Weighted Automata Extraction from Recurrent Neural Networks via Regression on State Spaces

Takamasa Okudono, Masaki Waga, Taro Sekiyama, Ichiro Hasuo
National Institute of Informatics & The Graduate University for Advanced Studies
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430
{tokudono, mwaga, sekiyama, hasuo}@nii.ac.jp

Abstract

We present a method to extract a weighted finite automaton (WFA) from a recurrent neural network (RNN). Our method is based on the WFA learning algorithm by Balle and Mohri, which is in turn an extension of Angluin's classic L^* algorithm. Our technical novelty is in the use of *regression* methods for the so-called equivalence queries, thus exploiting the internal state space of an RNN to prioritize counterexample candidates. This way we achieve a quantitative/weighted extension of the recent work by Weiss, Goldberg and Yahav that extracts DFAs. We experimentally evaluate the accuracy, expressivity and efficiency of the extracted WFAs.

1 Introduction

Background *Deep neural networks (DNNs)* have been successfully applied to domains such as text, speech, and image processing. *Recurrent neural networks (RNNs)* (Chung et al. 2014; Hochreiter and Schmidhuber 1997) is a class of DNNs equipped with the capability of processing sequential data of variable length. The great success of RNNs has been seen in, e.g., machine translation (Sutskever, Vinyals, and Le 2014), speech recognition (Zweig et al. 2017), and anomaly detection (Lv et al. 2018; Xiao et al. 2019).

While it has been experimentally shown that RNNs are a powerful tool to process, predict, and model sequential data, there are known drawbacks in RNNs such as interpretability and costly inference. A research line that attacks this challenge is *automata extraction* (Omlin and Giles 1996; Weiss, Goldberg, and Yahav 2018a). Focusing on RNNs' use as *acceptors* (i.e., receiving an input sequence and producing a single Boolean output), these works extract a *finite-state automaton* from an RNN as a succinct and interpretable surrogate. Automata extraction exposes internal transition between the states of an RNN in the form of an automaton, which is then amenable to algorithmic analyses such as reachability and model checking (Baier and Katoen 2008). Automata extraction can also be seen as *model compression*: finite-state automata are usually more compact, and cheaper to run, than neural networks.

Extracting WFAs from RNNs Most of the existing automata extraction techniques target Boolean-output RNNs, which however excludes many applications. In sentiment analysis, it is desired to know the quantitative strength of sentiment, besides its (Boolean) existence (Chaudhuri 2019). RNNs with real values as their output are also useful in classification tasks. For example, predicting class probabilities is a key in some approaches to semi-supervised learning (Yarowsky 1995) and ensemble (Bramer 2013).

This motivates extraction of *quantitative* finite-state machines as abstraction of RNNs. We find the formalism of *weighted finite automata* (WFAs) suited for this purpose. A WFA is a finite-state machine—much like a deterministic finite automaton (DFA)—but its transitions as well as acceptance values are real numbers (instead of Booleans).

Contribution: Regression-Based WFA Extraction from RNNs Our main contribution is a procedure that takes a (real-output) RNN R, and returns a WFA A_R that abstracts R. The procedure is based on the WFA learning algorithm in (Balle and Mohri 2015), that is in turn based on the famous L^* *algorithm* for learning DFAs (Angluin 1987). These algorithms learn automata by a series of so-called *membership queries* and *equivalence queries*. In our procedure, a membership query is implemented by an inference of the given RNN R. We iterate membership queries and use their results to construct a WFA A and to grow it.

The role of equivalence queries is to say when to stop this iteration: it asks if A and R are "equivalent," that is, if the WFA A obtained so far is comprehensive enough to cover all the possible behaviors of R. This is not possible in general—RNNs are more expressive than WFAs—therefore we inevitably resort to an approximate method. Our technical novelty lies in the method for answering equivalence queries; notably it uses a *regression* method—e.g., the Gaussian process regression (GPR) and the kernel ridge regression (KRR)—for abstraction of the state space of R.

We conducted experiments to evaluate the effectiveness of our approach. In particular, we are concerned with the following questions: 1) how similar the behavior of the extracted WFA A_R, and that of the original RNN R, are; 2) how applicable our method is to an RNN that is a more expressive model than WFAs; and 3) how efficient the in-

ference of the WFA A_R is, when compared with the inference of R. The experiments we designed for the questions 1) and 3) are with RNNs trained using randomly generated WFAs. The results show, on the question 1), that the WFAs extracted by our method approximate the original RNNs accurately. This is especially so when compared with a baseline algorithm (a straightforward adaptation of Balle and Mohri (2015)). On the question 3), the inference of the extracted WFAs are about 1300 times faster than that of the original RNNs. On the question 2), we devised an RNN that models a weighted variant of a (non-regular) language of balanced parentheses. Although this weighted language is beyond WFAs' expressivity, we found that our method extracts WFAs that successfully approximate the RNN up-to a certain depth bound.

The paper is organized as follows. Angluin's L^* algorithm and its weighted adaptation are recalled in §2. Our WFA extraction procedure is described in §3 focusing on our novelty, namely the regression-based procedure for answering equivalence queries. Comparison with the DFA extraction by Weiss, Goldberg, and Yahav (2018a) is given there, too. In §4 we discuss our experiment results.

Potential Applications A major potential application of WFA extraction is to analyze an RNN R via its interpretable surrogate A_R. The theory of WFAs offers a number of analysis methods, such as *bisimulation metric* (Balle, Gourdeau, and Panangaden 2017)—a distance notion between WFAs—that will allow us to tell how apart two RNNs are.

Another major potential application is as "a poor man's RNN R." While RNN inference is recognized to be rather expensive (especially for edge devices), simpler models by WFAs should be cheaper to run. Indeed, our experimental results show (§4) that WFA inference is about 1300 times faster than inference of original RNNs.

Since WFAs are defined over a finite alphabet, we restrict to RNNs R that take sequences over a *finite* alphabet. This restriction should not severely limit the applicability of our method. Indeed, such RNNs (over a finite alphabet) have successfully applications in many domains, including intrusion prediction (Lv et al. 2018), malware detection (Xiao et al. 2019), and DNA-protein binding prediction (Shen, Bao, and Huang 2018). Moreover, even if inputs are real numbers, *quantization* is commonly employed without losing a lot of precision. See, e.g., recent (Gupta et al. 2015).

Related Work The relationship between RNNs and automata has been studied in both non-quantitative (Omlin and Giles 1996; Weiss, Goldberg, and Yahav 2018b; 2018a; Michalenko et al. 2019) and quantitative (Ayache, Eyraud, and Goudian 2018; Rabusseau, Li, and Precup 2019) settings. Some of these works feature automata extraction from RNNs; we shall now discuss recent ones among them.

The work by Weiss, Goldberg, and Yahav (2018a) is a pioneer in automata extraction from RNNs. They extract DFAs from RNNs, using a variation of L^*algorithm, much like this work. We provide a systematic comparison in §3.3, identifying some notable similarities and differences.

Ayache, Eyraud, and Goudian (2018) extract a WFA from a black-box sequence acceptor whose example is an RNN. Their method does not use equivalence queries; in contrast, we exploit the internal state space of an RNN to approximately answer equivalence queries.

DeepStellar (Du et al. 2019) extracts Markov chains from RNNs, and uses them for coverage-based testing and adversarial sample detection. Their extraction, differently from our L^*-like method, uses profiling from the training data and discrete abstraction of the state space.

Wang and Niepert (2019) propose a new RNN architecture that makes it easier to extract a DFA from a trained model. To apply their method, one has to modify the structure of an RNN before training, while our method does not need any special structure to RNNs and can be applied to already trained RNNs.

Schwartz, Thomson, and Smith (2018) introduce a neural network architecture that can represent (restricted forms of) CNNs and RNNs. WFAs could also be expressed by their architecture, but extraction of automata is out of their interest.

A major approach to optimizing neural networks is by compression: *model pruning* (Han et al. 2015), *quantization* (Gupta et al. 2015), and *distillation* (Bucila, Caruana, and Niculescu-Mizil 2006). Combination and comparison with these techniques is interesting future work.

2 Preliminaries

We fix a finite alphabet Σ. The set of (finite-length) words over Σ is Σ^*. The *empty word* (of length 0) is denoted by ε. The length of a word $w \in \Sigma^*$ is denoted by $|w|$. We recall basic notions on WFAs. See (Droste, Kuich, and Vogler 2009) for details.

Definition 1 (WFA)**.** A *weighted finite automaton* (WFA) over Σ is a quadruple $A = \big(Q_A, \alpha_A, \beta_A, (A_\sigma)_{\sigma\in\Sigma}\big)$. Here Q_A is a finite set of *states*; α_A, β_A are row vectors of size $|Q_A|$ called the *initial* and *final* vectors; and A_σ is a *transition* matrix of σ, given for each $\sigma \in \Sigma$. For each $\sigma \in \Sigma$, A_σ is a matrix of size $|Q_A| \times |Q_A|$.

Definition 2 (configuration of a WFA)**.** Let A be the WFA in Def. 1. A *configuration* of A is a row vector $x \in \mathbb{R}^{Q_A}$. For a word $w = \sigma_1\sigma_2\dots\sigma_n \in \Sigma^*$ (where $\sigma_i \in \Sigma$), the *configuration* of A at w is defined by $\delta_A(w) = \alpha_A^\top \cdot \big(\prod_{i=1}^n A_{\sigma_i}\big)$.

Obviously $\delta_A(w) \in \mathbb{R}^{Q_A}$ is a row vector of size $|Q_A|$; it records the weight at each state $q \in Q_A$ after reading w.

Definition 3 (weight $f_A(w)$ of a word in a WFA)**.** Let a WFA A and a word $w = \sigma_1 \dots \sigma_n$ be as in Def. 2. The *weight* of w in A is given by $f_A(w) = \alpha_A^\top \cdot \big(\prod_{i=1}^n A_{\sigma_i}\big) \cdot \beta_A$, multiplying the final vector to the configuration at w.

Example 4. Let $\Sigma = \{a, b\}$, $Q_A = \{q_1, q_2, q_3\}$, $\alpha_A = (1\ 2\ 3)^\top$, $\beta_A = (0\ -1\ 1)^\top$, $A_a = \begin{pmatrix} 1 & 2 & -1 \\ 3 & 0 & 0 \\ 0 & 4 & 0 \end{pmatrix}$, and $A_b = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 3 & 0 \\ -2 & 4 & 0 \end{pmatrix}$. For the WFA $A = (Q_A, \alpha_A, \beta_A, (A_\sigma)_{\sigma\in\Sigma})$ over Σ, and $w = ba$, the config-

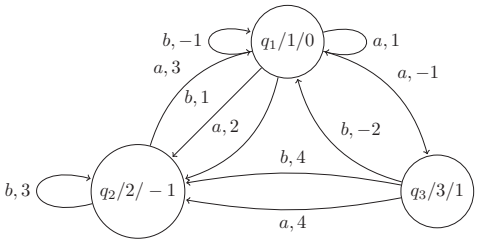

Figure 1: An illustration of WFA A in Example 4. In a state label "$q/m/n$", q is a state name and m and n are the initial and final values at q, respectively. In the label "σ, p" of the transition from q_i to q_j, p is $A_\sigma[i,j]$, where $\sigma \in \Sigma$ and $A_\sigma[i,j]$ is the entry of A_σ at row i and column j.

uration $\delta_A(w)$ and the weight $f_A(w)$ are as follows.

$$\begin{aligned}\delta_A(w) &= \alpha_A^\top A_b A_a \\ &= (1 \quad 2 \quad 3)\begin{pmatrix}-1 & 1 & 0\\ 0 & 3 & 0\\ -2 & 4 & 0\end{pmatrix}\begin{pmatrix}1 & 2 & -1\\ 3 & 0 & 0\\ 0 & 4 & 0\end{pmatrix} \\ &= (50 \quad -14 \quad 7)\end{aligned}$$

$$\begin{aligned}f_A(w) &= \alpha_A^\top A_b A_a \beta_A \\ &= (1 \quad 2 \quad 3)\begin{pmatrix}-1 & 1 & 0\\ 0 & 3 & 0\\ -2 & 4 & 0\end{pmatrix}\begin{pmatrix}1 & 2 & -1\\ 3 & 0 & 0\\ 0 & 4 & 0\end{pmatrix}\begin{pmatrix}0\\ -1\\ 1\end{pmatrix} \\ &= 21\end{aligned}$$

Fig. 1 illustrates the WFA $A = (Q_A, \alpha_A, \beta_A, (A_\sigma)_{\sigma\in\Sigma})$, where the transitions with weight 0 are omitted.

Definition 5 (DFA)**.** A *DFA* is defined much like in Def. 1, except that 1) the entries of matrices are `tt` and `ff`; 2) we replace the use of $+, \times$ with $\vee, \wedge$, respectively; and 3) we impose *determinacy*, that exactly one entry is `tt` in each row of A_σ, and that only one entry is `tt` in the initial vector α_A.

The definitions of δ_A and f_A in Def. 2–3 adapt to DFAs. For $w \in \Sigma^*$, the configuration vector $\delta_A(w) \in \{\texttt{tt}, \texttt{ff}\}^{Q_A}$ has exactly one `tt`; the state q whose entry is `tt` is called the w-*successor* of A. w is *accepted* by A if $f_A(w) = \texttt{tt}$.

Recurrent Neural Networks Our view of a recurrent neural network (RNN) is almost a black-box. We need only the following two operations: feeding an input word $w \in \Sigma^*$ and observing its output (a real number); and additionally, observing the internal state (a vector) after feeding a word. This allows us to model RNNs in the following abstract way.

Definition 6 (RNN)**.** Let $d \in \mathbb{N}$ be a natural number called a *dimension*. A (real-valued) *RNN* is a triple $R = (\alpha_R, \beta_R, g_R)$, where $\alpha_R \in \mathbb{R}^d$ is an *initial state*, $\beta_R \colon \mathbb{R}^d \to \mathbb{R}$ is an *output function*, and $g_R \colon \mathbb{R}^d \times \Sigma \to \mathbb{R}^d$ is called a *transition function*. The set $\mathbb{R}^d$ is called a *state space*.

Definition 7 (RNN configuration $\delta_R(w)$, output $f_R(w)$)**.** Let R be the RNN in Def. 6. The transition function g_R naturally extends to words as follows: $g_R^* \colon \mathbb{R}^d \times \Sigma^* \to \mathbb{R}^d$, defined inductively by $g_R^*(x, \varepsilon) = x$ and $g_R^*(x, w\sigma) = g_R\big(g_R^*(x, w), \sigma\big)$, where $w \in \Sigma^*$ and $\sigma \in \Sigma$.

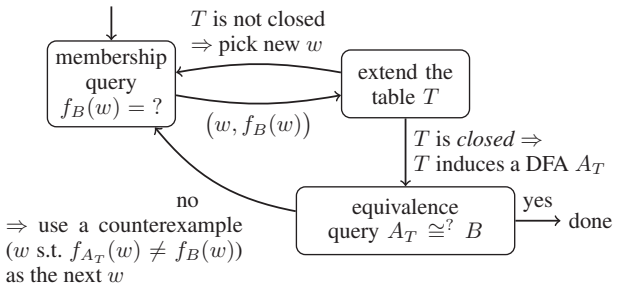

Figure 2: An outline of Angluin's L* algorithm. The target DFA is B; a table T gets gradually extended, yielding a DFA A_T when it is closed. See also Fig. 3a

$\mathcal{A}\backslash\mathcal{T}$	ε	0	1
ε	tt	ff	ff
0	ff	tt	ff
⋮	⋮	⋮	⋮
011	ff	tt	ff

(a) A table T for DFA learning

$\mathcal{A}\backslash\mathcal{T}$	ε	0	1
ε	0.5	0	0.5
0	0	1	0
⋮	⋮	⋮	⋮
001	0.2	0.4	0.4

(b) A table T for WFA learning

Figure 3: Observation tables for L*-style algorithms

The *configuration* $\delta_R(w)$ of the RNN R at a word w is defined by $\delta_R(w) = g_R^*(\alpha_R, w)$. The *output* $f_R(w) \in \mathbb{R}$, of R for the input w, is defined by $f_R(w) = \beta_R\big(\delta_R(w)\big)$.

2.1 Angluin's L* Algorithm

Angluin's L*-algorithm learns a given DFA B by a series of *membership* and *equivalence* queries. We sketch the algorithm; see (Angluin 1987) for details. Its outline is in Fig. 2.

A *membership query* is a black-box observation of the DFA B: it feeds B with a word $w \in \Sigma^*$; and obtains $f_B(w) \in \{\texttt{tt}, \texttt{ff}\}$, i.e., whether w is accepted by B.

The core of the algorithm is to construct the *observation table* T; see Fig. 3a. The table has words as the row and column labels; its entries are either `tt` or `ff`. The row labels are called *access words*; the column labels are *test words*. We let $\mathcal{A}, \mathcal{T}$ stand for the sets of access and test words. The entry of T at row $u \in \mathcal{A}$ and column $v \in \mathcal{T}$ is given by $f_B(uv)$—a value that we can obtain from a suitable membership query.

Therefore we extend a table T by a series of membership queries. We do so until T becomes closed; this is the top loop in Fig. 2. A table T is *closed* if, for any access word $u \in \mathcal{A}$ and $\sigma \in \Sigma$, there is an access word $u' \in \mathcal{A}$ such that

$$f_B(u\sigma\, v) = f_B(u'v) \quad \text{for each test word } v \in \mathcal{T}. \qquad (1)$$

The closedness condition essentially says that the role of the extended word $u\sigma$ is already covered by some existing word $u' \in \mathcal{A}$. The notion of "role" here, formalized in (1), is a restriction of the well-known Myhill–Nerode relation, from all words $v \in \Sigma^*$ to $v \in \mathcal{T}$.

A closed table T induces a DFA A_T (Fig. 2), much like in the Myhill–Nerode theorem. We note that the resulting DFA A_T is necessarily minimized. The DFA A_T undergoes an *equivalence query* that asks if $A_T \cong B$; an equivalence query is answered with a counterexample—i.e., $w \in \Sigma^*$ such that $f_{A_T}(w) \neq f_B(w)$—if $A_T \ncong B$.

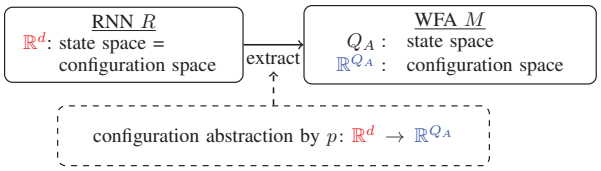

Figure 4: An outline of our WFA extraction

The L* algorithm is a *deterministic* learning algorithm (at least in its original form), unlike many recent learning algorithms that are statistical. The greatest challenge in practical use of the L* algorithm is to answer equivalence queries. When B is a finite automaton that generates a regular language, there is a complete algorithm for deciding the language equivalence $A_T \cong B$. However, if we use a more expressive model in place of a DFA B, checking $A_T \cong B$ becomes a nontrivial task.

2.2 L* Algorithm for WFA Learning

The classic L* algorithm for learning DFAs has seen a *weighted* extension (Balle and Mohri 2015): it learns a WFA B, again via a series of membership and equivalence queries. The overall structure of the WFA learning algorithm stays the same as in Fig. 2; here we highlight major differences.

Firstly, the entries of an observation table T are now real numbers, reflecting the fact that the value $f_B(uv)$ for a WFA B is in $\mathbb{R}$ instead of in $\{\mathtt{tt}, \mathtt{ff}\}$ (see Def. 3). An example of an observation table is given in Fig. 3b.

Secondly, the notion of closedness is adapted to the weighted (i.e., linear-algebraic) setting, as follows. A table T is *closed* if, for any access word $u \in \mathcal{A}$ and $\sigma \in \Sigma$, the vector $\big(\, f_B(u\sigma\, v)\,\big)_{v\in\mathcal{T}} \in \mathbb{R}^{|T|}$ can be expressed as a linear combination of the vectors in $\big\{\, \big(\, f_B(u'v)\,\big)_{v\in\mathcal{T}} \,\big|\, u' \in \mathcal{A} \big\}$. Note that the vector $\big(\, f_B(u'v)\,\big)^\top_{v\in\mathcal{T}}$ in the latter set is precisely the row vector in T at row u'.

For example, the table T in Fig. 3b is obviously closed, since the three row vectors are linearly independent and thus span the whole $\mathbb{R}^3$. The above definition of closedness comes natural in view of Def. 2. For a WFA, a configuration (during its execution) is not a single state, but a *weighted superposition* $x \in \mathbb{R}^{Q_A}$ of states. The closedness condition asserts that the role of $u\sigma$ is covered by a suitable superposition of words $u' \in \mathcal{A}$. The construction of the WFA A_T from a closed table T (see Fig. 2) reflects this intuition. See (Balle and Mohri 2015). We note that the resulting A_T is minimal, much like in §2.1.

In the literature (Balle and Mohri 2015), an observation table T is presented as a so-called *Hankel* matrix. This opens the way to further extensions of the method, such as an approximate learning algorithm via the singular-value decomposition (SVD).

3 WFA Extraction from an RNN

We present our main contribution, namely a procedure that extracts a WFA from a given RNN. After briefly describing its outline (that is the same as Fig. 2), we focus on the greatest challenge of answering equivalence queries.

Algorithm 1 Answering equivalence queries

procedure ANS-EQQ
Input: RNN $R = (\alpha_R, \beta_R, g_R)$, WFA $A = (Q_A, \alpha_A, \beta_A, (A_\sigma)_{\sigma\in\Sigma})$, error tolerance $e > 0$ and concentration threshold $M \in \mathbb{N}$
Output: a counterexample, or `Equivalent`
  Initialize $p\colon \mathbb{R}^d \to \mathbb{R}^{Q_A}$ so that $p(\delta_R(\varepsilon)) = \delta_A(\varepsilon)$
  `queue` $\leftarrow \langle\varepsilon\rangle$; `visited` $\leftarrow \emptyset$
  while `queue` is non-empty **do**
    $h \leftarrow \mathbf{pop}(\mathtt{queue})$
      ▷ Pop the element of the maximum priority
    if $|f_R(h) - f_A(h)| \geq e$ **then**
      return h ▷ return a counterexample
    `result` $\leftarrow$ CONSISTENT?$(h, \mathtt{visited}, p)$
    if `result` $= \mathtt{NG}$ **then**
      learn p by regression, so that $p(\delta_R(h')) = \delta_A(h')$ holds for all $h' \in \mathtt{visited} \cup \{h\}$
    `visited` $\leftarrow$ `visited` $\cup \{\, h \,\}$
    `visited`$' \leftarrow p(\delta_R(\mathtt{visited}))$
    $\#vn \leftarrow |\{\, x \in \mathtt{visited}' \mid x \simeq_A p(\delta_R(h)) \,\}|$
    if $\#vn \leq M$ **then**
      pr $\leftarrow \min_{h'\in\mathtt{visited}\setminus\{h\}} d(p(\delta_R(h)), p(\delta_R(h')))$
        ▷ d is the Euclidean distance
      push $h\sigma$ **to** `queue` with priority pr **for** $\sigma \in \Sigma$
  return `Equivalent`

3.1 Procedure Outline

Our procedure uses the weighted L* algorithm sketched in §2.2. As we discussed in §2.1, the greatest challenge is how to answer equivalence queries; our novel approach is to use *regression* and synthesize what we call a *configuration abstraction function* $p\colon \mathbb{R}^d \to \mathbb{R}^{Q_A}$. See Fig. 4.

The outline of our procedure thus stays the same as in Fig. 2, but we need to take care about noisy outputs from an RNN because they prevent the observation table from being precisely closed (in the sense of §2.2). To resolve this issue, we use a noise-tolerant algorithm (Balle and Mohri 2015) which approximately determines whether the observation table is closed. This approximate algorithm employs SVD and cuts off singular values that are smaller than a threshold called *rank tolerance*. In the choice of a rank tolerance, we face the trade off between accuracy and regularity. If the rank tolerance is large, the WFA learning algorithm tends to ignore the counterexample given by an equivalence query and results in producing an inaccurate WFA. We use a heuristic to decrease the rank tolerance when two or more equivalence queries return the same counterexample. See Appendix A.1 of (Okudono et al. 2019) for details.

3.2 Equivalence Queries for WFAs and RNNs

Algorithm 1 shows our procedure to answer an equivalence query. The procedure ANS-EQQ is the main procedure, and it returns either `Equivalent` or a counterexample word (as

in Fig. 2). It calls the auxiliary procedure CONSISTENT?, which decides if we refine the current configuration abstraction function $p\colon \mathbb{R}^d \to \mathbb{R}^{Q_A}$ in Fig. 4 (Line 10).

Best-First Search for a Counterexample The procedure ANS-EQQ is essentially a best-first search for a *counterexample*, that is, a word $h \in \Sigma^*$ such that the difference of the output values $f_R(h)$ from the RNN and $f_A(h)$ from the WFA is larger than *error tolerance* $e(> 0)$. We first outline ANS-EQQ and then go into the technical detail.

We manage counterexample candidates by the priority queue `queue`, which gives a higher priority to a candidate more likely to be a counterexample. The already investigated words are in the set `visited`. The queue `queue` initially contains only the empty word ε (Line 3).

We search for a counterexample in the main loop starting from Line 4. Let h be a word popped from `queue`, that is, the candidate most likely to be a proper counterexample among the words in the queue. If h is a counterexample, ANS-EQQ returns it (Lines 6–7). Otherwise, after refining the configuration abstraction function p (Lines 8–10), new candidates $h\sigma$, the extension of h with character σ, are pushed to `queue` with their priorities *only if* the neighborhood of h in the state space of the WFA A does not contain sufficiently many already investigated words—i.e., only if it has not been investigated sufficiently (Lines 13–16). This is because, if the neighborhood of h has been investigated sufficiently, we expect that the neighborhoods of the new candidates $h\sigma$ have also been investigated and, therefore, that the words $h\sigma$ do not have to be investigated furthermore. We use p: 1) to decide if the neighborhood of h has been investigated sufficiently (Line 13); and 2) to calculate the priorities of the new candidates (Line 15). Note that we add h to `visited` in Line 11 since it has been investigated there. If all the candidates are not a counterexample, ANS-EQQ returns `Equivalent` (Line 17).

Configuration Abstraction Function p To use p for the above purposes, the property we expect from the configuration abstraction function $p\colon \mathbb{R}^d \to \mathbb{R}^{Q_A}$ is as follows:

$$p(\delta_R(h)) \approx \delta_A(h) \quad \text{for as many } h \in \Sigma^* \text{ as possible.} \quad (2)$$

See Def. 2 and 7 for $\delta_A\colon \Sigma^* \to \mathbb{R}^{Q_A}$ and $\delta_R\colon \Sigma^* \to \mathbb{R}^d$, respectively. To synthesize such a function p, we employ *regression* using the data $\left\{ \left(\delta_R(h'), \delta_A(h')\right) \in \mathbb{R}^d \times \mathbb{R}^{Q_A} \,\middle|\, h \in \texttt{visited} \right\}$. See Line 10. Note that we can use any regression method to learn p.

We refine p during the best-first counterexample search. Specifically, in Line 8, we use the procedure CONSISTENT? to check if the current p—obtained by regression—is consistent with a counterexample candidate h. The consistency means that $p(\delta_R(h))$ and $\delta_A(h)$ are close to each other, which is formalized by the relation $\simeq_A$ defined later. If the check fails (i.e., if CONSISTENT? returns `NG`), we refine p by regression to make p consistent with h (and the already investigated words in `visited`). See Line 10.

Consistency Checking by CONSISTENT? The procedure CONSISTENT? in Line 8 is defined as follows: it returns `NG` if there exists $h' \in$ `visited` such that

$$\delta_A(h') \not\simeq_A p(\delta_R(h')) \text{ and } p(\delta_R(h')) \simeq_A p(\delta_R(h)), \quad (3)$$

and returns `OK` otherwise. The basic idea of CONSISTENT? is to return `NG` if $p(\delta_R(h)) \not\simeq_A \delta_A(h)$ because it means the violation of the desired property (2). However, to reduce the run-time cost of refining p and to prevent learning from outliers, we adopt the alternative approach presented above, which is taken from Weiss, Goldberg, and Yahav (2018a).

The existence of h' satisfying the condition (3) approximates the violation of the property (2) in the following sense. If there is a word h' satisfying the first part of the condition (3), p has to be refined because we find the property (2) violated with h'. The second part of the condition (3) means that h' seems to behave similarly to h according to the configuration abstraction function p. We expect the second part to prevent p from being refined with outliers because the neighborhoods of the words used for refining p must have been investigated twice or more.

Equivalence Relation $\simeq_A$ For a given WFA A, we define the relation $\simeq_A$ in the configuration space $\mathbb{R}^{Q_A}$ by

$$\vec{x} \simeq_A \vec{y} \iff \textstyle\sum_{i=1}^{|Q_A|} \beta_i^2 (x_i - y_i)^2 < \frac{e^2}{|Q_A|}, \quad (4)$$

where β is the final vector of the WFA A. It satisfies the following.

1. If $x \simeq_A y$ holds, the difference of the output values for the configurations x and y of the WFA A is smaller than the error tolerance e, that is, $|(x - y) \cdot \beta| < e$.
2. If the i-th element of the final vector β becomes large, the neighborhood $\{\, y \in \mathbb{R}^{Q_A} \mid x \simeq_A y \,\}$ of x shrinks in the direction of the i-th axis.
3. The neighborhood defined by $\simeq_A$ is an ellipsoid—a reasonable variant of an open ball as a neighborhood.

A Heuristic for Equivalence Checking of a WFA and an RNN Although the best-first search above works well, we introduce an additional heuristic to improve the runtime performance of our algorithm furthermore. The heuristic deems R and A to be equivalent if word h previously popped from `queue` is so long that it is impossible to occur in the training of the RNN. This heuristic is based on the expectation that, when an impossible word is the most likely to be a counterexample, all possible words are unlikely to be a counterexample, and so R and A are considered to be equivalent. This heuristic is adopted immediately after popping h (Line 5), as follows: we suppose that the maximum length L of possible words is given; and, if the length of h is larger than L, ANS-EQQ returns `Equivalent`. We confirm the usefulness of this heuristic in §4 empirically.

Termination of the Procedure Algorithm 1 does not always terminate in a finite amount of time. If the procedure does not find any counterexample at Line 7 and the points $p(\delta_R(\texttt{visited}))$ are so scattered in the configuration space that the value $\#vn$ at Line 13 is always small, words are always pushed to `queue` at Line 16. In that case, the condition to exit the main loop at Line 4 is never satisfied.

3.3 Comparison with Weiss et al., 2018

Our WFA extraction method can be seen as a weighted extension of the L*-based procedure (Weiss, Goldberg, and

Yahav 2018a) to extract a DFA from an RNN. Note that a WFA defines a function of type $\Sigma^* \to \mathbb{R}$ and a DFA defines a function of type $\Sigma^* \to \{\,\mathtt{tt}, \mathtt{ff}\,\}$.

The main technical novelty of the method in (Weiss, Goldberg, and Yahav 2018a) is how to answer equivalence queries. It features the *clustering* of the state space $\mathbb{R}^d$ of an RNN into finitely many clusters, using support-vector machines (SVMs). Each cluster of $\mathbb{R}^d$ is associated with a state $q \in Q_A$ of the DFA.

Our theoretical observation is that such clustering amounts to giving a function $p\colon \mathbb{R}^d \to Q_A$. Moreover, for a DFA, Q_A is the configuration space (as well as the state space, see Def. 5). Therefore, our WFA extraction method can be seen as an extension of the DFA extraction procedure in (Weiss, Goldberg, and Yahav 2018a).

4 Experiments

We conducted experiments to evaluate the utility of our regression-based WFA extraction method. Specifically, we pose the following questions.

RQ1 Do the WFAs extracted by our algorithm approximate the original RNNs accurately?

RQ2 Does our algorithm work well with RNNs whose expressivity is beyond WFAs?

RQ3 Do the extracted WFAs run more efficiently, specifically in inference time, than given RNNs?

For **RQ1**, we compared the performance with a baseline algorithm (a straightforward adaptation of Balle and Mohri (2015)'s L*algorithm). Here we focused on "automata-like" RNNs, that is, those RNNs trained from an original WFA $A^\bullet$. For **RQ2**, we used an RNN that exhibits "context-free" behaviors.

Experimental Setting We implemented our method in Python. We write $\mathbf{RGR}(n)$ for the algorithm where the concentration threshold M in ANS-EQQ is set to n; other parameters are fixed as follows: error tolerance $e = 0.05$ and heuristic parameter $L = 20$. We adopt the Gaussian process regression (GPR) provided by scikit-learn as a configuration abstraction function p (we also tried the kernel ridge regression but GPR worked empirically better). Throughout the experiments, our RNNs are 2-layer LSTM networks with dimension size 50, implemented by TensorFlow. The experiments were on a g3s.xlarge instance on Amazon Web Service (May 2019), with a NVIDIA Tesla M60 GPU, 4 vCPUs of Xeon E5-2686 v4 (Broadwell), and 8GiB memory.

4.1 RQ1: Extraction from RNNs Modeling WFAs

This experiment examines how well our algorithm work for RNNs modeling WFAs. To do so, we first train RNNs using randomly generated WFAs; we call those WFAs the *origins*. Then, we evaluate our algorithm compared with a baseline from two points: accuracy of the extracted WFAs against the trained RNNs and running times of the algorithms. We report the results after presenting the details of the baseline, how to train RNNs, and how to evaluate the two algorithms.

The Baseline Algorithm: $\mathbf{BFS}(n)$ As our extraction algorithm, the baseline algorithm $\mathbf{BFS}(n)$, which is parameterized over an integer n, is a straightforward adaptation of Balle and Mohri (2015)'s L*algorithm. The difference is that equivalence queries in the baseline are implemented in breath-first search, as follows. Let R be a given RNN and A be a WFA being constructed. For each equivalence query, the baseline searches for a word w such that $|f_A(w) - f_R(w)| > e$ (where $e = 0.05$), in the breadth-first manner. If such a word w is found it is returned as a counterexample. The search is restricted to the first $i + n$ words, where i is the index of the counterexample word found in the previous equivalence query. If no counterexample w is found within this search space, the baseline algorithm deems A to be equivalent to R. Obviously, if n is larger, more counterexample candidates are investigated.

Target RNNs R trained from WFAs $A^\bullet$ Table 1 reports the accuracy of the extracted WFAs A, where the target RNNs R are obtained from original WFAs $A^\bullet$ in the following manner. Given an alphabet Σ of a designated size (the leftmost column), we first generated a WFA $A^\bullet$ such that 1) the state-space size $|Q_{A^\bullet}|$ is as designated (the leftmost column), and 2) the initial vector $\alpha^\bullet_A$, the final vector $\beta^\bullet_A$, and the transition matrix $A^\bullet_\sigma$ are randomly chosen (with normalization so that its outputs are in $[0, 1]$). Then we constructed a dataset T by sampling 9000 words w such that $w \in \Sigma^*$ and $|w| \leq 20$; this T was used to train an RNN R, on the set of input-output pairs of $w \in T$ and $f_{A^\bullet}(w)$ for 10 epochs.

A simple way to sample words $w \in T$ is by the uniform distribution. With T covering the input space of the WFA $A^\bullet$ uniformly, we expect the resulting RNN R to inherit properties of $A^\bullet$ well. The top table in Table 1 reports the results in this "more WFA-like" setting.

However, in many applications, the input domain of the data used for training RNNs are nonuniform, sometimes even excluding some specific patterns. To evaluate our method in such realistic settings, we conducted another set of experiments whose results are in the bottom of Table 1. Specifically, for training R from $A^\bullet$, we used a dataset T that only contains those words $\sigma_1\sigma_2 \dots \sigma_n$ which satisfy the following condition: if $\sigma_i = \sigma_j$ $(i < j)$, then $\sigma_i = \sigma_{i+1} = \dots = \sigma_{j-1} = \sigma_j$. For example, for $\Sigma = \{a, b, c\}$, $aabccc$ and $baaccc$ may be in T, but $aaba$ may not.

Evaluation In order to evaluate accuracy, we calculated the mean square error (MSE) of the extracted WFA A against the RNN R, using a dataset V of words sampled from an appropriate distribution, namely the one used in training the RNN R from $A^\bullet$. The dataset V is sampled so that it does not overlap with the training dataset T for R.

Results and Discussions In the experiments in Table 1, we considered 8 configurations for generating the original WFA $A^\bullet$ (the leftmost column). The unit of MSEs are -4—given also that the outputs of the original WFAs $A^\bullet$ are normalized to $[0, 1]$, we can say that the MSEs are small enough.

In the top table in Table 1 (the "more WFA-like" setting), $\mathbf{BFS}(5000)$ and $\mathbf{RGR}(5)$ achieved the first- and second-best performance in terms of accuracy, respectively (see the "Total" row). More generally, we can find the trend that, as an extraction runs longer, it performs better. We conjecture its reason as follows. Recall that all the RNNs are trained on words sampled from the uniform distribution. This means

Table 1: Experiment results, where we extracted a WFA A from an RNN R that is trained to mimic the original WFA $A^\bullet$. In each cell "n/m", "n" denotes the average of MSEs between A and R (the unit is 10^{-4}), taken over five random WFAs $A^\bullet$ of the designated alphabet size $|\Sigma|$ and the state-space size $|Q_{A^\bullet}|$. "m" denotes the average running time (the unit is second). The "Total" row describes the average over all the experiment settings. The highlighted cell designates the best performer in terms of errors. Timeout was set at 10,000 sec. $\mathbf{RGR}(2\text{–}5)$ are our regression-based methods; $\mathbf{BFS}(500\text{–}5000)$ are the baseline. $\mathbf{BFS}(5000)$ is added to compare the accuracy when the running time is much longer.

Results for "more WFA-like" RNNs R, i.e., those RNNs R which are trained from WFAs $A^\bullet$ using a uniformly sampled data set $T \subseteq \Sigma^*$

$(\lvert\Sigma\rvert, \lvert Q_{A^\bullet}\rvert)$	$\mathbf{RGR}(2)$	$\mathbf{RGR}(5)$	$\mathbf{BFS}(500)$	$\mathbf{BFS}(1000)$	$\mathbf{BFS}(2000)$	$\mathbf{BFS}(3000)$	$\mathbf{BFS}(5000)$
$(4, 10)$	2.17 / 286	2.39 / 338	26.8 / 165	9.77 / 279	4.36 / 545	4.07 / 716	2.33 / 1390
$(6, 10)$	2.45 / 1787	2.54 / 1302	6.99 / 386	4.48 / 641	4.08 / 1218	3.15 / 1410	2.28 / 2480
$(10, 10)$	4.68 / 7462	4.46 / 5311	22.5 / 928	11.9 / 1562	5.90 / 3521	4.55 / 3638	3.55 / 5571
$(10, 15)$	5.62 / 8941	5.78 / 8564	21.2 / 2155	10.6 / 4750	7.87 / 5692	5.71 / 7344	5.27 / 7612
$(10, 20)$	3.70 / 7610	3.79 / 7799	6.24 / 2465	10.1 / 2188	6.13 / 3106	3.70 / 5729	3.63 / 7473
$(15, 10)$	7.34 / 9569	5.52 / 10000	13.5 / 3227	8.01 / 6765	6.07 / 7916	5.98 / 8911	6.17 / 8979
$(15, 15)$	8.44 / 10000	5.58 / 9981	16.3 / 2675	9.24 / 4850	7.28 / 5135	9.88 / 7204	6.44 / 8425
$(15, 20)$	9.16 / 7344	5.15 / 7857	13.7 / 2224	7.26 / 3823	6.60 / 5744	4.96 / 5674	4.01 / 9464
Total	5.45 / 6625	4.40 / 6394	15.9 / 1778	8.92 / 3107	6.04 / 4110	5.25 / 5078	4.21 / 6549

Results for "realistic" RNNs R with *nonuniform input domains*, i.e., those R which are trained from WFAs $A^\bullet$ using a data set $T \subseteq \Sigma^*$ in which some specific patterns are prohibited

$(\lvert\Sigma\rvert, \lvert Q_{A^\bullet}\rvert)$	$\mathbf{RGR}(2)$	$\mathbf{RGR}(5)$	$\mathbf{BFS}(500)$	$\mathbf{BFS}(1000)$	$\mathbf{BFS}(2000)$	$\mathbf{BFS}(3000)$	$\mathbf{BFS}(5000)$
$(4, 10)$	7.73 / 696	7.07 / 1135	15.0 / 199	7.96 / 424	6.62 / 650	6.61 / 762	9.06 / 1693
$(6, 10)$	4.92 / 1442	7.43 / 1247	1.46 / 552	6.95 / 660	5.90 / 1217	8.78 / 1557	3.54 / 2237
$(10, 10)$	5.02 / 5536	4.28 / 5951	7.70 / 1117	11.0 / 1738	4.77 / 2635	3.52 / 3926	4.52 / 4777
$(10, 15)$	7.15 / 6977	4.35 / 8315	19.4 / 1552	13.8 / 3271	16.8 / 3209	8.57 / 5293	5.08 / 6522
$(10, 20)$	6.98 / 4697	8.06 / 6704	18.6 / 1465	11.8 / 2046	12.7 / 2851	9.03 / 4259	8.01 / 4856
$(15, 10)$	5.97 / 8747	6.77 / 8882	23.3 / 2359	11.2 / 4668	9.88 / 6186	6.24 / 7557	6.02 / 8245
$(15, 15)$	5.78 / 8325	8.71 / 7546	16.6 / 2874	7.31 / 4380	9.92 / 6015	9.89 / 7110	6.40 / 8358
$(15, 20)$	4.60 / 7652	8.56 / 8334	36.9 / 1893	23.7 / 3069	12.8 / 3987	12.0 / 5262	8.38 / 6441
Total	6.02 / 5510	6.90 / 6015	19.0 / 1502	11.7 / 2532	9.92 / 3344	8.08 / 4466	6.38 / 5391

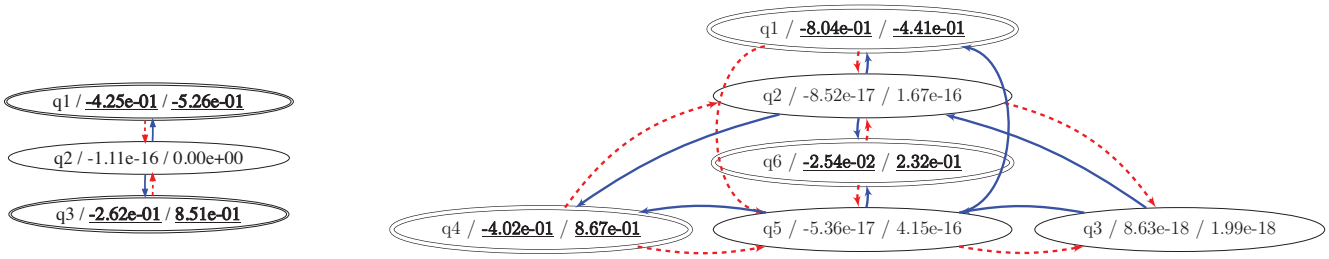

Figure 5: The WFAs extracted by $\mathbf{RGR}(5)$ (left) and $\mathbf{RGR}(15)$ (right). In a state label "$q/m/n$", q is the state name, m is the initial value and n is the final value. Bigger values are underlined; other values are negligibly small. The dotted and solid edges are labelled with "(" and ")", respectively; the edges with labels $0, 1, \ldots, 9$ are omitted. The edge weights are omitted for simplicity, too; the weight threshold for showing transitions is 0.01. Appendix B.2 of (Okudono et al. 2019).

that all words would be somewhat informative to approximate the RNNs. As a result, the performance is more influenced by the amount of counterexamples—i.e., how long time extraction takes—than on their "qualities."

The exception of this trend is $\mathbf{RGR}(2)$, which took a longer time but performed worse than $\mathbf{BFS}(3000)$, $\mathbf{BFS}(5000)$, and $\mathbf{RGR}(5)$. In particular, $\mathbf{RGR}(2)$ performed well for smaller alphabets ($|\Sigma| \in \{4, 6, 10\}$) but not so when $|\Sigma| = 15$. The role of the parameter M in $\mathbf{RGR}(M)$ (i.e., in Algorithm 1) is a threshold to control how many words configuration regions of a WFA are investigated with. Thus, we conjecture that the use of too small M limits the input space to be investigated excessively, which is more critical as the input space is larger, eventually biasing the counterexamples h (in Algorithm 1), though the RNNs are trained on the uniform distribution, and making refinement of WFAs less effective.

In the bottom table in Table 1 (the "realistic" setting), $\mathbf{RGR}(2)$ performs significantly better than the other (and the best among all the procedures) in terms of accuracy. This is the case even for a large alphabet ($|\Sigma| = 15$). This indicates that, in the cases that an RNN is trained with a nonuniform dataset, making the investigated input space larger by big M could even degrade the accuracy performance. A possible reason for this degradation is as follows. Some words (such as aba) are prohibited in the sample set T, and the behaviors of the RNN R for those prohibited words are unexpected. Therefore, those prohibited words should not be useful to refine a WFA. The use of small M could prevent such meaningless (or even harmful) counterexamples h from being investigated. This discussion raises another question: how can we find an optimal M? We leave it as future work.

Let us briefly discuss the sizes of the extracted WFAs. The general trend is that the extracted WFAs A have a few times greater number of states than the original WFAs $A^\bullet$ used in training R. For example, in the setting of the top table in Table 1, for $|\Sigma| = 15$ and $|Q_{A^\bullet}| = 20$, the average number of the states of the extracted A was 38.2.

4.2 RQ2: Expressivity beyond WFAs

We conducted experiments to examine how well our method works for RNNs modeling languages that cannot be expressed by any WFA. Specifically, we used an RNN that models the following function $\texttt{wparen}\colon \Sigma^* \to [0,1]\colon \Sigma = \{(,),0,1,...,9\}$, $\texttt{wparen}(w) = 1 - (1/2)^N$ if all the parentheses in w are balanced (here N is the depth of the deepest balanced parentheses in w); and $\texttt{wparen}(w) = 0$ otherwise. This wparen is a weighted variant of a (non-regular) language of balanced parentheses. For instance, $\texttt{wparen}(\text{“((3)(7))))”}) = 0$, $\texttt{wparen}(\text{“((3)(7))”}) = 1 - (1/2)^2 = 3/4$, and $\texttt{wparen}(\text{“(a)(b)(c)”}) = 1/2$.

We trained an RNN R as follows. We generated datasets T_{good} and T_{bad}, and trained an RNN R on the set of input-output pairs of $w \in T_{\mathrm{good}} \cup T_{\mathrm{bad}}$ and $\texttt{wparen}(w)$. The dataset T_{good} consists of randomly generated words where all the parentheses are balanced; T_{bad} is constructed similarly, except that we apply suitable mutation to each word, which most likely makes the parentheses unbalanced. See Appendix B.1 of (Okudono et al. 2019) for details.

Fig. 5 shows the WFAs extracted from R. Remarkable observations here are as follows.

- The shapes of the WFAs—obtained by ignoring clearly negligible weights—give rise to NFAs that recognize balanced parentheses up-to a certain depth.
- As the parameter M in $\mathbf{RGR}(M)$ grows, the recognizable depth bound grows: depth one with $\mathbf{RGR}(5)$; and depth two with $\mathbf{RGR}(15)$.

We believe these observations demonstrate important features, as well as limitations, of our method. Overall, the extracted WFAs expose interpretable structures hidden in an RNN: the NFA structures in Fig. 5 are human-interpretable (they are easily seen to encode bounded balancedness) and machine-processable (such as determinization and minimization). It is also suggested that the parameter M gives us flexibility in the trade-off of extraction cost and accuracy. At the same time, we can only obtain a truncated and regularized version of the RNN structure—this is an inevitable limitation as long as we use the formalism of WFAs.

We also note that, in each of the two extracted WFAs, the transition matrices A_σ are similar for all $\sigma \in \{0,1,\dots,9\}$ (the entries at the same position have the same order). This is as expected, too, since the function wparen does not distinguish the characters $0,1,\dots,9$.

4.3 RQ3: Accelerating Inference Time

We conducted experiments about inference time, comparing the original RNNs R and the WFAs A that we extracted from R. We used the same RNNs R and WFAs A as in §4.1, where the latter are extracted using $\mathbf{RGR}(2\text{–}5)$ and $\mathbf{BFS}(500\text{–}5000)$. We note that the inference of RNNs utilizes GPUs while that of WFAs is solely done by CPUs.

We observed that the inference time of the extracted WFAs A was about 1,300 times faster than the target RNNs R, taking the average over different settings (Appendix B.3 of (Okudono et al. 2019)). This demonstrates the potential use of the extracted WFAs as a computationally cheaper surrogate for RNNs. We attribute the acceleration to the following: 1) WFAs use only linear computation while RNNs involve nonlinear ones; and 2) overall, extracted WFAs are smaller in size. Provided that the accuracy of extracted WFAs can be high (as we observed in §4.1), we believe the replacement of RNNs by WFAs is a viable option in some application scenarios.

5 Conclusions and Future Work

We proposed a method that extracts a WFA from an RNN, focusing on RNNs that take a word $w \in \Sigma^*$ and return a real value. We used regression to investigate and abstract the internal states of RNNs. We experimentally evaluated our method, comparing its performance with a baseline whose equivalence queries are based on simple breadth-first search.

One future work is a detailed comparison with other methods for model compression. Another future work is to use machine learning methods to find a counterexample in the equivalence query, such as reinforcement learning (Mnih et al. 2015) adversarial attacks (Papernot et al. 2016), and acquisition functions of GPR. Finally, we need a means to optimize parameter M of our method for a specific problem. It may also be helpful to extend our method so that the investigated words can be restricted to a fixed language $L \subset \Sigma^*$; If L identifies the input space of the training dataset for RNNs, we could avoid investigating the input space on which the RNNs are not trained, and therefore we could seek only “meaningful” counterexamples even in using large M.

6 Acknowledgments

Thanks are due to Mahito Sugiyama and the anonymous reviewers of AAAI for a lot of useful comments. This work is partially supported by JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JSPS KAKENHI Grant Numbers JP15KT0012, JP18J22498, JP19K20247, JP19K22842, and JST-Mirai Program Grant Number JPMJMI18BA, Japan.

References

Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Inf. Comput.* 75(2):87–106.

Ayache, S.; Eyraud, R.; and Goudian, N. 2018. Explaining black boxes on sequential data using weighted automata. In Unold, O.; Dyrka, W.; and Wieczorek, W., eds., *Proc. ICGI 2018*, volume 93 of *Proceedings of Machine Learning Research*, 81–103. PMLR.

Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. The MIT Press.

Balle, B., and Mohri, M. 2015. Learning weighted automata. In Maletti, A., ed., *Proc. CAI 2015*, volume 9270 of *Lecture Notes in Computer Science*, 1–21. Springer.

Balle, B.; Gourdeau, P.; and Panangaden, P. 2017. Bisimulation metrics for weighted automata. In Chatzigiannakis, I.; Indyk, P.; Kuhn, F.; and Muscholl, A., eds., *Proc. ICALP 2017*, volume 80 of *LIPIcs*, 103:1–103:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Bramer, M. 2013. *Ensemble Classification*. London: Springer London. 209–220.

Bucila, C.; Caruana, R.; and Niculescu-Mizil, A. 2006. Model compression. In *Proc. KDD 2006*, 535–541.

Chaudhuri, A. 2019. *Visual and Text Sentiment Analysis through Hierarchical Deep Learning Networks*. Springer Briefs in Computer Science. Springer.

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

Droste, M.; Kuich, W.; and Vogler, H. 2009. *Handbook of Weighted Automata*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg.

Du, X.; Xie, X.; Li, Y.; Ma, L.; Liu, Y.; and Zhao, J. 2019. Deepstellar: model-based quantitative analysis of stateful deep learning systems. In Dumas, M.; Pfahl, D.; Apel, S.; and Russo, A., eds., *Proc. ESEC/FSE 2019*, 477–487. ACM.

Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; and Narayanan, P. 2015. Deep learning with limited numerical precision. In *Proc. ICML 2015*, 1737–1746.

Han, S.; Pool, J.; Tran, J.; and Dally, W. J. 2015. Learning both weights and connections for efficient neural network. In *Proc. NIPS 2015*, 1135–1143.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

Lv, S.; Wang, J.; Yang, Y.; and Liu, J. 2018. Intrusion prediction with system-call sequence-to-sequence model. *IEEE Access* 6:71413–71421.

Michalenko, J. J.; Shah, A.; Verma, A.; Baraniuk, R. G.; Chaudhuri, S.; and Patel, A. B. 2019. Representing formal languages: A comparison between finite automata and recurrent neural networks. In *Proc. ICLR 2019*. OpenReview.net.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Okudono, T.; Waga, M.; Sekiyama, T.; and Hasuo, I. 2019. Weighted automata extraction from recurrent neural networks via regression on state spaces. *CoRR* abs/1904.02931.

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

Papernot, N.; McDaniel, P. D.; Swami, A.; and Harang, R. E. 2016. Crafting adversarial input sequences for recurrent neural networks. In Brand, J.; Valenti, M. C.; Akinpelu, A.; Doshi, B. T.; and Gorsic, B. L., eds., *Proc. MILCOM 2016*, 49–54. IEEE.

Rabusseau, G.; Li, T.; and Precup, D. 2019. Connecting weighted automata and recurrent neural networks through spectral learning. In Chaudhuri, K., and Sugiyama, M., eds., *Proc. AISTATS 2019*, volume 89 of *Proceedings of Machine Learning Research*, 1630–1639. PMLR.

Schwartz, R.; Thomson, S.; and Smith, N. A. 2018. Bridging CNNs, RNNs, and weighted finite-state machines. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 295–305. Melbourne, Australia: Association for Computational Linguistics.

Shen, Z.; Bao, W.; and Huang, D.-S. 2018. Recurrent neural network for predicting transcription factor binding sites. *Scientific reports* 8(1):15270.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Proc. NIPS 2014*, 3104–3112.

Wang, C., and Niepert, M. 2019. State-regularized recurrent neural networks. In Chaudhuri, K., and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, 6596–6606. PMLR.

Weiss, G.; Goldberg, Y.; and Yahav, E. 2018a. Extracting automata from recurrent neural networks using queries and counterexamples. In Dy, J. G., and Krause, A., eds., *Proc. ICML 2018*, volume 80 of *JMLR Workshop and Conference Proceedings*, 5244–5253. JMLR.org.

Weiss, G.; Goldberg, Y.; and Yahav, E. 2018b. On the practical computational power of finite precision rnns for language recognition. In Gurevych, I., and Miyao, Y., eds., *Proc. ACL 2018, Volume 2: Short Papers*, 740–745. Association for Computational Linguistics.

Xiao, X.; Zhang, S.; Mercaldo, F.; Hu, G.; and Sangaiah, A. K. 2019. Android malware detection based on system call sequences and LSTM. *Multimedia Tools Appl.* 78(4):3979–3999.

Yarowsky, D. 1995. Unsupervised word sense disambiguation rivaling supervised methods. In *Proc. ACL 1995*, 189–196.

Zweig, G.; Yu, C.; Droppo, J.; and Stolcke, A. 2017. Advances in all-neural speech recognition. In *Proc. ICASSP 2017*, 4805–4809. IEEE.