Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples

ICML, 2018
Gail Weiss; Yoav Goldberg; Eran Yahav

LIN 629, Nov 1, 2022
Zhengxiang (Jack) Wang

Quick Facts

- **Versions**:
  - arXiv Version (with Supplementary Material): https://arxiv.org/abs/1711.09576
  - Published Version: http://proceedings.mlr.press/v80/weiss18a/weiss18a.pdf

- **Source Code**: https://github.com/tech-srl/lstar_extraction
- ***L**** **Extraction Implementation**:
  - Authors Version: download and run locally or use a non-SBU Google account
  - Modified Version: run directly online

Overview the paper with code

- Abstract
- Key concepts
- Extracting DFA from RNNs using *L**
- Experimental results
- Advantages and limitations
- Contributions of the paper revisited

Abstract

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

Abstract

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

Key concepts: Deterministic finite automaton (DFA); Recurrent Neural Network (RNN); exact learning; abstraction; Angluin's *L** algorithm

Abstract

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

Key Contribution: The paper presents an algorithm/technique that is novel, efficient, and robust in extracting accurate deterministic finite automata from trained RNNs.

Key Concepts

- Deterministic finite automaton (DFA)
- Recurrent Neural Networks (RNNs)
- Exact learning
- Angluin’s *L** algorithm
- Abstraction

DFA: Notations

- **Components**:
  - Σ: alphabet
  - Q: the set of automaton states
  - q_0 : the initial state, $q_0 \in Q$
  - F: accepting states, $F \subseteq Q$
  - δ: transition function $\delta : Q \times \Sigma \rightarrow Q$
  - Classification function $f_A : Q \rightarrow \{Acc, Rej\}$

- **Recursive state transitions to a sequence**: $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$
  - Stay at the current state: $\hat{\delta}(q, \varepsilon) = q$ for every $q \in Q$
  - Transition to the next state: $\hat{\delta}(q, w \cdot \sigma) = \delta(\hat{\delta}(q, w), \sigma)$ for every $w \in \Sigma^*$ and $\sigma \in \Sigma$

DFA: Examples

- **Language 1**: a^*, $\Sigma := \{a\}$

DFA: Examples

- **Language 1**: *a**, Σ := {*a*}

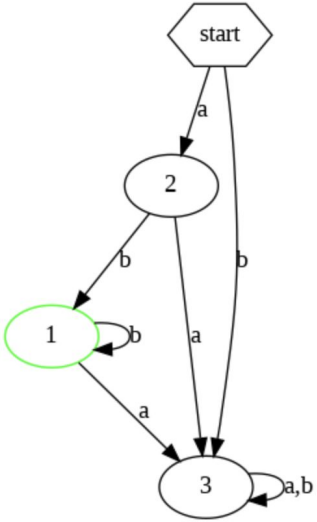
***Flowchart drawn using the authors' L* Extraction Implementation Code*. "start" denotes initial state.**

DFA: Examples

- **Language 2**: ab^+, $\Sigma := \{a, b\}$

DFA: Examples

- **Language 2**: ab^+, Σ := {*a, b*}



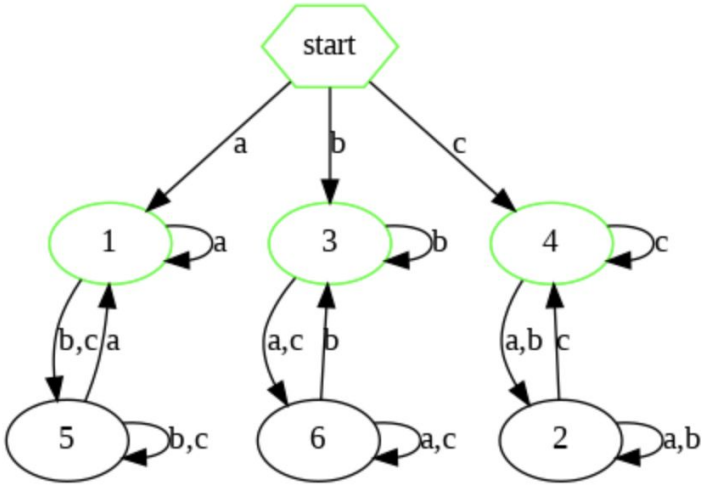
Flowchart drawn using the authors' L Extraction Implementation Code. "start" denotes initial state.*

DFA: Examples

- **Language 3**: initial char must match with final char, Σ := {*a, b, c*}

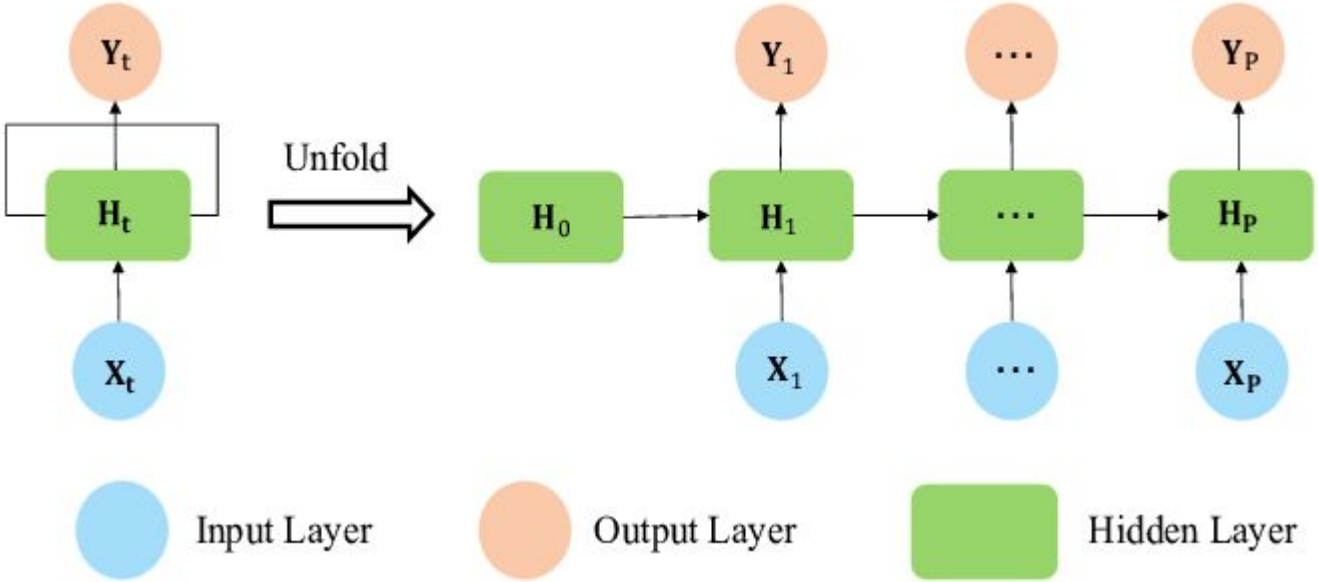
DFA: Examples

- **Language 3**: initial char must match with final char, Σ := {*a, b, c*}



Flowchart drawn using the authors' L Extraction Implementation Code. "start" denotes initial state.*

RNNs: Basic Architecture



Binary RNN-acceptor: Notations

- **Components**:
  - Σ: alphabet
  - S_R: the set of state spaces (for hidden spaces)
  - $h_{0,R}$: the initial state, $h_{0,R} \in S_R$
  - F: accepting states, $F \subseteq S_R$ (given by Jack)
  - g_R: transition function $g_R : S_R \times \Sigma \to S_R$
  - Classification function $f_R : S_R \to \{Acc, Rej\}$

- **Recursive state transitions to a sequence**: $\hat{g_R} : S_R \times \Sigma^* \to S_R$
  - Stay at the current state: $\hat{g_R}(h, \varepsilon) = h$, for every $h \in S_R$
  - Transition to the next state: $\hat{g_R}(h, w \cdot \sigma) = g_R(\hat{g_R}(h, w), \sigma)$ for every $w \in \Sigma^*$ and $\sigma \in \Sigma$

DFA and RNN

- **Additional shorthand** (recursive application of state transitions omitted for convenience):
  - Classification function by DFA: $f_A(w) \longrightarrow \{Acc, Rej\}$
  - Classification function by RNN: $f_R(w) \longrightarrow \{Acc, Rej\}$

DFA: Notations

- **Components**:
  - Σ: alphabet
  - Q: the set of automaton states
  - q_0 : the initial state, $q_0 \in Q$
  - F: accepting states, $F \subseteq Q$
  - δ: transition function $\delta : Q \times \Sigma \longrightarrow Q$
  - Classification function $f_A : Q \longrightarrow \{Acc, Rej\}$

- **Recursive state transitions to a sequence**: $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$
  - Stay at the current state: $\hat{\delta}(q, \varepsilon) = q$, for every $q \in Q$
  - Transition to the next state: $\hat{\delta}(q, w \cdot \sigma) = \delta(\hat{\delta}(q, w), \sigma)$ for every $w \in \Sigma^*$ and $\sigma \in \Sigma$

Binary RNN-acceptor: Notations

- **Components**:
  - Σ: alphabet
  - S_R: the set of state spaces (for hidden spaces)
  - $h_{0,R}$: the initial state, $h_{0,R} \in S_R$
  - F: accepting states, $F \subseteq S_R$ (given by Jack)
  - g_R: transition function $g_R : S_R \times \Sigma \longrightarrow S_R$
  - Classification function $f_R : S_R \longrightarrow \{Acc, Rej\}$

- **Recursive state transitions to a sequence**: $\hat{g_R} : S_R \times \Sigma^* \rightarrow S_R$
  - Stay at the current state: $\hat{g_R}(h, \varepsilon) = h$, for every $h \in S_R$
  - Transition to the next state: $\hat{g_R}(h, w \cdot \sigma) = g_R(\hat{g_R}(h, w), \sigma)$ for every $w \in \Sigma^*$ and $\sigma \in \Sigma$

Exact learning

- In the field of exact learning, concepts (sets of instances) can be learned precisely from *a minimally adequate teacher* — an oracle capable of answering two query types (Goldman & Kearns, 1995):
  - **membership queries**: label a given instance
  - **equivalence queries**: state whether a given hypothesis (set of instances) is equal to the concept held by the teacher. If not, return an instance on which the hypothesis and the concept disagree (a counterexample).

- My takeaways:
  - This is learning from both positive and negative examples.
  - Equivalence query is easy, but being a minimally adequate teacher is hard!

Angluin's *L** algorithm

- The *L** algorithm (Angluin, 1987) is an exact learning algorithm for learning a DFA from a *minimally adequate teacher* that can answer *membership queries* and *equivalence queries* for some regular language *L*.
- *L** algorithm always proposes a minimal DFA in equivalence queries (Jack's takeaway: because the teacher is good!).
- *L** algorithm is treated as a black box in the study, but an informal description about the algorithm is provided in Appendix A in the arXiv version.

Abstraction

- Given a neural network R with state space S and alphabet Σ, and a partitioning function p: $S \to \mathbb{N}$, Omlin and Giles (1996) presented a method for extracting a DFA for which every state is a partition from p, and the state transitions and classifications are defined by a single sample from each partition.
- The method is effectively a BFS exploration of the partitions defined by p, beginning with $p(h_0)$, where h_0 is the network's initial state, and continuing according to the network's transition function g_R.
- We denote by $A^{R,p}$ the DFA extracted by this method from a network R and partitioning p, and denote all its related sets and functions by subscript R, p.

- **Why doing this**: abstraction is finite, making search for counterexample much faster or even possible!

Extraction: Learning DFA from RNNs using *L**

- **For membership queries**, $f_R(w) \rightarrow \{Acc, Rej\}$.
- **For equivalence queries**:
  - Alphabet and two labeled examples provided to initialize the *L** DFA *A* and the $A^{R,p}$
  - While disagreement between *L** DFA *A* and the $A^{R,p}$, and not timeout
    - Check RNNs for ground truth
    - If $A^{R,p}$ is wrong on the classification, refine $A^{R,p}$
    - Otherwise, return the disagreement as counterexample to *L** DFA *A*
  - Return the last *L** DFA *A*

- ***L**** **Extraction Implementation (toy example)**: Google Colab Notebook

Important notes

- *L** DFA *A* only learns from queries/counterexamples classified by RNNs:
  - If RNNs misclassify some queries/counterexamples, *L** DFA *A* learns them anyway
  - If RNNs classify all queries/counterexamples correctly, *L** DFA *A* may end up learning the true grammar, whereas RNNs do not (e.g., for those simple grammars)
  - If *L** DFA *A* sees the queries/counterexamples needed to learn the true grammar, but is also slightly misguided by some misclassification by RNNs, *L** DFA *A* may still outperform RNNs in recognizing the target language

- The extraction is just an **approximation, not equivalence**:
  - No guarantee that the extracted *L** DFA *A* accept exactly the same language as the RNNs do

Experiments on Tomita Languages

- RNNs trained to nearly 100% accuracy on both train and development sets
- Extracted DFA accuracy scores (see paper): reported on the basis of RNNs training sets and additional randomly generated samples
- Tomita languages: [implementation in Python]

[7]The Tomita grammars are the following 7 languages over the alphabet $\{0, 1\}$: [1] 1^*, [2] $(10)^*$, [3] the complement of $((0|1)^*0)^*1(11)^*(0(0|1)^*1)^*0(00)^*(1(0|1)^*)^*$, [4] all words w not containing 000, [5] all w for which $\#_0(w)$ and $\#_1(w)$ are even (where $\#_a(w)$ is the number of a's in w), [6] all w for which $(\#_0(w) - \#_1(w)) \equiv_3 0$, and [7] $0^*1^*0^*1^*$.

Adversarial inputs

- The paper reports adversarial inputs for the nearly “perfect” RNNs found from the equivalence queries (counterexamples) during extraction
- Adversarial inputs can also be found pre/post-extraction (check the last section) using a less efficient method, i.e., random sampling

Table 4. Counterexamples generated during extraction from an LSTM email network with 100% train and test accuracy. Examples of the network deviating from its target language are shown in bold.

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

Advantages and limitations

- Compared to other approaches (i.e., a-priori Quantization, random sampling, k-means clustering), it is faster, more efficient, or more accurate

- Extraction can be slow and cumbersome affected by the complexity of RNNs

> **Limitations** Due to L*'s polynomial complexity and intolerance to noise, for networks with complicated behavior, extraction becomes extremely slow and returns large DFAs. Whenever applied to an RNN that has failed to generalize properly to its target language, our method soon finds several adversarial inputs, builds a large DFA, and times out while refining it.[12]

Contributions of the paper

- The paper is the first attempt to apply exact learning to a given RNN
- The paper presents an algorithm/technique that is novel, efficient, and robust in extracting accurate deterministic finite automata from trained RNNs.
- The method is guaranteed to never extract a DFA more complicated than the language of the RNN being considered.
- The method is able to find adversarial inputs for RNNs during the extraction process efficiently (how efficient?) even if the RNNS were trained to 100% train and development set accuracy.