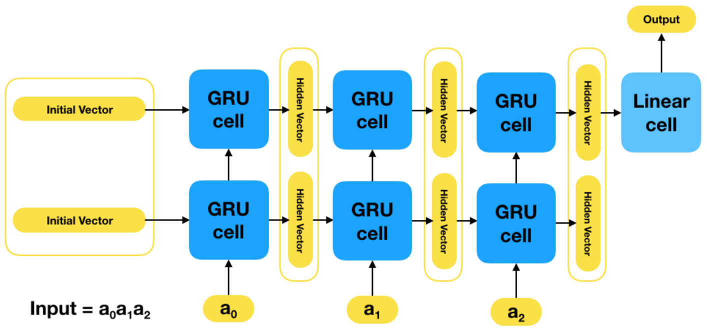


Fig. 1: Illustration of an RNN based network, with two layers and a linear classifier, unrolled on an input sequence of length 3. At each iteration, the concatenation of each layer's state vectors can be seen as the current network state, with the network implicitly defining a deterministic transition function between these states. The linear cell is applied to the final output of the top layer of the network to give its classification on the entire input sequence.

matrix. We use one-hot encoding in this work.[4] As an abuse of notation, we refer to the symbols and their corresponding vectors interchangeably. We denote the state space of the network by $S_R = \mathbb{R}^{d_s}$. In a binary *RNN-acceptor*, there is an additional function $f_R : S_R \to \{Acc, Rej\}$ 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.

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 $h \in S_R$, $\hat{g_R}(h, \epsilon) = h$, 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 $\hat{g_R}(h_{0,R}, w)$, and $f_R(w)$ to denote $f_R(\hat{g_R}(w))$. Within this notation, the classification of a word $w \in \Sigma^*$ by the network is given by $f_R(w)$.

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

A given RNN-acceptor can be interpreted as a deterministic, though possibly infinite, state machine, which we do note is a more powerful model than that of DFAs.

RNN Flavors: The parameterized 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. We use a linear transformation in this work. A common form of g_R is the Elman RNN [11], in which g_R takes the form of an affine transform followed by a non-linearity, $g_R(h, x) = \tanh(W^x x + W^h h + b)$, where W^x and W^h are matrices of dimensions $d_s \times d_i$ and $d_s \times d_s$ respectively and b is a d_s dimensional vector, which are the parameters of the function that needs to be trained. Other popular forms are the Long Short-Term Memory (LSTM) [17] and the Gated Recurrent Unit (GRU) [8,9]. These functions take a more elaborate form based on a differentiable gating mechanism, and were repeatedly demonstrated to be easier to train than the simpler Elman RNN, and to robustly handle long-range sequential dependencies. In this work, we experiment with

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

LSTM and GRU networks as implemented in the PyTorch framework. However, our method is agnostic to the internal form of the RNN functions and treats them as black boxes, requiring access only to the produced state vectors.[5] We refer the interested readers to textbooks such as [15,13] or to the documentation of the PyTorch framework, for the exact forms of the GRU and the LSTM.

Multi-layer RNNs: RNNs are often arranged in layers ("deep RNNs"). In a k-layers layered configuration, there are k RNN functions $g_1, ..., g_k$, where the input sequence $x_1, ..., x_m$ is mapped by g_1 to a sequence of state vectors $h_{1,1}, ..., h_{1,m}$, and the sequence $h_{i,1}, ..., h_{i,m}$ is used as an input to RNN g_i. In a k-layer RNN-acceptor, the function f is applied to the last state vector of g_k. For such multi-layer configurations, we take the state-vector at time t to be the concatenation of the individual layer's state vectors: $h_t = h_{1,t} \cdot\ h_{2,t} ... \cdot h_{k,t}$.

Figure 1 depicts a 2-layer binary RNN-acceptor with a GRU transition function, unrolled on an input sequence of length 3.

3 Extraction Approaches and Their Limitations

Direct Extraction A direct approach to DFA extraction from a trained recurrent neural network is to treat each of its legally reachable state vectors (i.e. reachable during executions of the RNN on legal input sequences) as a state in the extracted DFA. The states may be extracted by exhaustive exploration starting from the initial state. Once all states are explored—i.e. once the RNN is fully explored—one may perform DFA minimization on the extracted DFA to get a clear and concise representation of the network.

The problem with this approach is the sheer enormity of the task at hand—the state vectors maintained and computed in an RNN have values in a continuous interval, and so *exploring the network in this way could theoretically and most likely also practically simply not terminate*.

A-priori State-space Quantization In their 1996 paper [25], Omlin and Giles showed that RNN-states tend to cluster in small areas in the network state space, and—along with an assumption on the continuity of the network's behavior—concluded that it was safe to cluster like-valued state vectors together as one state. For networks with bounded output values, they suggested dividing each dimension of the network state space into $q \in \mathbb{N}$ equal intervals, yielding q^{d_s} subsets of the output space with d_s being the length of the state vectors.

Given a neural network R with state space S and alphabet Σ, and a partitioning function p: $S \to \mathbb{N}$, Omlin and Giles presented a method 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.

[5] The transition function of a single LSTM cell is often described as taking a triplet of input-vector, state-vector and memory-vector, and returning a pair of state-vector and memory-vector. For our purposes, 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.

This extraction method is presented in Figure. 2. It can be considered as a sheared BFS exploration of the continuous states of the network, with the exploration prescribing the classifications and transitions of every abstract state it passes through, and shearing wherever it reaches an abstract state that has already been visited. We note that it is guaranteed to extract a deterministic, finite automaton (DFA) from any network and finite partitioning.

In both their work [25] and more recent research by others [10,32], this extraction method has been shown to produce DFAs that are reasonably representative of the network at hand—provided the given partitioning captures the differences between the network states well enough.

```
$Q, F, \delta \leftarrow \emptyset$
$New \leftarrow \{h_0\}$
while $New \neq \emptyset$ do
    $h \leftarrow$ pick and remove from New
    $q \leftarrow p(h)$
    if $q \notin Q$ then
        $Q \leftarrow Q \cup \{q\}$
        if $f_N(h) = Acc$ then $F \leftarrow F \cup \{q\}$
        for $\sigma \in \Sigma$ do
            $h' \leftarrow g_N(h, \sigma)$
            $\delta \leftarrow \delta \cup \{((q, \sigma), p(h'))\}$
            $New \leftarrow New \cup \{h'\}$
        end
    end
end
```

Fig. 2: Pseudo-code of network unrolling with state space partitioning $p : R_N \to \mathbb{N}$. The functions of the network are marked N subscript.

Notation: For convenience, we denote the DFA extracted by this method from a network R and partitioning p as $A^{R,p}$, and its states by $Q^{R,p}$. In addition, for every $q \in Q^{R,p}$, we refer to its 'discovering' BFS search path as w_q and the continuous state with which it was discovered as h_q. Precisely, h_q is the continuous state $h_q = \hat{g}(w_q)$ through which q was added to $Q^{R,p}$, and $w_q \in \Sigma^*$ is the BFS path with which h_q was reached in the exploration.

Limitations: As networks of standard size may have hundreds to thousands of hidden values (i.e. state vectors with thousands of values), blindly exploring them as proposed in [25]—even with a low quantization level—is often practically impossible. Without an oracle's guidance, minimizing an automaton is impossible before unrolling it in its entirety, and so this is true even if, when minimized, the representative automaton for the network is quite small. This scaling limitation is inherent to any approach using blind unrolling, including recent attempts [10].

4 Learning Automata from RNN using L*

In this section we explain how we apply the L* algorithm to learn a DFA from a given recurrent neural network. First, we briefly review the L* algorithm, and then we explain how to implement its operations using the underlying RNN.

4.1 The L* Algorithm

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

Fig. 3: L* Algorithm with explicit membership and equivalence queries.

Angluin's L* algorithm [3], 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 [3,5].

Fig. 3 shows the L* algorithm. This version is adapted from [2] where the membership and equivalence queries have been made more explicit than they appear in [3].

The algorithm requires a teacher to answer two types of queries: membership queries, in which the teacher must classify words presented by the learner, and equivalence queries, in which the teacher must accept or reject automata proposed by the learner, based on whether or not they correctly represent the target language. If the teacher rejects an automaton, it must also provide a counterexample — a word the proposed automaton misclassifies w.r.t. the target language.

The algorithm maintains an *observation table* (S, E, T) that records whether strings belong to U. In Fig. 3, this table is represented by the two-dimensional array T, with dimensions $|S| \times |E|$, where S is intuitively a set of words that lead from the initial state to states of the hypothesized automaton, and E is 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 (intuitively 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 for the 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 refine the automaton some more.

The L* algorithm is guaranteed to always present a minimal DFA consistent with all samples queried so far, and to return a minimal DFA for the target language 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 [3,5].

4.2 Learning with an RNN as a Teacher

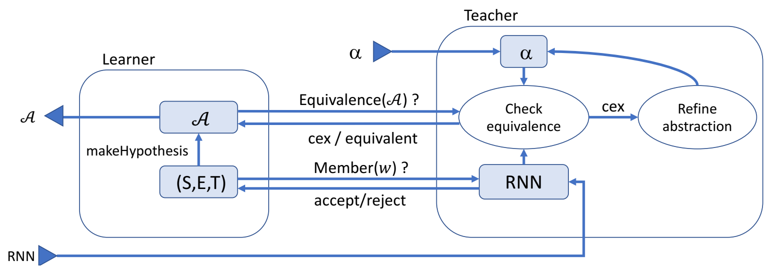


Fig. 4: An overview of our approach for learning with an RNN as a Teacher. The input is an RNN and an initial abstraction α, the output is a DFA $\mathcal{A}$ that maintains at least the observations made by the RNN under α.

To use the L* algorithm, we need to provide an implementation for the **Member** and **Equivalence** operations of Fig. 3. A high-level overview of our approach is presented in Fig. 4.

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

To implement **Equivalence** we check the equivalence of the L* hypothesized automaton $\mathcal{A}$, against an abstraction $A^{R,p}$ of the network, where p is an abstraction function over the network's state space. The abstraction has to be coarse enough to facilitate feasible computation, and fine enough to capture the interesting observations made by the network. We therefore consider an abstraction refinement setting in which we start with a coarse abstraction of the network, and refine it only as required by the learner. Note that while it can be said with certainty that each automaton L* provides is more complicated than the previous, this is not necessarily true for the network abstractions — which may by change of the partitioning happen to minimize to smaller automatons after refinement, or even extract smaller automatons entirely (a transition function, partitioning, and refinement operation for which this may happen can be constructed). The learner and teacher iteratively refine their automatons and abstractions, generating two series of automata — $\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n$ and $A^{R,p_0}, A^{R,p_1}, ..., A^{R,p_m}$ — until the automata either converge, or the interaction is terminated.

If the two converge, provided some further mathematical proof of the network's stability, we may find that the final automatons proposed by the two players are truly representative of the network at hand. More likely, the two will not converge before the iterations become extremely slow, or some memory bound is passed. In this case we take the last automaton proposed by L*. In practice this automaton is shown to generalise well to the entire network (section 7).

We may also choose to return the last abstracted automaton, focusing on refining the abstraction as much as possible at each equivalence query before returning a counterexample to L*. For aggressive refinement techniques, in which the abstraction defines a very large number of states, this method may be unreasonable. For coarser refinement techniques, such as that presented in 6, the extracted DFAs are not as good as those presented by L*, and so we still prefer L*. We suspect their poor performance in relation to L* is due to the lack of 'foresight' they have in comparison to L*'s many separating suffix strings.

5 Answering Equivalence Queries

Given a network R, a partitioning function p over its state space, 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 exploring as little of $A^{R,p}$ as necessary to provide an answer. 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 $A^{R,p}$).

For this section we assume we have a partitioning function of the network state space, $p : S_R \to \mathbb{N}$, and a refinement operation $ref : p, h, H \mapsto p'$ which receives a partitioning function p, a network state h, and a set of network states $H \subseteq S \setminus \{h\}$, and returns a new partitioning function 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 $p'(h_1) \neq p'(h_2)$.

We note that in practice, condition 1 may be relaxed to just separating at least one of the vectors H from h, and that our method can and has overcome an imperfect split between h and H before. One refinement operation satisying the relaxed conditions is presented in section 6.

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 automatons are equivalent — be equivalent to exactly one state in the DFA $\mathcal{A}$, w.r.t. classification of the states and projection of $\delta^{A^{R,p}}$ onto $\mathcal{A}$. This allows us to associate between states of the two automatons on the fly, using conflicts in the association as definite indicators of inequivalence of the automatons.

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

As the exploration of the network R with the partitioning p in effect carries out a (sheared) BFS traversal of $A^{R,p}$, we can associate equivalent A-states and L-states between the two automatons according to a parallel traversal of the two, verifying as we go that the state classifications and the projection of the transition functions are equivalent.

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 (minimal) L-states, as *clustering conflicts*. (The inverse, in which one minimal L-state is associated with several A-states, is not necessarily a problem, as $A^{R,p}$ is not necessarily minimal and these states may truly be equivalent.)

As the ulterior motive is to find inconsistencies between the proposed automaton $\mathcal{A}$ and the given network R, and the exploration of $A^{R,p}$ runs atop an exploration of the actual R-states, we also assert during the exploration the identical classification of each R-state $h \in S_R$ encountered while mapping $A^{R,p}$ with the L-state $q^* \in Q_{\mathcal{A}}$ the parallel traversal of $\mathcal{A}$ reaches with h's discovery. As the classification of a newly discovered A-state is determined by the R-state with which it was first mapped, this assertion will also find any abstract classification conflict. We thus refer to failures of this assertion as *classification conflicts*, and check only for them and for clustering conflicts.

5.1 Conflict Resolution and Counterexample Generation

Classification conflicts are a sign that a path w has been traversed in the exploration of $A^{R,p}$ for which $f_R(w) \neq f_{\mathcal{A}}(w)$, and so necessarily that w is a counterexample to the equivalence of the proposed automaton $\mathcal{A}$ and the network R. They are resolved by returning the path w as a counterexample to L* so it may refine $\mathcal{A}$ and provide a new automaton.

Clustering 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, has been reached again with a new path w_2 for which $q_1^* = \hat{\delta_{\mathcal{A}}}(w_1) \neq \hat{\delta_{\mathcal{A}}}(w_2) = q_2^*$ (i.e. the L-states reached by w_1 and w_2 are different). As $\mathcal{A}$ is a minimal automaton, q_1^* and q_2^* are necessarily inequivalent states, meaning there exists a sequence $s \in \Sigma^*$ for which $f_{\mathcal{A}}(\hat{\delta_{\mathcal{A}}}(q_1^*, s)) \neq f_{\mathcal{A}}(\hat{\delta_{\mathcal{A}}}(q_2^*, s))$, and so in particular for which $f_{\mathcal{A}}(w_1 \cdot s) \neq f_{\mathcal{A}}(w_2 \cdot s)$. Conversely, the arrival of both w_1 and w_2 to the same A-state q in $A^{R,p}$ implies that for any sequence $s' \in \Sigma^*$, and in particular for $s' = s$, $\delta^{\hat{R},p}(w_1 \cdot s') = \delta^{\hat{R},p}(w_2 \cdot s')$, 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. In order to determine the 'offending' automaton, we pass both 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. 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 the refinement operation to yield a new partitioning p' for which 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' that have reached the conflicted A-state $q \in Q^{R,p}$ without conflict before w was traversed. As such, the classifications of *all* the words $w_2 \cdot s, w' \cdot s$ are tested against the network, prioritising (as before) returning a counterexample over refining the partitioning. If eventually it is the partitioning that is refined, then the R-state $h = \hat{g}(w_2)$ is split from all (or at least some, with the relaxed contitions) of the R-states $h' = \hat{g}(w')$ for w' that have already reached q in the exploration.

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 memory, time, or refinement) is exceeded. We note that in practice — and very often so with the refinement operation that we present — there are cases in which starting over is equivalent to merely fixing the abstraction of the R-state that triggered the refinement and continuing from there.

Pseudocode for this entire equivalence checking procedure is presented in Fig. 5.

```
$p \leftarrow p_0$
While True:
    $Q, F, \delta \leftarrow \emptyset$
    New $\leftarrow$ Empty FIFO List
    $q_0 \leftarrow p(h_0)$
    Push(New, $h_0$)
    Pairings $\leftarrow \{(q_0, q_0^*)\}$
    Paths $\leftarrow \{h_0, \epsilon\}$
    Visitors $\leftarrow \{q_0, \{h_0\}\}$
    # Exploration:
    While New $\neq \emptyset$:
        $h \leftarrow$ Peek(New)
        $q \leftarrow p(h)$
        $q^* \leftarrow$ Pairings($q$)
        If $f_R(h) \neq f_\mathcal{A}(q^*)$:
            return Reject, Paths($h$)
        If $q \notin Q$:
            # Add new state
            $Q \leftarrow Q \cup \{q\}$
            If $f_R(h) = Acc$:
                $F \leftarrow F \cup \{q\}$
            # Explore new state
            For $\sigma \in \Sigma$: # in alphabetical order
                $h' \leftarrow g_R(h, \sigma)$
                $q' \leftarrow p(h')$
                $\delta \leftarrow \delta \cup \{((q, \sigma), q')\}$
                Push(New, $h'$)
                $q'^* \leftarrow \delta_\mathcal{A}(q^*, \sigma)$
                Paths $\leftarrow$ Paths$\cup(h', Paths(h) \cdot \sigma)$
                Visitors($q'$) $\leftarrow$ Visitors($q'$)$\cup\{h'\}$
                If $\exists(q', q_2^*) \in$ Pairings s.t. $q_2^* \neq q'^*$:
                    Find $s \in \Sigma^*$ s.t. $f_\mathcal{A}(\hat{\delta_\mathcal{A}}(q'^*, s)) \neq f_\mathcal{A}(\hat{\delta_\mathcal{A}}(q_2^*, s))$
                    For $h \in$Visitors($q'$):
                        If $f_R$(Paths($h$) $\cdot\, s$) $\neq f^*$(Paths($h$) $\cdot\, s$):
                            return Reject, Paths($h$)$\cdot s$
                    # no problem found in $A^*$ - partitioning must be too coarse
                    $p \leftarrow ref(p, h'$, Visitors($q'$)$\backslash h'$)
                    break
                Pairings $\leftarrow$ Pairings$\cup(q', q'^*)$

        Pop(New)
    If New $= \emptyset$:
        return Accept
```

Fig. 5: Pseudo-code for equivalence checking of a recurrent neural network R and a minimal automaton A^*, with initial state space partitioning p_0. In sets of tuples T, $T(e)$ is used to denote the element e' for which $(e, e') \in T$. With two exceptions, all such sets here are maintained such that there is at most one such e' for every element e, and that such a pair is always added to T before $T(e)$ is accessed. The two exceptions are Visitors, for which Visitors(q) is taken to be the empty set if there is no such e', and Paths, for which one could theoretically reach a state where a continuous network state $h \in S_R$ is accessed twice, with two different traversal paths. This can be remedied in a variety of ways, such as marking each network state with a counter and treating different encounters as different states, or maintaining in Paths not tuples of states and paths but rather states and lists of paths.

6 Abstraction Refinement

In this section, we present a novel network state space quantisation and refinement method.

Our method is entirely scaleable (it is not affected by the size of the R-states) and very conservative — each refinement increases the number of A-states by a constant amount (generally adding only a single A-state). It is also cautious enough to (generally) not affect the abstraction of all R-states visited so far in the exploration, allowing an equivalence query exploration to continue from where the refinement was invoked rather than start the exploration from the top.

In Section 7 we show that this conservative abstraction is strong and fast enough to allow the equivalence query exploration to quickly find counterexamples to L*'s proposed automatons.

6.1 Initial Abstraction

The R-state of a multi-layer RNN R is the concatenation of all of the state vectors of its individual layers, such that its state space S_R is the d_s-dimensional space I^{d_s} where I is the range of possible values its state vectors may get ($[-1, 1]$ for GRUs and $\mathbb{R}$ for LSTMs[6]) and d_s is the total length of its state vectors. We call a partitioning function $\alpha : S_R \to \mathbb{N}$ an abstraction of the state space S_R and refer to the group $Im(\alpha)$ as the abstract states (A-states) imposed by α on the network ($Q^{R,\alpha} \subseteq Im(\alpha)$).

We note that over $\mathbb{R}^{d_s}$ a fixed per-dimension quantization such as that suggested in [25] over $\mathbb{R}$ would be either infinite or illogical (treating an infinitely wide range of values as equivalent), and also that for any hidden value range $I \subseteq \mathbb{R}$, on networks with dimensions of today's standards, such a quantization would create an A-state space far larger than could reasonably be explored. (Consider for instance a network with hidden size 500, a completely reasonable size by today's standards, and the most basic non-trivial quantization level: 2).

As such, we actually begin our abstraction with no separation at all: $\alpha_0 : h \mapsto 0$, and refine it only when such an operation is invoked by the equivalence query exploration.

6.2 Support-Vector based Refinement

We recall from section 5 that in order to resolve certain cluster conflicts in equivalence queries we need a refinement operation $ref : \alpha_i, h, H \mapsto \alpha_{i+1}$ that receives an abstraction α_i, an R-state $h \in S_R$, and a set of R-states $H \subseteq S_R \setminus \{h\}$, and returns a refined abstraction α' satisfying:

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

In addition to this, we want this operation to be:

[6] More precisely, an LSTM's cell vector has values in range $[-1, 1]$ and its memory vector has values in $\mathbb{R}$, but this is not important for our method

3. efficient with regard to its own implementation,
4. time and memory efficient with regard to the refined abstraction α' (storage of α' and computation of its values),
5. not so heavy handed as to lead to unnecessary A-state explosion (i.e., able to give a new abstraction α' for which $Im(\alpha')$ is as small as possible), and
6. capable of generalizing well enough to separate future values in the possible true 'cluster' containing the R-states H from the given, inequivalent, R-state h

To illustrate the last point, imagine a refinement separating $h = (h_1, h_2, ..., h_d)$ from H by (further) splitting the first dimension of S_R along the value $(h_1 - \epsilon)$, with ϵ being some extremely small value and all vectors in H being on the other side of the split. The exhaustive exploration of the RNN with the returned abstraction α' runs the risk of encountering another vector h' that is similar to h and should be clustered with it, but is just on the wrong side of the split, causing effectively the same cluster conflict and requiring another refinement.

Intuitively, we would like to allocate a region around the R-state h that is large enough to contain other continuous R-states that behave similarly, yet still separate it from neighboring R-state vectors (i.e., vectors in H) that behave differently. We achieve this by fitting an SVM classifier with RBF kernel to separate the single vector h from the set H.[7] The max-margin property of the SVM ensures a large space around h, while the gaussian RBF kernel allows for a complex and non-linear partitioning of the space.

The A-state $\alpha(h)$ to which all of these continuous vectors (R-states) were previously mapped can then be split by this classifier, yielding a new abstraction α' with exactly one more A-state than defined by α. We track the sequence of refinements made by the algorithm by arranging the obtained SVM classifiers in a decision-tree data-structure where each node's decision is the corresponding SVM, and the leaves correspond to the current A-states.

Barring failure of the SVM classifier, this approach satisfies requirements 1 and 2 of the classification, and avoids A-state explosion completely by adding only a single A-state at every refinement (note that several R-states may map to the same A-state, but every A-state is *explored* only once—using the first R-state to map to it). SVM classifiers also manage to reasonably separate the R-state space S_R in a way that quickly helps find counterexamples, as can be seen in section 7. We note that the abstraction's storage is linear in the number of A-states it can map to, and that its computation may be linear in this number as well (as opposed to a per-dimension interval based refinement, which is logarithmically stored and computed). However, as this number of A-states also grows very slowly (linearly in the number of refinements carried out), this does not become a problem.

In the case that the SVM classifier applied to the R-states fails to achieve perfect separation of the two clusters, this method will fail to satisfy the first requirement of refinement operations. Nevertheless, in this case most or at least one of the R-states of H will be separated from h, and—should it still be necessary for the equivalence queries posed by L*—a later exploration of the new abstraction can invoke a further

[7] While we view the large-margin classifier with RBF kernel as a natural choice, other kernels or classifiers may yield similar results. We did not explore such variations in this work.

refinement. We see in practice that this fallibility does not pose a hindrance to the goal of the abstraction, which is returning counterexamples to L* equivalence queries.

Note that with this method we split a single A-state $s = \alpha(h)$ into two new A-states, according to the values of h and H. If there were some R-state $h' \in H$ for which $\alpha(h') \neq s$, this R-state would have an unnecessary influence on the split of s. In our equivalence queries, we only pass to the refinement operation vectors h and H that all map to the same A-state, but were this not the case we could remove from H any R-states h' which do not satisfy $\alpha(h') = \alpha(h)$ before training the SVM. (Requirement 1 on these R-states h' against the R-state h would still be satisfied due to the fact that they already map to a separate A-state, and no A-states are merged or mixed by the refinement).

6.3 Practical Considerations

As the initial abstraction function $\alpha_0 : h \mapsto 0$ and refinement operation are extremely coarse, in its initial stages the method runs the risk of not separating the R-states strongly enough to generate counterexamples for the extremely small automatons that L* may propose (i.e., not creating enough A-states to give informative responses to L*'s equivalence queries).

For instance, in the case of a network that does not accept any words of length shorter than 2, the initial automaton $\mathcal{A}_0$ proposed by L* will comprise of a single, rejecting, L-state. As the initial abstraction α_0 also defines a single A-state, with classification according to a single sample of that state — the first visitor, which is the empty sequence — A^{R,α_0} will also be a single-state rejecting automaton. In this scenario the teacher will accept $\mathcal{A}_0$ and the extraction will terminate even though the network may be far more complex.

To counter this initial too-simple interpretation of the network, two measures are taken:

1. A single accepting and rejecting sequence are provided to the teacher as potential counterexamples to be checked against every automaton the learner (L*) proposes. These can be saved from the training set of the network, or found by sampling if the distribution is favorable enough to both classes.
2. The first refinement is done not by SVM, but by a far more aggressive method that generates a great (but not unmanageable) number of new A-states.

The first measure is necessary to prevent the extraction from terminating on a single state automaton and requires only two samples from the training data or some other source. If there do not exist two such samples (the network rejects (or accepts) all sequences it was trained on), then it may be that the network does actually classify all sequences identically, and is representable by a single state automaton.

The second measure prevents the extraction from too readily terminating on a small automaton, by moving to an abstraction with many A-states. One possible way to implement this measure is difference-based refinement on a user defined number of dimensions d, generating a new abstraction α_1 for which $|Im(\alpha_1)| = 2^d$ (provided d is smaller than d_s, the length of the continuous network vectors (R-states) $h \in S_R$).

Such a refinement may seem overly or unnecessarily aggressive—generating far more A-states in the quantization than may even have been explored in the network so far at that point — but practice has shown that without such a heavy handed initiation, the extraction repeatedly accepts overly simplistic and non representative automatons for the networks. This particular type of heavy handed refinement is somewhat intuitive in that it resembles the quantization suggested by Omlin and Giles, but focuses only on the dimensions with the greatest deviation of values between the states being split (which we may guess are the dimensions on which the network stores data most clearly), and splits exactly that range of values (as opposed to an arbitrary division of a dimension, such that the values on one side may never be reached to begin with). In addition—unlike Omlin and Giles' quantization—the amount of potential A-states it generates is controllable, as we decide the initial split depth d.

Aggressive Difference-based Refinement An R-state h may be separated from all or most R-states $h' \in H$ by splitting S_R along the d dimensions on which there is the largest difference between h and the mean h_m of the R-states in H.[8]

The user may choose the value of d according to their own judgement, increasing if they suspect the extraction has converged too readily on a small network. Experimentally, we have seen that d values of around 7-10 (yielding partitions to 128 to 1024 subspaces) generally provide a strong enough initial partitioning of the state space to get the extraction started, without making the abstraction so large as to make exploration infeasible. Note that this partitioning is for future exploration—possibly generating more potential A-states than there are observed R-states at that point—and will not necessarily impact the clustering of the already seen R-states. Intuitively, it provides a controllably-fine view of the network state space S_R based on a basic initial understanding of the dimensions and values with most meaning in S_R.

7 Experimental Results

In this section, we describe the implementation and experimental evaluation of our approach.

7.1 Prototype Implementation and Settings

Implementation Details We implemented the approach in Python, using the PyTorch library for the neural network components, sklearn for the SVM-based refinement, and graphviz for DFA visualization. The total implementation for the various extractions took approximately 2600 lines of code, including an explicit implementation of L* and DFAs. For the SVM classifiers, we used the SVC class of the svm module of sklearn,

[8] This kind of split can be comfortably maintained in a decision tree, generating at the point of the split a tree of depth d with 2^d leaves, such that on each of the layers $i = 1, 2, ..., d$ of the tree each of the nodes are split along the dimension with the i-th largest gap between h and h_m, down the middle of that gap.

with regularization factor $C = 10^4$ to encourage perfect splits and otherwise default parameters.

As experimental results showed that overly long counterexamples could cause the L* automatons to quickly blow up in size, while not necessarily generalizing well (effectively 'overfitting' L* to certain network behaviors), we always returned the shortest possible counterexample at a given point. This included taking the shortest two initial accepting and rejecting samples from the training set for the provided initial counterexamples.

We note that whenever an SVM refinement perfectly split the R-states H from h, the clustering of all explored R-states up to the refinement point remained unchanged (provided the new A-states were labelled accordingly), such that an exploration of the network abstraction from the top would be identical up to the point at which the refinement was invoked. We utilized this to make equivalence queries more efficient, taking only one step back in the exploration—instead of starting from the top—wherever possible.

The networks were extracted using the CPU on Amazon instances of type *p2.xlarge*.

Experiments We trained GRU and LSTM networks with varying state-vector sizes ranging from (50 to 500) on a variety of small randomly-generated regular languages and a few 'real', not necessarily regular, languages such as balanced parentheses and email addresses. The larger randomly generated regular languages have 10 states and alphabet of size 5. While these are rather small DFAs, we note that they are substantially larger than the Tomita grammars [31] used in previous work [32], or the languages explored in [10]. The 'real' languages have larger alphabet sizes.

Each network was trained to accuracy 100% on its training set and compared to its extracted automaton on that same set. Each extraction was run with a time limit of 30 to 400 seconds, after which the last automaton proposed by L* was taken as the result.

7.2 Small Random Regular Languages

We trained a number of 2-layer LSTM and GRU networks with varying hidden-state sizes (50, 100, and 500) on a variety of randomly-generated regular languages. Each network was trained to accuracy 100% on the training samples and had accuracy of at least 99.9% on a random set of test samples. We then applied our method to these networks, with initial refinement of depth 10 (generating 1024 potential A-states after the initial split, along 10 greatest-difference dimensions). The extractions were run with a time limit of 30 seconds, though most completed before that, having reached equivalence (though this is not necessarily a guarantee of true equivalence, it does generally indicate strong similarity). Extracted automatons were compared 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. For each of the combinations of state size and target language complexity, 3 networks of each type were trained. The results of these experiments are shown in Table 1. 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 36 of 54 experiments, the extraction process reached equivalence on a regular language identical to the target language the network had been trained on.

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	2.896	5,5,6	100.0	99.96	99.86	99.90	100.0
100	3	5	2.858	5,5,2	92.96	92.96	93.73	93.46	91.06
500	3	5	11.827	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, 7.715, 30	57, 5, 38	99.96	99.96	99.96	99.90	100.0
500	5	5	30, 20.704, 19.018	5, 5, 5	100.0	100.0	99.93	99.90	100.0
50	3	10	30, 30, 11.120	10, 10, 10	99.96	99.96	99.90	99.90	100.0
100	3	10	7.639, 30, 7.716	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	1.703	5,5,6	100.0	100.0	99.86	99.96	100.0
100	3	5	4.123	5,5,5	100.0	100.0	100.0	99.96	100.0
500	3	5	7.015	5,5,5	100.0	100.0	100.0	100.0	100.0
50	5	5	30, 30, 8.226	150,93,5	100.0	99.90	99.93	99.86	100.0
100	5	5	9.089, 8.029, 30	5,5,16	100.0	100.0	99.96	99.96	99.99
500	5	5	15.499, 30, 25.579	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	11.023	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 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 3 experiments with the same parameters (network hidden-state size, alphabet size, and minimal target DFA size). In each experiment, a random DFA is generated and a network is trained on it to accuracy 100% on the train set, after which a DFA is extracted from and compared to the network. Single values represent the average of the 3 experiments, multiple values list the result for each experiment. Each extraction was run with a time limit of 30 seconds, and 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). For the accuracies on the different lengths, 1000 random words of each length were sampled and compared, and for the accuracy on the training set all of the training set was compared. We 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 extracted.

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	0.00	NA	13.39	76.64

Table 2: Results for automatons extracted using Omlin & Giles' a-priori quantisation as described in [25], with quantisation level 2, from the same networks used in table 1 (3 networks for each set of parameters and network type). Each extraction was limited to 50 seconds. When an extraction timed out, an incomplete automaton was returned — for which some sequences may not have a full transition function mapped. The coverage of the extracted (incomplete) automatons, and their accuracy on the covered words against the networks, is computed on (up to) 1000 samples of each of the lengths 1,5,10,15, and 50, and on all of the training samples. The accuracy is recorded only on those words for which the networks had a classification. We can see that the extraction behaves as not much better than a brute force exploration of the network state space. For networks with larger state sizes, the extraction returns smaller automatons after 50 seconds — as the computation of the successive states is slower. The difference between the success of this method against ours on longer input sequences, both in terms of coverage (ability to provide classification for any input) and accuracy (of that classification), is very stark.

Comparison with Omlin and Giles' a-priori Quantization For comparison, we extracted from each of the above-mentioned networks a DFA according to Omlin and Giles' method of a-priori quantization. These networks had very large R-state vectors, and so this method could not scale (being that it creates an A-state space exponential in the size of the R-state vectors). To avoid excessive memory consumption, we extracted these automatons with a time limit of 50 seconds, after which an incomplete automaton was stored and the classification of words reaching an unmapped A-state (missing transitions) was considered simply unknown. The results of these extractions are recorded in table 2. We note that already in these 50 seconds, the method generally manages to extract an automaton with over 1000 A-states—and that this number is far higher when the method is left to run without a time limit (easily reaching upwards of $30,000$ A-states). The smaller DFA sizes extracted for the networks with larger state sizes (larger R-state vectors) may be a result of the more complicated calculation of the network R-state transitions, slowing down the exploration.

Here we clearly realise an interesting practical advantage our method has over any state-space exploration based extraction: in contrast to exploration based methods, for almost any time limit, our method can always return *some* complete automaton for the network. This is because all exploration based methods have no way to preemptively return any DFA that is meaningful with respect to the unexplored states, whereas our method maintains from a very early point in the extraction a closed DFA that constitutes an ever-improving representation of the network being considered.

7.3 Comparison with Brute-Force Counterexample Generation

The question may arise whether there is merit to the exploration and refinement of abstractions of the network over a simple brute force approach to counterexample generation for L^* equivalence queries—i.e. whether the BFS exploration of the network's R-states turns out to be equivalent to an exploration of all possible input sequences to the network up to a certain length.

We suspect networks in which the ratio between accepting and rejecting sequences is very stark may be closely resembled by a very simple automaton—making it hard to differentiate between the two with random sampling. We train a GRU and an LSTM network on one such language: the language of balanced parentheses (BP) over the 28-letter alphabet $\{$`a`,`b`, ...,`z`,`(`,`)`$\}$ (the language of all sequences over `a-z()` in which every opening parenthesis is eventually followed by a single corresponding closing parenthesis, and vice versa).

We extracted DFAs from these networks using L^*, generating counterexamples either with our method or a brute force counterexample generator. For fairness, in all experiments we provide the brute force generator with the same two initial samples our refinement based counterexample generator was given, allowing it to check and possibly return them at every equivalence query. We also implement it not as an exhaustive search (which is very easily defeated in the case of a large alphabet and regular language that accepts only words past a certain length), but rather as a random sampling of up to 1000 words of each length in increasing order.

We allowed each method 400 seconds to extract automatons from networks trained to 100% accuracy on the training sets. The accuracy of these extracted automatons against the original networks on their training sets is recorded in Table 3, as well as the maximum parentheses nesting depth the L* proposed automatons reached during extraction.

	Accuracy on Training		Max Nesting Depth			
Network	Refinement Based	Brute Force	Refinement Based	Brute Force	Hidden Size	#Layers
GRU	99.98	87.12	8	2	50	2
LSTM	99.98	94.19	8	3	50	2

Table 3: Accuracy of extracted automatons against their networks, which were trained to 100% training accuracy on the balanced parantheses (BP) language. The comparisons were done on the training sets of the networks. The maximum nesting depth the extracted automatons reach (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 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.)

We list the counterexamples and counterexample generation times for each of the BP network extractions in Table 4. Note the generation speed and the succinctness of the counterexamples generated by our method, as opposed to that of those found by the brute force method.

The extraction of the BP automatons had particularly pleasing results: each subsequent automaton 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 (exemplified in Fig. 6). 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 deep-nesting examples in its training set, and extracting the slightly more complicated automaton seen in Fig. 7.

7.4 Other Interesting Examples

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.

Refinement-based vs. Brute-Force Counterexample Generation on the Balanced Parentheses Language

GRU

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

Refinement Based		Brute Force	
Counterexample	Time (seconds)	Counterexample	Time (seconds)
))	provided	))	provided
(())	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 4: Extraction of automatons 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 is returned as a counterexample to L* (whose automaton currently rejects it). For both networks and both methods, the initial counterexample `))` was provided from the network training data.

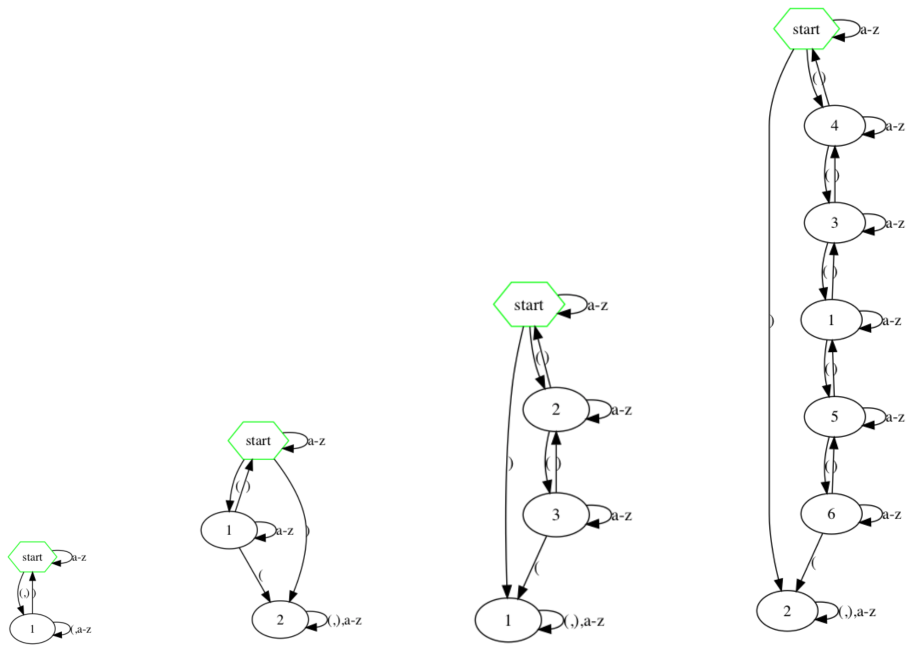


Fig. 6: Select automata of increasing size for recognising balanced parentheses over the 28 letter alphabet a-z,(,) , up to nesting depths 1 (flawed), 1 (correct), 2, and 5, respectively.

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

Table 5: 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 behaviour and the target behaviour are shown in bold.

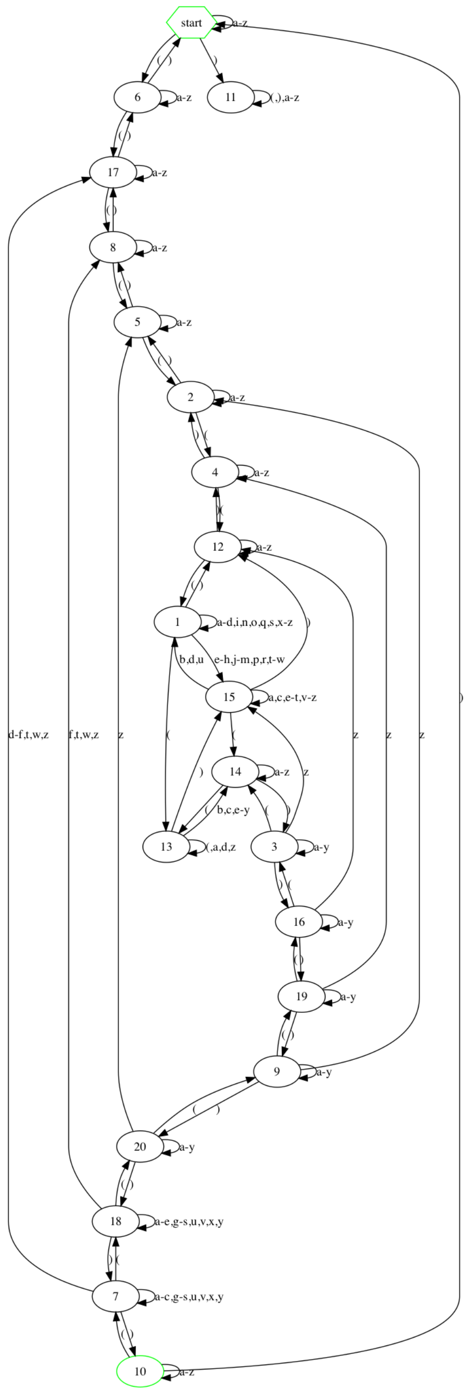


Fig. 7: Automaton no longer representing a language of balanced parentheses up to a certain depth. (Showing how a trained network may be overfitted past a certain sample complexity.)

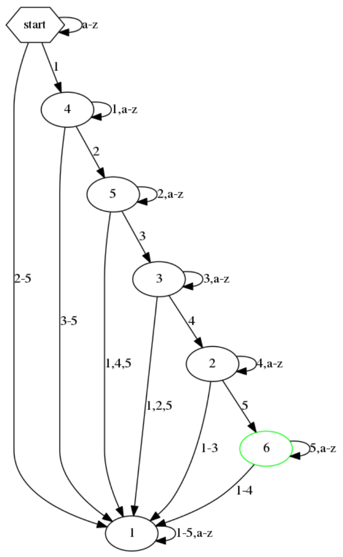
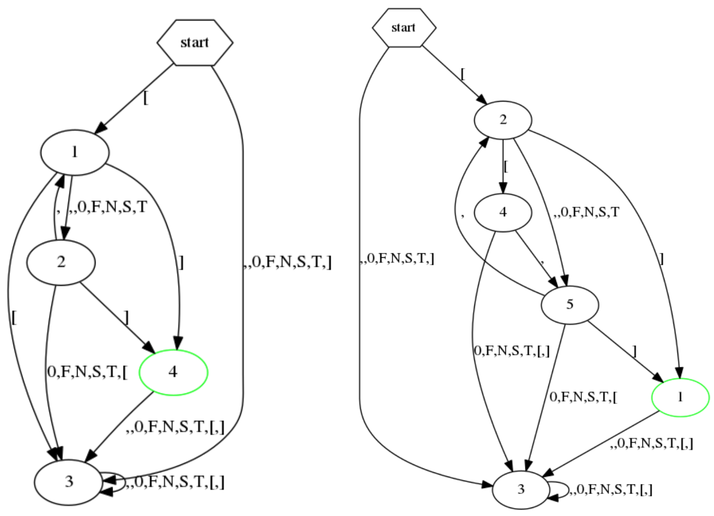


Fig. 8: DFA representing the regular language [a-z]*1[a-z1]*2[a-z2]*3[a-z3]*4[a-z4]*5[a-z5]*$ over the alphabet $\{$`a`,`b`, ...,`z`,`1`,`2`, ...,`5`$\}$

.

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. 8. However, our method did not accept this DFA as the correct DFA for the network. Instead, after a further 85.4 seconds of exploration and refinement, the counterexample `aca11` was found and returned to L*, i.e.: 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 5 and achieving 100% accuracy against the network on its train set, and 99.9+% accuracy on all sampled sequence lengths. We note that as L* always returns the minimal automaton consistent with all data (membership queries and counterexamples) it has received so far, and all data given to L* is based on real executions of the network (without abstraction), every state split of the extracted DFA during extraction is justified by concrete input to the network. This means that extracting a large automaton is not a sign of unnecessary complexity in the extraction but rather a sign of the inherent complexity of the network's learned behavior.



(a) **(b)**

Fig. 9: Two DFAs resembling, but not perfectly, the correct DFA for the regular language of tokenised JSON lists, (\[\])|(\[[S0NTF](,[S0NTF])*\])$. DFA 9a 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 9b is returned by L* after the teacher (network) rejects 9a, 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`

.

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 6: 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.

Tokenised JSON Lists We trained a GRU network with 2 layers and hidden size 100 on a 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. 9a, 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. 9b, 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 6 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.

7.5 Limitations

As L^* is polynomial in $(|Q| + |c| + |\Sigma|)$, where $|Q|$ is the number of states in the automaton being built, c is the longest counterexample returned to L^*, and Σ is the alphabet, this extraction becomes slow once the equivalence queries pass a certain size, and indeed often timeouts in the extraction with our method would happen not during equivalence queries but rather during the refinement of L^*'s proposed automaton $\mathcal{A}$. [9]

Whenever a network did not train well to either its target language or some other regular language, i.e. whenever a network had only a 'fuzzy' understanding of the language being taught, our refinement-based extraction would quickly pick up on the misclassified examples, building a large DFA and then timing out while refining it further. This means that for networks with irregular behavior, extraction can become slow and return large DFAs.

While this means that extraction is not always successful, it also demonstrates that the method can very quickly point out networks that have not trained in the manner which we have expected.

During the course of our experiments, we realized that such cases are annoyingly frequent: for many RNN-acceptors that train to 100% accuracy and exhibit perfect test set behavior on large test sets, our method was able to find many simple examples which the network misclassifies.

[9] Of course for the brute force counterexample generation method, on networks trained to skewed languages (i.e. most words classified identically), timeouts would also often happen during equivalence queries — even if the extracted DFAs had not yet reached the correct representative DFA for the network.

Counterexample Generation for the Email-addresses Language

Counterexample	Time (seconds)	Network Classification	Target Classification
0@m.com	provided	True	True
@@y.net	2.93	False	False
25.net	1.60	True	False
5x.nem	2.34	True	False
0ch.nom	8.01	False	False
9s.not	3.29	False	False
2hs.net	3.56	True	False
@cp.net	4.43	False	False

Table 7: Counterexamples generated during extraction of DFA from a seemingly great LSTM email network. Some of the counterexamples are clear examples of sequences on which the network classification is incorrect with respect to its target language, despite the network having 100% accuracy on both its train and its test set. These counterexamples appear in bold.

For instance, for a network trained to classify simple email addresses over the 38-letter alphabet {`a`,`b`, ...,`z`,`0`,`1`, ...,`9`,`@`,`.`} as defined by the regular expression

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

with 100% accuracy on a 40,000 sample train set and 100% accuracy on a 2,000 sample test set (i.e., a seemingly perfect network), the refinement-based L* extraction quickly returned the counterexamples seen in Table 7, showing clear examples of words the network classifies incorrectly (e.g., the network accepted the non-email sequence `25.net`).[10] While we could not extract a representative DFA from the network in the allotted time frame, our method did show that the network learned a far more elaborate (and incorrect) function than needed.

Beyond demonstrating the counterexample generation capabilities of our extraction method, these results also highlight the brittleness in generalization of trained RNN networks, and suggests that evidence based on test-set performance should be taken with extreme caution.

This reverberates the results of Gorman and Sproat [16], in which a neural architecture based on a multi-layer LSTM was trained to mimic a finite state transducer (FST) for number normalization. Gorman and Sproat show that the RNN-based network, trained on 22M examples and validated on a 2.2M examples development set to 0% error on both, still had occasional errors (though with error rate < 0.0001) when applied to a 240,000 examples blind test set.

8 Conclusions

We present a novel technique for extracting deterministic finite automatons from recurrent neural networks with roots in exact learning. Our approach separates continuous

[10] In contrast, the brute force counterexample generator managed only to return the provided counterexample `0@m.com` to get the L* extraction going, and then timed out after spending the remaining 389 seconds of extraction time searching for a counterexample to L*'s subsequent equivalence query.

network R-states not in an a-priori manner but rather a very coarse, as-needed basis for the purpose of attaining the minimal information necessary to continue the extraction.

Our method scales to networks of any state-size and successfully extracts representative automatons for them provided they can indeed be represented by a DFA. As L* always returns a minimal automaton consistent with all examples it has seen, our method is guaranteed to never extract an automaton with behaviour more complicated than that necessary to represent the network. Moreover, when extraction from a network with a seemingly-simple underlying language becomes complicated, the counterexamples returned during the extraction can point us in the direction of misclassified examples and incorrect patterns that the network has learned without our awareness.

Using our technique, we have shown that our extracted automatons are observationally equivalent or very similar to the networks from which they are extracted. To show that the novelty is not limited to one application, we have also shown that our method may extract informative automatons even for several types of recurrent networks trained on irregular languages, and can provide explicit and otherwise difficult to find misclassified examples for networks that may appear to have trained 'perfectly' on even simple regular languages.

Another advantage of our method is that, in contrast to previous techniques, it can always give reasonable results in a short amount of time, as it maintains an ever-improving hypothesis of the network behaviour and is thus 'always ready'—whereas for exploration based methods, the entire exploration needs to be completed before meaningful result can be returned.

Regardless of its ability to always return *some* result, we have shown that for networks that do correspond to succinct automata, our method actually extracts very good results quickly (generally extracting small, succinct DFAs with accuracies of over 99% with respect to the networks on which they were applied in seconds or tens of seconds even on networks with state dimensions in the hundreds), whereas existing exploration based methods may have nothing to return in the same or even double the time, and left to run to completion often return large and cumbersome automata (with tens of thousands of states).

Another benefit of our method is that, while other approaches require access to a representative sample of positive and negative examples of the language the network was trained on, our method requires very little prior information (only the alphabet and 2 labeled examples) to get started. Relatedly, the method is able to produce failure cases of the network with respect to the intended concept class, which methods that rely on state explorations based on training data cannot do.

9 Related Work

The question of what precisely a black box system has learned can be interesting both for verification purposes and for gaining insight into unknown patterns, and answers for it have been sought in a variety of works dating from the 1990's and until today [25,24,7,32,20,22,33,23,30,4,21,19,28,1,18,27].

This work is an application of Angluin's prominent L* algorithm, [3], to recurrent neural networks trained on regular or near-regular languages, placing the networks in the position of the teacher for the learning process. Angluin's algorithm is known to return a minimal DFA for the language being learned, in polynomial time in Q, $|\Sigma|$ and m — the DFA size, the alphabet size, and longest counterexample size, respectively. In our work we took care to always return the shortest counterexample found, but nevertheless this algorithm becomes quite slow when the networks are representable only by large automatons, and have large alphabets. There exist works on optimizing the L* algorithm [5], which may improve the results of our method on networks that have failed to capture a compact regular structure for their underlying data.

Other methods for extracting automata from recurrent neural networks generally rely on an a-priori quantization of the network's state space, S_R. In their line of work exemplified by their 1996 paper [25], Omlin and Giles proposed a global partitioning of the network state space according to q equal intervals along every dimension (with q being the quantization level). This method has proven non-scalable to networks of today's standards, as the partitioning produces an explosion of the state-space, as we have shown in this work. More recent papers [7,10,32] also propose an a-priori quantization, this time according to a clustering of the network space applied after thorough sampling of the states reachable by real input to the network. As we note earlier, approaches using a priori quantization cannot merge automaton states until the extraction is complete, and are therefore inherently limited in the size of automata they can handle.

References

1. Adi, Y., Kermany, E., Belinkov, Y., Lavi, O., and Goldberg, Y. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. *CoRR abs/1608.04207* (2016).
2. 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* (New York, NY, USA, 2005), POPL '05, ACM, pp. 98–109.
3. Angluin, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (1987), 87–106.
4. Arras, L., Montavon, G., Müller, K., and Samek, W. Explaining recurrent neural network predictions in sentiment analysis. *CoRR abs/1706.07206* (2017).
5. Berg, T., Jonsson, B., Leucker, M., and Saksena, M. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci. 118* (2005), 3–18.
6. Casey, M. Correction to proof that recurrent neural networks can robustly recognize only regular languages. *Neural Computation 10*, 5 (1998), 1067–1069.
7. 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* (Washington, DC, USA, 2003), SCCC '03, IEEE Computer Society, pp. 73–78.
8. 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).
9. Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR abs/1412.3555* (2014).
10. Cohen, M., Caciularu, A., Rejwan, I., and Berant, J. Inducing Regular Grammars Using Recurrent Neural Networks. *ArXiv e-prints* (Oct. 2017).

11. ELMAN, J. L. Finding structure in time. *Cognitive Science 14*, 2 (1990), 179–211.
12. GOLDBERG, Y. A primer on neural network models for natural language processing. *J. Artif. Intell. Res. 57* (2016), 345–420.
13. GOLDBERG, Y. *Neural Network Methods for Natural Language Processing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2017.
14. GOLDMAN, S. A., AND KEARNS, M. J. On the complexity of teaching. *J. Comput. Syst. Sci. 50*, 1 (1995), 20–31.
15. GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. The MIT Press, 2016.
16. GORMAN, K., AND SPROAT, R. Minimally supervised number normalization. *Transactions of the Association for Computational Linguistics 4* (2016), 507–519.
17. HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.
18. JACOBSSON, H. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Comput. 17*, 6 (June 2005), 1223–1263.
19. KÁDÁR, Á., CHRUPALA, G., AND ALISHAHI, A. Representation of linguistic form and function in recurrent neural networks. *CoRR abs/1602.08952* (2016).
20. KARPATHY, A., JOHNSON, J., AND LI, F. Visualizing and understanding recurrent networks. *CoRR abs/1506.02078* (2015).
21. LEI, T., BARZILAY, R., AND JAAKKOLA, T. S. Rationalizing neural predictions. *CoRR abs/1606.04155* (2016).
22. LI, J., CHEN, X., HOVY, E. H., AND JURAFSKY, D. Visualizing and understanding neural models in NLP. *CoRR abs/1506.01066* (2015).
23. LINZEN, T., DUPOUX, E., AND GOLDBERG, Y. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics 4* (2016), 521–535.
24. MURDOCH, W. J., AND SZLAM, A. Automatic rule extraction from long short term memory networks. *CoRR abs/1702.02540* (2017).
25. OMLIN, C. W., AND GILES, C. L. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks 9*, 1 (1996), 41–52.
26. OMLIN, C. W., AND GILES, C. L. Symbolic knowledge representation in recurrent neural networks: Insights from theoretical models of computation. In *Knowledge-based Neurocomputing*, I. Cloete and J. M. Zurada, Eds. MIT Press, Cambridge, MA, USA, 2000, pp. 63–116.
27. PELED, D. A., VARDI, M. Y., AND YANNAKAKIS, M. Black box checking. *Journal of Automata, Languages and Combinatorics 7*, 2 (2002), 225–246.
28. SHI, X., PADHI, I., AND KNIGHT, K. Does string-based neural mt learn source syntax? In *EMNLP* (2016), pp. 1526–1534.
29. SIEGELMANN, H., AND SONTAG, E. On the computational power of neural nets. *Journal of Computer and System Sciences 50*, 1 (1995), 132 – 150.
30. STROBELT, H., GEHRMANN, S., HUBER, B., PFISTER, H., AND RUSH, A. M. Visual analysis of hidden state dynamics in recurrent neural networks. *CoRR abs/1606.07461* (2016).
31. TOMITA, M. 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, 1982), pp. 105–108.
32. WANG, Q., ZHANG, K., II, A. G. O., XING, X., LIU, X., AND GILES, C. L. An empirical evaluation of recurrent neural network rule extraction. *CoRR abs/1709.10380* (2017).
33. ZENG, Z., GOODMAN, R. M., AND SMYTH, P. Learning finite state machines with self-clustering recurrent networks. *Neural Computation 5*, 6 (1993), 976–990.