On-the-fly Black-Box Probably Approximately Correct Checking of Recurrent Neural Networks

Franz Mayr, Ramiro Visca, Sergio Yovine

On-the-fly Black-Box Probably Approximately Correct Checking of Recurrent Neural Networks

Franz Mayr[1], Ramiro Visca[2], and Sergio Yovine[3]

[1] Universidad ORT Uruguay, mayr@ort.edu.uy
[2] Universidad ORT Uruguay, visca@ort.edu.uy
[3] Universidad ORT Uruguay, yovine@ort.edu.uy

Abstract. We propose a procedure for checking properties of recurrent neural networks used for language modeling and sequence classification. Our approach is a case of black-box checking based on learning a probably approximately correct, regular approximation of the intersection of the language of the black-box (the network) with the complement of the property to be checked, without explicitly building individual representations of them. When the algorithm returns an empty language, there is a proven upper bound on the probability of the network not verifying the requirement. When the returned language is nonempty, it is certain the network does not satisfy the property. In this case, an explicit and interpretable characterization of the error is output together with sequences of the network truly violating the property. Besides, our approach does not require resorting to an external decision procedure for verification nor fixing a specific property specification formalism.

1 Introduction

Today, deep recurrent neural networks (RNN) are used for sequence modeling and classification in order to accomplish a variety of safety- and security-critical tasks in a number of application areas, such as autonomous driving [18, 34], intrusion detection [17, 46], malware detection [29, 32, 41], and human activity recognition [36]. Therefore, there is increasing interest in verifying their behavior with respect to the requirements they must fulfill to correctly perform their task.

Whenever a property is not satisfied, it is important to be able to adequately characterize and interpret network's misbehaviors, so as to eventually correct them. Here, we consider the notion of *interpretability* provided in [7, 26], which defines it as the degree to which an observer can understand the cause of a decision. Neural networks' nature undermines human capabilities of getting such understanding since their deep and complex architectures, with up to thousands of millions of parameters, makes it impossible for a human to comprehend the rationale of their outputs, even if the underlying mathematical principles are understood and their internal structure and weights are known [33]. Moreover, when it comes to interpreting RNN errors, it is useful to do it through an *operational* and *visual* characterization, as a means for gaining insight into the set of incorrect RNN outputs in reasonable time.

For RNN devoted to sequence classification, one way of checking properties consists in somehow extracting a deterministic finite automaton (DFA) from the network. Since RNN are more expressive than DFA [24], the language of the automaton is, in general, an approximation of the sequences classified as positive by the RNN. Once the automaton is obtained, it can be model-checked against a desired property using an appropriate model-checker [8]. This approach can be implemented by resorting to white-box learning algorithms such as the ones proposed in [42, 44, 45]. However, these procedures do not provide quantitative assessments on how precisely the extracted automaton characterizes the language of the RNN. Actually, this issue is overcome by the black-box learning algorithm proposed in [20] which outputs DFA which are probably correct approximations [39] of the RNN.

In practice, this general approach has several drawbacks, notably but not exclusively that the automaton learned from the RNN may be too large to be explicitly constructed. Another important inconvenience is finding real counterexamples on the RNN when the model-checker fails to verify the property on the DFA. Indeed, since the latter is an approximation of the former, counterexamples found on the DFA could be false negatives. Moreover, it has been advocated in [27] that there is also a need for property checking techniques that interact directly with the actual software that implements the network.

To some extent, these issues can be dealt with learning-based black-box checking (BBC) [30]. BBC is a refinement procedure where finite automata are incrementally built and model-checked against a requirement. Counterexamples generated by the model-checker are validated on the black-box and false negatives are used to refine the automaton. However, BBC requires fixing a formalism for specifying the requirements, typically linear-time temporal logic, and an external model-checker. Besides, the black-box is assumed to be some kind of finite-state machine.

To handle the problem of checking properties over RNN on a black-box setting without the downsides of BBC, we propose a method which performs on-the-fly checking during learning without resorting to an external model-checker.[4] Our approach considers both the RNN and the property as black-boxes and it does not explicitly build nor assumes any kind of state-based representation of them. The key idea consists in learning a regular language which is a probably correct approximation of the intersection of the language of the RNN and the negation of the property.

We show that, when the returned language is empty, the proposed procedure ensures there is an upper bound on the probability of the RNN not satisfying the property. This bound is a function of the parameters of the algorithm. Moreover, if the returned language is nonempty, we prove the requirement is guaranteed to be false with probability 1, and truly bad sequences of the RNN are provided

[4] Our approach differs from on-the-fly BBC as defined in [30] which relies on a strategy for seeking paths in the automaton of the requirement. Also, [45] applies a refinement technique, but it is white-box.

together with an interpretable characterization of the error, in the form of a DFA which is probably correct approximation of the language of faulty behaviors.

The paper is organized as follows. Section 2 briefly reviews probably approximately correct learning and defines the notion of on-the-fly property checking through learning. Section 3 revisits the problem of regular language learning and develops the main theoretical results. Section 4 experimentally validates practical application of the approach in various case studies from different domains. The other sections are devoted to related work and conclusions.

2 PAC learning and black-box property checking

We briefly revisit here *Probably Approximately Correct* (PAC) learning [39]. This summary is mostly based on [5]. We also prove a few useful results regarding the use of PAC learning as a means for checking properties of black boxes.

2.1 PAC learning

Let $\mathcal{X}$ be the *universe* of examples. The *symmetric difference* of $X, X' \subset \mathcal{X}$, denoted $X \oplus X'$, is defined as $X \setminus X' \cup X' \setminus X$, where $X \setminus X'$ is $X \cap \overline{X'}$ and $\overline{X}$ is the complement of X. Examples are assumed to be identically and independently distributed according to an unknown probability distribution $\mathcal{D}$ over $\mathcal{X}$.

A *concept* C is a subset of $\mathcal{X}$. A concept class $\mathcal{C}$ is a set of concepts. Given an *unknown* concept $C \in \mathcal{C}$, the purpose of a *learning* algorithm is to output a hypothesis $H \in \mathcal{H}$ that *approximates* the target concept C, where $\mathcal{H}$, called *hypothesis space*, is a class of concepts possibly different from $\mathcal{C}$.

Approximation between concepts C and H is measured with respect to $\mathcal{D}$ as the probability of an example $x \in \mathcal{X}$ to be in their symmetric difference. This measure, also called *prediction error*, is formalized as $\mathbb{P}_{x \sim \mathcal{D}}[x \in C \oplus H]$.

An *oracle* **EX**, which depends on C and $\mathcal{D}$, takes no input and draws i.i.d examples from $\mathcal{X}$ following $\mathcal{D}$, and tags them as *positive* or *negative* according to whether they belong to C or not. Calls to **EX** are independent of each other.

A PAC-learning algorithm takes as input an *approximation* parameter $\epsilon \in (0,1)$, a *confidence* parameter $\delta \in (0,1)$, a *target* concept $C \in \mathcal{C}$, an oracle **EX**, and a hypothesis space $\mathcal{H}$, and if it terminates, it outputs an $H \in \mathcal{H}$ which satisfies $\mathbb{P}_{x \sim \mathcal{D}}[x \in C \oplus H] \leq \epsilon$ with confidence at least $1 - \delta$, for any $\mathcal{D}$. The output hypothesis H is said to be an ϵ-approximation of C with confidence $1-\delta$. Hereinafter, we refer to H as an (ϵ, δ)-approximation.

The concept class $\mathcal{C}$ is said to be *learnable* in terms of $\mathcal{H}$ if there exists a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a PAC learning algorithm that, when run on a set of examples S, generated by **EX**, of size m_S larger than $m_{\mathcal{H}}(\epsilon, \delta)$, it terminates in polynomial time, measured in terms of its relevant parameters ϵ, δ, m_S, and the size of the representations of examples and concepts.

PAC-learning algorithms may be equipped with other oracles. In this paper, we consider algorithms that make use of *membership* and *equivalence* query oracles, denoted **MQ** and **EQ**, respectively. **MQ** takes as input an example

$x \in \mathcal{X}$ and returns whether $x \in C$ or not. **EQ** takes as input a hypothesis H and answers whether H is an (ϵ, δ)-approximation of C by drawing a sample $S \subset \mathcal{X}$ using **EX**, and checking whether for all $x \in S$, $x \in C$ iff $x \in H$, or equivalently, $S \cap (C \oplus H) = \emptyset$. We will make use of the following results in Sec. 2.2.

Lemma 1. *Let H be an (ϵ, δ)-approximation of C. For any $X \subseteq C \oplus H$, we have that $\mathbb{P}_{x \sim \mathcal{D}}[x \in X] \le \epsilon$ with confidence $1 - \delta$.*

Proof. For any $X \subseteq C \oplus H$, we have that $\mathbb{P}_{x \sim \mathcal{D}}[x \in X] \le \mathbb{P}_{x \sim \mathcal{D}}[x \in C \oplus H]$. Then, $\mathbb{P}_{x \sim \mathcal{D}}[x \in C \oplus H] \le \epsilon$ implies $\mathbb{P}_{x \sim \mathcal{D}}[x \in X] \le \epsilon$. Now, for any $S \subset \mathcal{X}$ such that $S \cap (C \oplus H) = \emptyset$, it follows that $S \cap X = \emptyset$. Therefore, any sample drawn by **EQ** that ensures $\mathbb{P}_{x \sim \mathcal{D}}[x \in C \oplus H] \le \epsilon$ with confidence $1 - \delta$ also guarantees $\mathbb{P}_{x \sim \mathcal{D}}[x \in X] \le \epsilon$ with confidence $1 - \delta$. □

Proposition 1. *Let H be an (ϵ, δ)-approximation of C. For any $X \subseteq \mathcal{X}$:*

$$\mathbb{P}_{x \sim \mathcal{D}}\left[x \in C \cap \overline{H} \cap X\right] \le \epsilon \tag{1}$$

$$\mathbb{P}_{x \sim \mathcal{D}}\left[x \in \overline{C} \cap H \cap X\right] \le \epsilon \tag{2}$$

with confidence at least $1 - \delta$.

Proof. From Lemma 1 because $C \cap \overline{H} \cap X$ and $\overline{C} \cap H \cap X$ are subsets of $C \oplus H$. □

2.2 Using learning for property checking

Property checking consists in verifying whether given any two concepts $C, P \in \mathcal{C}$, it holds that $C \subseteq P$, or equivalently $C \cap \overline{P} = \emptyset$. P is called the *property* to be checked on C. Here, we are interested in devising a PAC-learning based approach to property checking.

2.2.1 Property checking on PAC-learned hypothesis A first idea consists in resorting to a *model-checking* approach. That is, build a model of C and then check whether it satisfies property P. In a black-box setting, we can apply it as follows. Let us assume property $P \in \mathcal{H}$. Given a concept $C \in \mathcal{C}$, use a PAC-learning algorithm to learn a hypothesis $H \in \mathcal{H}$, and then check whether H satisfies P. In order to do this, there must be an effective model-checking procedure. Let us assume there is such a procedure for checking emptiness in $\mathcal{H}$ and $\overline{P} \in \mathcal{H}$. Clearly, in this case, we can pose the problem as checking whether $H \cap \overline{P} = \emptyset$, where H is a PAC-learned model of C.

The question is what could be said about the outcome of this procedure. The following proposition shows that whichever the verdict of the model-checking procedure for $H \cap \overline{P}$, the probability of it not holding for C is bounded by the approximation parameter ϵ, with confidence at least $1 - \delta$.

Proposition 2. *Let H be an (ϵ, δ)-approximation of C. For any $\overline{P} \in \mathcal{H}$:*

1. *if $H \cap \overline{P} = \emptyset$ then $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in C \cap \overline{P}\right] \le \epsilon$, and*
2. *if $H \cap \overline{P} \neq \emptyset$ then $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in \overline{C} \cap H \cap \overline{P}\right] \le \epsilon$,*

with confidence at least $1-\delta$.

Proof.
1. If $H \cap \overline{P} = \emptyset$ then $\overline{P} = \overline{H} \cap \overline{P}$. Thus, $C \cap \overline{P} = C \cap \overline{H} \cap \overline{P}$ and from Proposition 1(1) it follows that $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in C \cap \overline{H} \cap \overline{P}\right] \leq \epsilon$, with confidence at least $1-\delta$.
2. If $H \cap \overline{P} \neq \emptyset$, from Proposition 1(2) we have that $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in \overline{C} \cap H \cap \overline{P}\right] \leq \epsilon$, with confidence at least $1-\delta$. □

In practice, this approach has a few drawbacks.

- When $H \cap \overline{P} \neq \emptyset$, even if with small probability, counterexamples found by the model-checking procedure may not be in C. Therefore, whenever that happens, we would need to make use of the oracle **EX** to draw examples from $H \cap \overline{P}$ and tag them as belonging to C or not in order to trying finding a concrete counterexample in C.
- This approach could only be applied for checking properties for which there exists a model-checking procedure in $\mathcal{H}$. Moreover, the computational time of learning a hypothesis adds up to the time of checking whether it satisfies the property.

2.2.2 On-the-fly property checking through learning To overcome the aforementioned issues, rather than learning an (ϵ, δ)-approximation of C, an appealing alternative is to use the PAC-learning algorithm to learn an (ϵ, δ)-approximation of $C \cap \overline{P} \in \mathcal{C}$. In this context, we have the following result.

Proposition 3. *Let H be an (ϵ, δ)-approximation of $C \cap \overline{P} \in \mathcal{C}$. Then:*

1. *if $H = \emptyset$ then $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in C \cap \overline{P}\right] \leq \epsilon$, and*
2. *if $H \neq \emptyset$ then $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in H \setminus (C \cap \overline{P})\right] \leq \epsilon$,*

with confidence at least $1-\delta$.

Proof. Straightforward from the fact that $\mathbb{P}_{x \sim \mathcal{D}}\left[x \in (C \cap \overline{P}) \oplus H\right] \leq \epsilon$, with confidence at least $1-\delta$. □

The above proposition shows that on-the-fly property checking through learning yields the same theoretical probabilistic assurance as the first one. Nevertheless, from a practical point of view, it has several interesting advantages over the latter:

- A model of the *target* concept C is not explicitly built. This may result in a lower computational time. Besides, it could also be used in cases where it is computationally too expensive to build a hypothesis for C.
- There is no need to resort to model-checking procedures. It may be applied even for checking properties for which no such algorithms exist.
- If the result of the PAC-learning algorithm turns up to be nonempty, it may be the case the oracle **EX** does actually generate an example $x \in C \cap \overline{P}$ during the run of the algorithm. Thus, in this case, x serves as a real witness of the violation of the property.

Algorithm 1: Bounded-L^*

```
Input  : MaxQueryLength, MaxStates, $\epsilon$, $\delta$
Output: DFA $A$
Initialize;
$i \leftarrow 0$;
repeat
    $i \leftarrow i + 1$;
    while OT is not closed or not consistent do
        if OT is not closed then
            $OT$, QueryLengthExceeded $\leftarrow$ Close($OT$);
        end
        if OT is not consistent then
            $OT$, QueryLengthExceeded $\leftarrow$ Consistent($OT$);
        end
    end
    $A \leftarrow$ BuildAutomaton($OT$);
    Answer $\leftarrow$ EQ($A$, $i$, $\epsilon$, $\delta$);
    MaxStatesExceeded $\leftarrow$ STATES($A$) > MaxStates;
    if Answer $\neq$ Yes and not MaxStatesExceeded then
        $OT \leftarrow$ Update($OT$, Answer);
    end
    BoundReached $\leftarrow$ QueryLengthExceeded or MaxStatesExceeded;
until Answer = Yes or BoundReached;
return $A$, Answer;
```

Hereinafter, we exploit this idea in the context of property checking and error characterization for RNN. Concretely, concepts inside the black-box are sequence classifiers implemented as RNN, and properties are formal languages.

3 Learning-based property checking over languages

In this section, we consider the case where the universe $\mathcal{X}$ is the set of words Σ^* over a set of symbols Σ, the target concept is a language $C \subseteq \Sigma^*$, and the hypothesis class $\mathcal{H}$ is the set of *regular languages*, or equivalently of *deterministic finite automata* (DFA).

For the sake of simplicity, we refer to a regular language or its DFA representation indistinctly. For instance, we write $A = \emptyset$ and $A \neq \emptyset$ to mean the language of DFA A is empty and nonempty, respectively.

3.1 Learning DFA with Bounded-L^*

DFA can be learned with L^* [4] which is an iterative learning algorithm that constructs a DFA by interacting with a teacher which makes use of oracles **MQ** and **EQ**. The PAC-based version of L^* satisfies the following property.

Property 1 (From [4])**.** *(1) If L^* terminates, it outputs an (ϵ, δ)-approximation of the target language. (2) L^* always terminates if the target language is regular.*

However, when applied to learning regular approximations of concepts belonging to a more expressive class of languages, L^* may not terminate. In particular, this arrives when using this approach for learning languages of recurrent neural networks (RNN), since in general, this class of networks is strictly more expressive than DFA [24, 35, 37]. To cope with this issue, Bounded-L^* (Algorithm 1) has been proposed in [20]. It bounds the number of iterations of L^* by constraining the maximum number of states of the automaton to be learned and the maximum length of the words used to calling **EX**, which are typically used as parameters to determine the complexity of a PAC-learning algorithm [14].

Bounded-L^* works as follows. The learner builds a table of observations OT by interacting with the teacher. This table is used to keep track of which words are and are not accepted by the target language. OT is built iteratively by asking the teacher membership queries through **MQ**.

OT is a finite matrix $\Sigma^* \times \Sigma^* \to \{0, 1\}$. Its rows are split in two. The 'upper' rows represent a prefix-closed set words and the 'lower' rows correspond to the concatenation of the words in the upper part with every $\sigma \in \Sigma$. Columns represent a suffix-closed set of words. Each cell represents the membership relationship, that is, $OT[u][v] = \mathbf{MQ}(uv)$.

We denote $\lambda \in \Sigma^*$ the empty word and OT_i the value of the observation table at iteration i. The algorithm starts by initializing OT_0 (line 1) with a single upper row $OT_0[\lambda]$, a lower row $OT_0[\sigma]$ for every $\sigma \in \Sigma$, and a single column for the empty word $\lambda \in \Sigma^*$, with values $OT_0[u][\lambda] = \mathbf{MQ}(u)$.

At each iteration $i > 0$, the algorithm makes OT_i *closed* (line 7) and *consistent* (line 10). OT_i is closed if, for every row in the bottom part of the table, there is an equal row in the top part. OT_i is consistent if for every pair of rows u, v in the top part, for every $\sigma \in \Sigma$, if $OT_i[u] = OT_i[v]$ then $OT_i[u\sigma] = OT_i[v\sigma]$.

Once the table is closed and consistent, the algorithm proceeds to build the conjectured DFA A_i (line 13) which accepting states correspond to the entries of OT_i such that $OT_i[u][\lambda] = 1$.

Then, Bounded-L^* calls **EQ** (line 14) to check whether A_i is PAC-equivalent to the target language. For doing this, **EQ** draws a sample S_i of size [4]:

$$m_{S_i}(i, \epsilon, \delta) = \left\lceil \frac{i}{\epsilon} \log \frac{2}{\delta} \right\rceil \tag{3}$$

If for every $s \in S_i$, s belongs to the target language if and only if it belongs to the hypothesis A_i, the equivalence test is passed. In this case, Bounded-L^* terminates and returns A_i.

Property 2 (From [20])**.** *If Bounded-L^* terminates with an automaton A which passes the* **EQ** *test, A is an (ϵ, δ)-approximation of the target language.*

If **EQ** does not pass, the learner receives a counterexample which violates the test. If the maximum number of states of the output hypothesis and/or the maximum length of a **MQ** are not achieved, the learner uses the counterexample to update OT (line 17). Then, it performs a new iteration. Otherwise, it terminates. Thus, upon termination Bounded-L^* may output an automaton A which fails to pass the **EQ** test, that is, A and the target language eventually disagree in $k > 0$ sequences of the sample S drawn by **EQ**. In such cases, the approximation bound guaranteed by the hypotheses produced by Bounded-L^* is given by the following property, which subsumes the previous one (case $k = 0$).

Property 3 (From [20])**.** *If Bounded-L^* terminates with an automaton A producing $k \in [0, m]$ **EQ**-divergences on a sample of size m, computed as in Eq. (3), then A is an $(\hat{\epsilon}, \delta)$-approximation of the target language, for every $\hat{\epsilon} > \epsilon^*$, where*

$$\epsilon^*(m, k, \delta) = \frac{1}{m-k} \log \frac{\binom{m}{k}}{\delta} \tag{4}$$

That is, $\epsilon^*(m, k, \delta)$ is the infimum of the approximation bounds assured by the output hypothesis, with confidence at least $1 - \delta$, provided $k \geq 0$ divergences with the target language are found on a sample S of size m drawn by **EQ**.

Notice that, for fixed k and $\delta \in (0, 1)$, ϵ^* tends to 0 as m_S tends to ∞. That is, it is possible to make ϵ^* smaller than any desired approximation parameter ϵ by letting **EQ** to draw a large enough sample.

Example 1. *Figure 1 shows an example of a run of Bounded-L^* that outputs the DFA of $(ab)^*$ after 2 iterations.*

OT_0	λ
λ	1
a	0
b	0

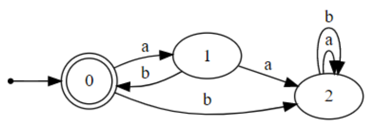
(a)

OT_1	λ
λ	1
a	0
b	0
aa	0
ab	1

(b)

OT_2	λ	b
λ	1	0
a	0	1
b	0	0
bb	0	0
aa	0	0
ab	1	0
ba	0	0
bba	0	0
bbb	0	0

(c)



(d)

Fig. 1. Bounded-L^* example run.

3.2 Property checking and error characterization

Given $C, P \subseteq \Sigma^*$, the property checking problem is posed as determining whether $C \cap \overline{P} = \emptyset$. The error characterization problem consists in learning a regular language which is an (ϵ, δ)-approximation of $C \cap \overline{P}$.

Remark 1. *It is important to notice that C, P, and $\overline{P}$ by no means need to or are assumed to be regular languages. DFA serve to characterize the error and Bounded-L^* is a tool to solve the problem. Hereinafter, no language is regular except otherwise stated.*

Let us start by showing that if the output of Bounded-L^* is nonempty, then the target language inside the black-box is also nonempty.

Lemma 2. *For every $i > 1$, if $A_i \neq \emptyset$ then the target language is nonempty.*

Proof. If $A_i \neq \emptyset$, there exists at least one accepting state, that is, there exists $u \in \Sigma^*$ such that $OT_i[u][\lambda] = 1$. Therefore, at some iteration $j \in [1, i]$, there is a positive membership query for u, i.e, $\mathbf{MQ}_j(u) = 1$. Hence, u belongs to the target language. □

This result is important because it entails that whenever the output for the target language $C \cap \overline{P}$ is nonempty, C does not satisfy P. Moreover, for every entry of the observation table such that $OT[u][v] = 1$, the sequence $uv \in \Sigma^*$ is a counterexample.

Corollary 1. *If Bounded-L^* returns a nonempty DFA for $C \cap \overline{P}$, then $C \cap \overline{P} \neq \emptyset$. Moreover, every $u, v \in \Sigma^*$ such that $OT[u][v] = 1$ is a counterexample.*

Proof. Immediate from Lemma 2. □

It is worth noticing that, by Lemma 2, the execution of Bounded-L^* for $C \cap \overline{P}$ could just be stopped as soon as the observation table has a non-zero entry. This would serve to prove C does not satisfy P. However, we seek providing a more detailed and explanatory characterization of the error, even if approximate, by running the algorithm upon normal termination.

Theorem 1 (Main result). *If Bounded-L^* terminates with an automaton A producing $k \in [0, m]$ $\mathbf{EQ}$-divergences on a sample of size m for $C \cap \overline{P}$, then:*

1. *A is an $(\hat{\epsilon}, \delta)$-approximation of $C \cap \overline{P}$, for every $\hat{\epsilon} > \epsilon^\star(m, k, \delta)$.*
2. *If $A \neq \emptyset$ or $k > 0$, then $C \cap \overline{P} \neq \emptyset$.*

Proof.
1. It follows directly from Property 3.
2. There are two cases. *a)* If $A \neq \emptyset$, then $C \cap \overline{P} \neq \emptyset$ by Corollary 1. *b)* If $A = \emptyset$ and $k > 0$, then $\emptyset \neq A \oplus (C \cap \overline{P}) = \emptyset \oplus (C \cap \overline{P}) = C \cap \overline{P}$. □

4 Experimental results

We implemented a prototype of the proposed black-box on-the-fly property checking through learning based on Bounded-L^*. The approach consists in giving C and P as inputs to the teacher which serves as a proxy of $C \cap \overline{P}$. It is important to emphasize that this approach does not require modeling $\overline{P}$ in any particular way. Instead, to answer $\mathbf{MQ}(u)$ on a word u, the teacher evaluates $P(u)$, complements the output and evaluates the conjunction with the output of $C(u)$. To answer $\mathbf{EQ}(H)$, it draws a sample S of the appropriate size and evaluates $\mathbf{MQ}(u) \iff H(u)$ on every $u \in S$. Therefore, P may be any kind of property, even not regular, or another RNN. Here, we present the results obtained on case studies from different application domains.

1. In the first experiment, training and testing data is generated from a DFA specification of the behavior of a car's cruise-controller subsystem defined in [22]. This case study illustrates the situation where the model-checker found an error on the DFA extracted from the RNN but it was impossible to reproduce it back on the RNN under analysis.
2. The second experiment deals with the model of an e-commerce web service application described in [20, 25]. In this case, we injected on purpose bad sequences in the training set in order to showcase they are actually found by our on-the-fly checking procedure which outputs a DFA depicting the error.
3. The third case study comes from the field of system security. Here, we analyze the behavior of an RNN trained with logs of a Hadoop Distributed File System (HDFS) from [10]. In this example, no a priori characterization of the language of normal logs was known. The experiment showed that on-the-fly checking exposed the fact that the RNN could actually incur in false positives, that is, predicting a sequence is normal when it is not, even if no such cases were found during training on a test dataset. The output DFA depicts the error and helps understanding the logs where such misleading classifications occur.
4. The last experiment analyzes a case study from bioinformatics, namely TATA-box subsequence recognition in promoter DNA sequences [28]. In this example, it was impossible to actually extract a DFA from the RNN. Nevertheless, on-the-fly checking managed to check the RNN was compliant with the specified requirement.

4.1 Cruise controller

We trained an RNN with a dataset containing 200K positive and negative sequences upto a maximum length of 16 from a cruise controller model from [22] (Fig. 3). The measured error of the RNN on a test set of 16K sequences was 0,09%. The property P used in this example is shown in Fig. 2. It models the requirement that a *break* event can only happen if a $gas|acc$ has already occurred before and no other *break* has happened inbetween.

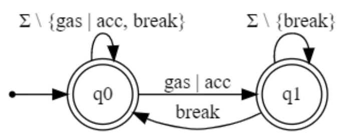


Fig. 2. Requirement of the cruise controller example

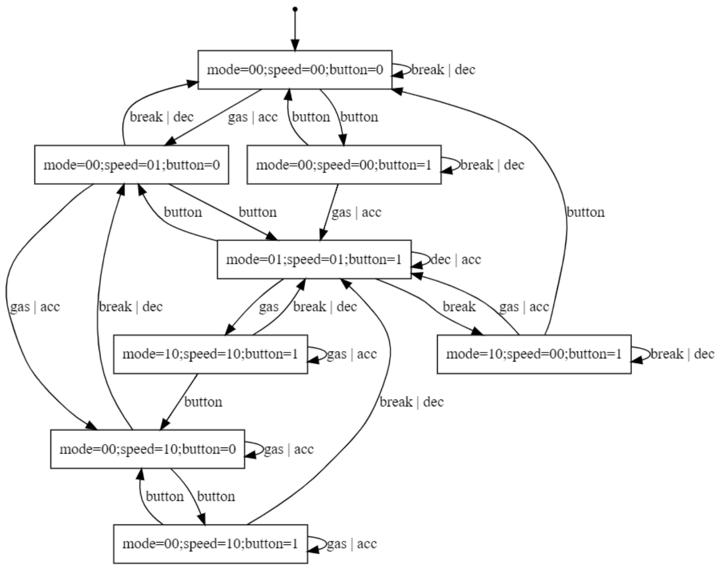


Fig. 3. Model of the cruise controller example

First, we used our on-the-fly technique and found out that every run ended up conjecturing $C \cap \overline{P} = \emptyset$ with perfect **EQ** tests. Running times, **EQ** test sizes and ϵ^* for different values of ϵ and δ of these experiments are shown in Table 1.

Configuration		**Exec. time (s)**			**First counter-example**	**Average EQ test size**	**Average ϵ^***
ϵ	δ	**min**	**max**	**avg**			
0.01	0.01	0.003	0.006	0.004	-	669	0.00688
0.001	0.01	0.061	0.096	0.075	-	6,685	0.00069
0.0001	0.01	0.341	0.626	0.497	-	66,847	0.00007

Table 1. Cruise controller: On-the-fly verification of RNN.

Second, we extracted PAC DFA from the network alone, with a timeout of 200s (Table 2). For the first configuration, one run timed out and four completed. All extracted DFA exceeded the maximum number of states bound, and three of them did not verify the property. For the second one, there were two time outs, and three successful extractions. Every one of the extractions exceeded the maximum states bound and two of them did not verify the property. Finally, for the third one, every run timed out.

We then checked these DFA against the property with an external verification algorithm for computing intersection of DFA. It turned out that all the counterexamples found by the model-checking procedure where actually classified as negative by the RNN under analysis.

Configuration		Exec. time (s)			Average EQ test size	Average ϵ^*
ϵ	δ	min	max	avg		
0.01	0.01	11.633	200.000	67.662	808	0.05
0.001	0.01	52.362	200.000	135.446	8,071	0.03
0.0001	0.01	-	-	-	-	-

Table 2. Cruise controller: PAC DFA extraction from RNN.

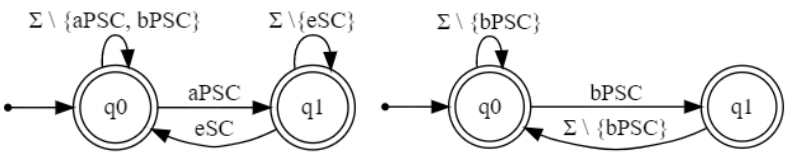
Then, we used **EX** to generate 2 million sequences for each of the DFA H not checking the property. It turned out that none was accepted by both $H \cap \overline{P}$ and the RNN. Thus, we cannot disprove the conjecture that the RNN is correct with respect to P obtained with the on-the-fly technique. Moreover, the second approach required considerable more effort because of its misleading verdict.

This experiment showcases a key aspect where on-the-fly property checking stands out: when it presents a witness it is real with probability 1, whereas this is not the case when verifying a property on a model of a target RNN.

4.2 E-commerce web application API

We analyzed an RNN trained with a dataset of 100K positive and negative sequences upto length 16 drawn from the model of the e-commerce system from [20, 25], together with sequences that violate the properties to be checked. These *canary* sequences were added on purpose to check whether the RNN actually learned those faulty behaviors and whether our technique was able to figure that out. The RNN was trained until no error was measured on a test set of 16K.

We checked the RNN against the following regular properties. 1) It is not possible to buy products in the shopping cart (modelled by symbol $bPSC$) when the shopping cart is empty. Symbols $aPSC$ and eSC model adding products to and emptying the shopping cart, respectively (Fig. 4a). 2) It is not possible to execute the action $bPSC$ two or more times in a row (Fig. 4b).



(a) E-commerce property 1 (b) E-commerce property 2

Fig. 4. E-commerce properties

In this experiment, we only checked properties with the on-the-fly approach. Nevertheless, we exctrated DFA for different values of ϵ and δ as a reference for comparing computational performance (Table 3).

Configuration		Exec. time (s)			Average EQ test size	Average ϵ^*
ϵ	δ	min	max	avg		
0.01	0.01	16.863	62.125	36.071	863	0.00534
0.001	0.01	6.764	9.307	7.864	8,487	0.00054
0.0001	0.01	18.586	41.137	30.556	83,482	0.00006

Table 3. E-commerce: PAC DFA extraction from RNN.

On-the-fly property checking concluded both properties were not satisfied (Table 4). In average, it took more time to output a PAC DFA of the language of faulty behaviors than extracting a PAC DFA of the RNN alone. Nevertheless, counterexamples were found (in average) orders of magnitude faster than the latter for requirement 1), while it took comparable time for requirement 2), which revealed to be harder to check than the former.

Prop	Configuration		Excecution time (s)			First counter-example	Average EQ test size	Average ϵ^*
	ϵ	δ	min	max	avg			
1	0.01	0.01	87.196	312.080	174.612	3.878	891	0.00517
	0.001	0.01	0.774	203.103	102.742	0.744	9,181	0.00050
	0.0001	0.01	105.705	273.278	190.948	2.627	94,573	0.00005
2	0.01	0.01	0.002	487.709	148.027	80.738	752	0.00619
	0.001	0.01	62.457	600.000	428.400	36.606	8,765	0.00053
	0.0001	0.01	71.542	451.934	250.195	41.798	87,641	0.00005

Table 4. E-commerce: On-the-fly verification of RNN.

Examples of outputs of the on-the-fly property checking algorithm for properties 1 and 2, with parameters $\epsilon = 0.0001$ and $\delta = 0.01$, are shown in Figs. 5 and 6, respectively. They contain valuable information about all possible consequences of the errors in terms of understanding and correcting them. At the same time, the technique is able to present real witnesses that belong to this language and to the network under analysis.

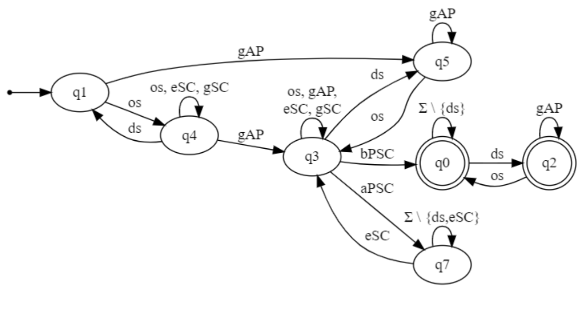


Fig. 5. Example of output for Property 1 on the e-commerce RNN.

The DFA depicted in Fig. 5 shows that the RNN classifies as correct a sequence where event gAP (get available products) has occurred but either no product has been added to the shopping basket (i.e., no $aPSC$ has occurred) or this has been emptied (i.e., eSC happened), within an active session (i.e, the last open session event os is not followed by a closed session event cs). This is illustrated, for instance, by the sub-DFA involving states q_1, q_4, q_3, and q_0. Actually, Fig. 6 shows that not only property 2 is not satisfied by the RNN but also property 1, as well, since it is possible to reach q_3 without executing $aPSC$.

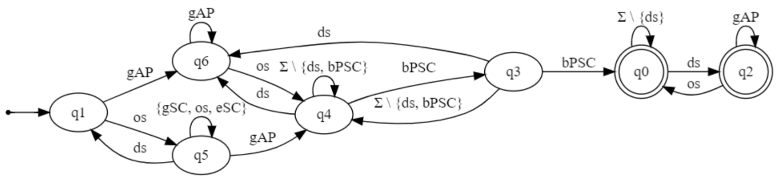


Fig. 6. Example of output for Property 2 on the e-commerce RNN.

4.3 Analysis of HDFS logs

In this case study we deal with a dataset of logs of a Hadoop Distributed File System (HDFS) from [10]. The logs were labeled as normal or abnormal by Hadoop experts. It contains 4855 training normal variable-length logs. Each log is pre-processed into a sequence of numeric symbols from 0 to 28. These logs were used to train an RNN-based auto-regressive language model (LM). That is, the output of the RNN is the conditional probability of the next symbol given all the previous ones [6]. This RNN can be used to build a sequence classifier in several ways. In this case, we use the RNN to predict the probability of a sequence. Such prediction is then compared to a given threshold. If the probability is greater than the threshold, the sequence is considered to be normal (positive), otherwise is declared to be abnormal (negative).

For this experiment, the classifier was implemented as a Python function which queried the RNN and returned the prediction. We used a threshold of 2×10^{-7} and a perfectly balanced test set containing 33600 normal and abnormal logs. The classifier achieved an accuracy of 98.35% with no false positives. That is, no abnormal log in the test set is ever misclassified as normal by the classifier. We verified the following properties. 1) The classifier does not classify as normal a sequence that contains a symbol which only appears in abnormal logs. This set, identified as A, contains 12 symbols, namely $6, 7, 9, 11-14, 18, 19, 23, 26-28$. 2) The classifier always classifies as normal logs where the sum of occurrences of symbols "4" and "21", often seen at the beginning of normal logs, is at most 5. The purpose of checking these properties is to determine whether the RNN actually learned these patterns as characteristic of normal logs.

The results of on-the-fly checking are shown in Table 5. For each configuration, 5 runs were executed. For property 2), we found that it is satisfied by

Prop	**Configuration**		**Excecution time (s)**			**First counter-example**	**Average EQ test size**	**Average ϵ^***
	ϵ	δ	**min**	**max**	**avg**			
1)	0.01	0.01	209.409	1,121.360	555.454	5.623	932	0.0050
	0.001	0.001	221.397	812.764	455.660	1.321	12,037	0.0006
2)	0.01	0.01	35.131	39.762	37.226	-	600	0.0077
	0.001	0.001	252.202	257.312	254.479	-	8,295	0.0008

Table 5. HDFS logs: On-the-fly verification of RNN.

the classifier with PAC guarantees. However, in the case of property 1), all runs found counterexamples and output a PAC DFA of the sequences violating the property. This means the classifier can label as normal a log containing symbols in the set A of symbols that only appeared in logs tagged as abnormal by experts. Notice that this observation highlights a discrepancy with the results obtained on the test set where the classifier incurred in no false positives, therefore, it did not classify as normal any log containing symbols in A.

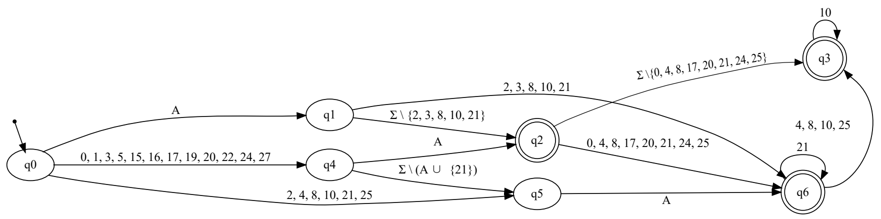


Fig. 7. Model of the error of the Deep Log network verified against property 1.

The output PAC DFA can be used to understand in more detail mistakes of the classifier in order to improve its performance. Fig. 7 depicts the DFA obtained in a run of the algoritm with parameters $\epsilon = 0.01$ and $\delta = 0.01$. For instance, we can see that the RNN can recognize as normal logs of length two which start with a symbol in the set A. This is shown by paths q_0, q_1, q_2 and q_0, q_1, q_6 of the DFA. Fig. 8 shows that the predicted probability of logs corresponding to these paths were always significantly above the threshold (by an order of magnitude of 4). The fact that the classifier could produce such classifications is not obvious, since the probability assigned by the RNN to an unseen symbol is expected to be low (because no such symbol has been seen during training).

4.4 TATA-box recognition in DNA promoter sequences

Promoter region recognition in DNA sequences is an active research area in bioinformatics. Recently, tools based on neural networks, such as CNN and LSTM, have been proposed for such matter [28]. Promoters are located upstream near

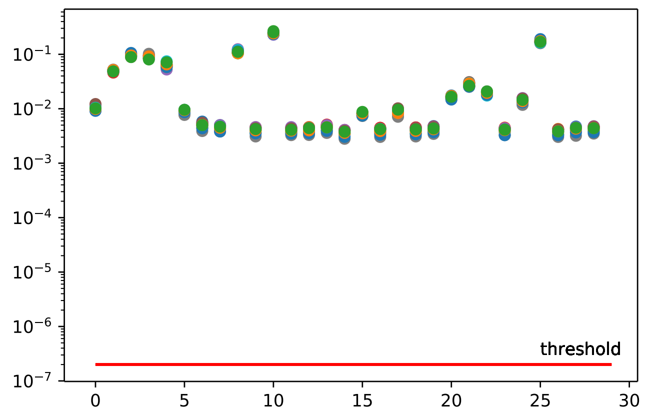


Fig. 8. Predicted probability for logs of length 2 starting with A (in log scale).

the gene transcription start site (TSS) and control the activation or repression of the genes. The TATA-box is a particular promoter subsequence that indicates to other molecules where transcription begins. It is a T/A-rich (i.e., more T's and A's than C's and G') subsequence of 6 base pairs (bp) located between positions –30bp to –25bp, where +1bp is the TSS. The goal of this experiment is not to develop a neural network for promoter classification, but to study whether an RNN trained with TATA and non-TATA promoter sequences is able to distinguish between them, that is, it is capable of determining whether a DNA sequence contains a TATA region. For such task, we trained an RNN composed of an LSTM and a dense layer for classification, with a dataset of the most representative TATA (2067 sequences) and non-TATA (14388 sequences) human promoters of length 50bp from positions -48bp to +1bp. The dataset was downloaded from the website EPDnew[5]. The RNN was trained until achieving an accuracy of 100%.

Configuration		**Exec. time (s)**			**Average EQ test size**	**Average ϵ^***
ϵ	δ	**min**	**max**	**avg**		
0.01	0.01	5.098	5.259	5.168	600	0.00768
0.001	0.001	65.366	66.479	65.812	8,295	0.00083
0.0001	0.0001	865.014	870.663	867.830	105,967	0.00008

Table 6. TATA-box: On-the-fly verification of RNN.

We ran the on-the-fly algorithm to check whether the language of the RNN was included in the set of sequences containing a TATA-box. The property was coded as a Python program which counts the number of T's, A's, C's and G's

[5] `https://epd.epfl.ch//index.php`

in the subsequence from position -30bp to -25bp of the genomic sequence, and checks whether the sum of T's and A's is greater than the sum of C's and G's. In this case, the oracle **EX** was parameterized to generate sequences of length 50 over the alphabet $\{A, T, C, G\}$. We were not able to extract an automaton representing the target network because Bounded-L^* always hit a timeout before being able to construct a DFA. Therefore, extracting a model and then checking it was not achievable in practice due to state explosion. Nevertheless, we were able to perform the analysis with on-the-fly checking (Table 6). In this case, every run of the algorithm output an empty language of the error, thus conjecturing the RNN PAC-verifies the property. Notice that for parameters $\epsilon = \delta = 0.0001$, the PAC **EQ** sample is more than 6 times larger than the training set.

5 Related Work

Learning and automata-theoretic verification have been combined in several ways for checking temporal requirements of systems. For instance, [3, 9, 12] do compositional verification by learning assumptions. These methods are white-box and require an external decision procedure. Learning regular approximations of FIFO automata for verifying regular properties has been explored in [40]. A technique for verifying properties on learned automata is presented in [13]. It iteratively applies Trakhtenbrot-Barzdin algorithm [38] on several training sets until the inferred automaton is an invariant sufficient to prove the property.

Learning based testing (LBT) [23] is a BBC approach for generating test cases. It relies on incrementally building hypotheses of the system under test (SUT) and verifying whether they satisfy the requirement through an external model-checker. Counterexamples serve as test-cases for the SUT. To the best of our knowledge, it does not provide provable probabilistic guarantees. Recent work [21] proposes a sound extension but it requires relaxing the black-box setting by observing and recording the internal state of the SUT.

For checking safety properties on feed-forward neural networks (FFNN), a general white-box approach based on Linear Programming (LP) and Satisfiability Modulo Theories (SMT) has been first explored in [31]. Lately, this approach has been further explored, for instance, in [11, 15, 16]. For RNN, an approach for adversarial accuracy verification is presented in [43] based a white-box technique to extract DFA from RNN. Experimental evaluation is carried out work on Tomita grammars, which are small regular languages over the $\{0, 1\}$-alphabet. That approach does not offer any guarantee on how well the DFA approximates the RNN. RNSVerify [2] implements white-box verification of safety properties by unrolling the RNN and resorting to LP to solve a system of constraints. The method strongly relies on the internal structure and weight matrices of the RNN. Overall, these techniques are white-box and are not able to handle non-regular properties. Besides, they do not address the problem of producing interpretable error characterizations.

Finally, statistical model checking (SMC) the system under analysis and/or the property is stochastic [1, 19]. The objective of SMC is to check whether a stochastic system, such as a Markov decision process, satisfies a property with a probability greater or equal to a certain threshold θ. The problem we address in this work is different as neither the system nor the property is stochastic. Our approach provides statistical guarantees that the language of an RNN C is included in another language (the property P) or provides a PAC model of the language $C \cap \overline{P}$, along with actual counterexamples showing it is not.

6 Conclusions

The contribution of this paper is a black-box on-the-fly learning-based approach for checking properties on RNN. Our technique interacts with the software artifact implementing the RNN-based classifier through queries. It does not build a priori the state-space of the RNN but directly constructs an approximation of the intersection of the RNN with the negation of the requirement. Besides, in contrast to other approaches, the computational complexity of our technique does not depend on the size (hidden layers, weights) of the RNN and of external model-checkers or solvers, but rather on the size of its alphabet Σ, together with user-controlled inputs of the algorithm such as the approximation ϵ and confidence δ parameters, and the maximum number of states and length of membership queries. Moreover, it is not restricted to any particular class of RNN and can also be used to check non-regular properties. Our algorithm outputs an interpretable characterization of an approximation of the set of incorrect behaviors.

We implemented the approach and shown its practical applicability for checking properties on several case studies from different application domains. We compared, when possible, the results of on-the-fly checking through learning against model-checking the extracted PAC models of the RNN alone. The experiments were promising as they provided empirical evidence that the on-the-fly approach typically performs faster than extracting a DFA from the RNN under analysis whenever the property is probably approximately satisfied. Moreover, when a property is found not to be satisfied by the RNN, the experiments shown the output DFA contains valuable information about all possible consequences of the error in terms of understanding and correcting it.

Acknowledgments This work has been partially funded by an ICT4V - Information and Communication Technologies for Verticals master thesis grant `POS_ICT4V_2016_1_15`, and ANII - Agencia Nacional de Investigación e Innovación research grants `FSDA_1_2018_1_154419` and `FMV_1_2019_1_155913`.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1) (Jan 2018)
2. Akintunde, M.E., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of rnn-based neural agent-environment systems. In: AAAI. pp. 6006–6013 (2019)

3. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: CAV. pp. 548–562. Springer (2005)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (Nov 1987)
5. Angluin, D.: Computational learning theory: Survey and selected bibliography. In: STOC. pp. 351–369. ACM (1992)
6. Bengio, Y., Ducharme, R., Vincent, P., Janvin, C.: A neural probabilistic language model. J. Mach. Learn. Res. **3**, 1137–1155 (Mar 2003)
7. Biran, O., Cotton, C.V.: Explanation and justification in machine learning : A survey. In: IJCAI Workshop on Explainable Artificial Intelligence (XAI) (2017)
8. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
9. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: TACAS 2003. pp. 331–346. Springer (2003)
10. Du, M., Li, F., Zheng, G., Srikumar, V.: Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: SIGSAC CCS. p. 1285–1298. ACM (2017)
11. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: ATVA. LNCS, vol. 10482, pp. 269–286. Springer (2017)
12. Feng, L., Han, T., Kwiatkowska, M., Parker, D.: Learning-based compositional verification for synchronous probabilistic systems. In: ATVA. pp. 511–521. Springer (2011)
13. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. ENTCS **138**, 21–36 (09 2004)
14. Heinz, J., de la Higuera, C., van Zaanen, M.: Formal and empirical grammatical inference. In: ACL Annual Meeting. pp. 2:1–2:83. ACL (2011)
15. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV. LNCS, vol. 10426, pp. 3–29. Springer (2017)
16. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: CAV. LNCS, vol. 10426, pp. 97–117. Springer (2017)
17. Kim, J., Kim, J., Thu, H.L.T., Kim, H.: Long short term memory recurrent neural network classifier for intrusion detection. In: PlatCon. pp. 1–5. IEEE (2016)
18. Kocić, J., Jovičić, N., Drndarević, V.: An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. Sensors **19**(9), 2064 (2019)
19. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S.A., Grosu, R.: Statistical Model Checking, pp. 478–504. Springer, Cham (2019)
20. Mayr, F., Yovine, S.: Regular inference on artificial neural networks. In: Machine Learning and Knowledge Extraction. pp. 350–369. Springer, Cham (2018)
21. Meijer, J., van de Pol, J.: Sound black-box checking in the learnlib. Innovations in Systems and Software Engineering **15**(3-4), 267–287 (2019)
22. Meinke, K., Sindhu, M.A.: LBTest: A learning-based testing tool for reactive systems. In: STVV. pp. 447–454. IEEE (March 2013)
23. Meinke, K.: Learning-based testing: recent progress and future prospects. In: Machine Learning for Dynamic Software Analysis: Potentials and Limits, pp. 53–73. Springer (2018)
24. Merrill, W.: Sequential neural networks as automata. arXiv preprint arXiv:1906.01615 (2019)
25. Merten, M.: Active automata learning for real life applications. Ph.D. thesis, Technischen Universität Dortmund (2013)

26. Miller, T.: Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence **267**, 1 – 38 (2019)
27. Odena, A., Olsson, C., Andersen, D., Goodfellow, I.J.: Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In: ICML. vol. 97, pp. 4901–4911. PMLR (2019)
28. Oubounyt, M., Louadi, Z., Tayara, H., Chong, K.T.: Deepromoter: Robust promoter predictor using deep learning. Frontiers in genetics **10** (2019)
29. Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M., Thomas, A.: Malware classification with recurrent networks. In: ICASSP. pp. 1916–1920. IEEE (2015)
30. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. Journal of Automata, Languages and Combinatorics **7**(2), 225–246 (2002)
31. Pulina, L., Tacchella, A.: Challenging SMT solvers to verify neural networks. AI Commun. **25**(2), 117–135 (2012)
32. Rhode, M., Burnap, P., Jones, K.: Early stage malware prediction using recurrent neural networks. Computers & Security **77** (08 2017)
33. Ribeiro, M.T., Singh, S., Guestrin, C.: "why should i trust you?": Explaining the predictions of any classifier. In: SIGKDD Knowledge Discovery and Data Mining. p. 1135–1144. ACM (2016)
34. Scheiner, N., Appenrodt, N., Dickmann, J., Sick, B.: Radar-based road user classification and novelty detection with recurrent neural network ensembles. In: Intelligent Vehicles Symposium. pp. 722–729. IEEE (2019)
35. Siegelmann, H.T., Sontag, E.D.: On the computational power of neural nets. In: COLT. pp. 440–449. ACM (1992)
36. Singh, D., Merdivan, E., Psychoula, I., Kropf, J., Hanke, S., Geist, M., Holzinger, A.: Human activity recognition using recurrent neural networks. In: International Cross-Domain Conference for Machine Learning and Knowledge Extraction. pp. 267–274. Springer (2017)
37. Suzgun, M., Belinkov, Y., Shieber, S.M.: On evaluating the generalization of lstm models in formal languages. CoRR **abs/1811.01001** (2018)
38. Trakhtenbrot, B.A., Barzdin, I.M.: Finite automata : behavior and synthesis. North-Holland (1973)
39. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)
40. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Actively learning to verify safety for fifo automata. In: Foundations of Software Technology and Theoretical Computer Science. pp. 494–505. Springer (2004)
41. Vinayakumar, R., Alazab, M., Soman, K., Poornachandran, P., Venkatraman, S.: Robust intelligent malware detection using deep learning. IEEE Access **7**, 46717–46738 (2019)
42. Wang, Q., Zhang, K., II, A.G.O., Xing, X., Liu, X., Giles, C.L.: A comparison of rule extraction for different recurrent neural network models and grammatical complexity. CoRR **abs/1801.05420** (2018)
43. Wang, Q., Zhang, K., Liu, X., Giles, C.L.: Verification of recurrent neural networks through rule extraction. In: AAAI Spring Symposium on Verification of Neural Networks (VNN19) (2019)
44. Wang, Q., Zhang, K., Ororbia, II, A.G., Xing, X., Liu, X., Giles, C.L.: An empirical evaluation of rule extraction from recurrent neural networks. Neural Comput. **30**(9), 2568–2591 (Sep 2018)
45. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. In: ICML. vol. 80. PMLR (2018)
46. Yin, C., Zhu, Y., Fei, J., He, X.: A deep learning approach for intrusion detection using recurrent neural networks. IEEE Access **5**, 21954–21961 (2017)