Property-Directed Verification and Robustness Certification of Recurrent Neural Networks⋆

Igor Khmelnitsky[a,b], Daniel Neider[c], Rajarshi Roy[c], Xuan Xie[c],
Benoît Barbot[d], Benedikt Bollig[a], Alain Finkel[a,g], Serge Haddad[a,b],
Martin Leucker[e], and Lina Ye[a,b,f]

[a] Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, France
[b] Inria, France
[c] Max Planck Institute for Software Systems, Kaiserslautern, Germany
[d] Université Paris-Est Créteil, France
[e] Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Germany
[f] CentraleSupélec, Université Paris-Saclay, France
[g] Institut Universitaire de France, France

Abstract. This paper presents a property-directed approach to verifying recurrent neural networks (RNNs). To this end, we learn a deterministic finite automaton as a *surrogate model* from a given RNN using active automata learning. This model may then be analyzed using *model checking* as a verification technique. The term *property-directed* reflects the idea that our procedure is guided and controlled by the given property rather than performing the two steps separately. We show that this not only allows us to discover *small* counterexamples fast, but also to generalize them by pumping towards faulty flows hinting at the underlying error in the RNN. We also show that our method can be efficiently used for *adversarial robustness certification* of RNNs.

1 Introduction

Recurrent neural networks (RNNs) are a state-of-the-art tool to represent and learn sequence-based models. They have applications in time-series prediction, sentiment analysis, and many more. In particular, they are increasingly used in safety-critical applications and act, for example, as controllers in cyber-physical systems [1]. Thus, there is a growing need for formal verification. However, research in this domain is only at the beginning. While formal-methods based techniques such as *model checking* [4] have been successfully used in practice and reached a certain level of industrial acceptance, a transfer to machine-learning

⋆ The first four authors contributed equally, the remaining authors are ordered alphabetically. This work was partly supported by the PHC PROCOPE 2020 project LeaRNNify (number 44707TK), funded by DAAD and Campus France and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) grant number 434592664.

algorithms has yet to take place. We apply it on machine-learning artifacts rather than on the algorithm.

An emerging research stream aims at extracting, from RNNs, state-based formalisms such as finite automata [3, 16, 17, 20, 21, 25]. Finite automata turned out to be useful for understanding and analyzing all kinds of systems using testing or model checking. In the field of formal verification, it has proven to be beneficial to run the extraction and verification process simultaneously. Moreover, the state space of RNNs tends to be prohibitively large, or even infinite, and so do incremental abstractions thereof. Motivated by these facts, we propose an intertwined approach to verifying RNNs, where, in an incremental fashion, grammatical inference and model checking go hand-in-hand. Our approach is inspired by black-box checking [22], which *exploits* the property to be verified *during* the verification process. Our procedure can be used to find misclassified examples or to verify a system that the given RNN controls.

Property-directed verification. Let us give a glimpse of our method. We consider an RNN R as a binary classifier of finite sequences over a finite alphabet Σ. In other words, R represents the set of strings that are classified as positive. We denote this set by $L(R)$ and call it the *language* of R. Note that $L(R) \subseteq \Sigma^*$. We would like to know whether R is compatible with a given specification A, written $R \models A$. Here, we assume that A is given as a (deterministic) finite automaton. Finite automata are algorithmically feasible, albeit having a reasonable expressive power: many abstract specification languages such as temporal logics or regular expressions can be compiled into finite automata [10].

But what does $R \models A$ actually mean? In fact, there are various options. If A provides a complete characterization of the sequences that are to be classified as positive, then $\models$ refers to language equivalence, i.e., $L(R) = L(A)$. Note that this would imply that $L(R)$ is supposed to be a regular language, which may rarely be the case in practice. Therefore, we will focus on checking inclusion $L(R) \subseteq L(A)$, which is more versatile as we explain next.

Suppose N is a finite automaton representing a negative specification, i.e., R must classify words in $L(N)$ as negative at any cost. In other words, R does not produce false positives. This amounts to checking that $L(R) \subseteq L(\overline{N})$ where $\overline{N}$ is the "complement automaton" of N. For instance, assume that R is supposed to recognize valid XML documents over a finite predefined set of tags. Seen as a set of strings, this is not a regular language. However, we can still check whether $L(R)$ only contains words where every opening tag <*tag-name*> is eventually followed by a closing tag </*tag-name*> (while the number of opening and the number of closing tags may differ). As negative specification, we can then take an automaton N accepting the corresponding *regular* set of strings. For example, `<book><author></author><author></book>` $\in L(N)$, since the second occurrence of `<author>` is not followed by some `</author>` anymore. On the other hand, we have `<book><author><author></author></book>` $\in L(\overline{N})$, as `<book>` and `<author>` are always eventually followed by their closing counterpart.

Symmetrically, suppose P is a finite automaton representing a *positive* specification so that we can find false negative classifications: If P represents the words that R *must classify as positive*, we would like to know whether $L(P) \subseteq L(R)$. Our procedure can be run using the complement of P as specification and inverting the outputs of R, i.e., we check, equivalently, $L(\overline{R}) \subseteq L(\overline{P})$.

An important instance of this setting is *adversarial robustness certification*, which measures a neural network's resilience against adversarial examples. Given a (regular) set of words L classified as positive by the given RNN, the RNN is *robust* wrt. L if slight modifications in a word from L do not alter the RNN's judgement. This notion actually relies on a distance function. Then, P is the set of words whose distance to a word in L is bounded by a predefined threshold, which is regular for several popular distances such as the *Hamming* or *Levenshtein distance*. Similarly, we can also check whether the neighborhood of a regular set of words preserves a negative classification.

So, in all these cases, we are faced with the question of whether the language of an RNN R is contained in the (regular) language of a finite automaton A. Our approach to this problem relies on black-box checking [22], which has been designed as a combination of model checking and testing in order to verify finite-state systems and is based on Angluin's L^* learning algorithm [2]. L^* produces a sequel of *hypothesis* automata based on queries to R. Every such hypothesis $\mathcal{H}$ may already share some structural properties with R. So, instead of checking conformance of $\mathcal{H}$ with R, it is worthwhile to first check $L(\mathcal{H}) \subseteq L(A)$ using classical model-checking algorithms. If the answer is affirmative, we apply statistical model checking to check $L(R) \subseteq L(\mathcal{H})$ to confirm the result. Otherwise, a counterexample is exploited to refine $\mathcal{H}$, starting a new cycle in L^*. Just like in black-box checking, our experimental results suggest that the process of interweaving automata learning and model checking is beneficial in the verification of RNNs and offers advantages over more obvious approaches such as (pure) statistical model checking or running automata extraction and model checking in sequence. A further key advantage of our approach is that, unlike in statistical model checking, we often find a *family* of counterexamples, in terms of loops in the hypothesis automaton, which testify conceptual problems of the given RNN.

Note that, though we only cover the case of binary classifiers, our framework is in principle applicable to multiple labels using one-vs-all classification.

Related Work. Mayr and Yovine describe an adaptation of the PAC variant of Angluin's L* algorithm that can be applied to neural networks [17]. As L* is not guaranteed to terminate when facing non-regular languages, the authors impose a bound on the number of states of the hypotheses and on the length of the words for membership queries. In [16,18], Mayr et al. propose *on-the-fly property checking* where one learns an automaton approximating the intersection of the RNN language and the complement of the property to be verified. Like the RNN, the property is considered as a black box, only decidability of the word problem is required. Therefore, the approach is suitable for non-regular specifications.

Weiss et al. introduce a different technique to extract finite automata from RNNs [25]. It also relies on Angluin's L* but, moreover, uses an orthogonal abstraction of the given RNN to perform equivalence checks between them.

The paper [1] studies formal verification of systems where an RNN-based agent interacts with a linearly definable environment. The verification procedure proceeds by a reduction to feed-forward neural networks (FFNNs). It is complete and fully automatic. This is at the expense of the expressive power of the specification language, which is restricted to properties that only depend on bounded prefixes of the system's executions. In our approach, we do not restrict the kind of regular property to verify. The work [13] also reduces the verification of RNNs to FFNN verification. To do so, the authors calculate inductive invariants, thereby avoiding a blowup in the network size. The effectiveness of their approach is demonstrated on audio signal systems. Like in [1], a time interval is imposed in which a given property is verified.

For adversarial robustness certification, Ryou et al. [23] compute a convex relaxation of the non-linear operations found in the recurrent cells for certifying the robustness of RNNs. The authors show the effectiveness of their approach in speech recognition. Besides, MARBLE [8] builds a probabilistic model to quantize the robustness of RNNs. However, these approaches are white-box based and demand the full structure and information of neural networks. Instead, our approach is based on learning with black-box checking.

Elboher et al. present a counter-example guided verification framework whose workflow shares similarities with our property-guided verification [9]. However, their approach addresses FFNNs rather than RNNs. For recent progress in the area of safety and robustness verification of deep neural networks, see [15].

Outline. In Section 2, we recall basic notions such as RNNs and finite automata. Section 3 describes two basic algorithms for the verification of RNNs, before we present property-directed verification in Section 4. How to handle adversarial robustness certification is discussed in Section 5. The experimental evaluation and a thorough discussion can be found in Section 6.

2 Preliminaries

In this section, we provide definitions of basic concepts such as languages, recurrent neural networks, finite automata, and Angluin's L* algorithm.

Words and Languages. Let Σ be an alphabet, i.e., a nonempty finite set, whose elements are called *letters*. A (finite) word w over Σ is a sequence $a_1 \dots a_n$ of letters $a_i \in \Sigma$. The length of w is defined as $|w| = n$. The unique word of length 0 is called the *empty word* and denoted by λ. We let Σ^* refer to the set of all words over Σ. Any set $L \subseteq \Sigma^*$ is called a *language* (over Σ). Its complement is $\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$. For two languages $L_1, L_2 \subseteq \Sigma^*$, we let $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$. The symmetric difference of L_1 and L_2 is defined as $L_1 \oplus L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1)$.

Probability Distributions. In order to sample words over Σ, we assume a probability distribution $(p_a)_{a \in \Sigma}$ on Σ (by default, we pick the uniform distribution) and a "termination" probability $p \in (0, 1]$. Together, they determine a natural probability distribution on Σ^* given, for $w = a_1 \dots a_n \in \Sigma^*$, by $\Pr(w) = p_{a_1} \cdot \ldots \cdot p_{a_n} \cdot (1-p)^n \cdot p$. According to the geometric distribution, the expected length of a word is $(1/p)-1$, with a variance of $(1-p)/p^2$. Let $0 < \varepsilon < 1$ be an error parameter and $L_1, L_2 \subseteq \Sigma^*$ be languages. We call L_1 *ε-approximately correct* wrt. L_2 if $\Pr(L_1 \setminus L_2) = \sum_{w \in L_1 \setminus L_2} \Pr(w) < \varepsilon$.

Finite Automata and Recurrent Neural Networks. We employ two kinds of language acceptors: finite automata and recurrent neural networks.

Recurrent neural networks (RNNs) are a generic term for artificial neural networks that process sequential data. They are particularly suitable for classifying sequences of varying length, which is essential in domains such as natural language processing (NLP) or time-series prediction. For the purposes of this paper, it is sufficient to think of an RNN R as an effective function $R : \Sigma^* \to \{0, 1\}$, which determines its language as $L(R) = \{w \in \Sigma^* \mid R(w) = 1\}$. Its complement $\overline{R}$ is defined by $\overline{R}(w) = 1 - R(w)$ for all $w \in \Sigma^*$. There are several ways to effectively represent R. Among the most popular architectures are (simple) Elman RNNs, long short-term memory (LSTM) [11], and GRUs [6]. Their expressive power depends on the exact architecture, but generally goes beyond the power of finite automata, i.e., the class of regular languages.

A *deterministic finite automaton (DFA)* over Σ is a tuple $A = (Q, \delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \colon Q \times \Sigma \to Q$ is the transition function. We assume familiarity with basic automata theory and leave it at mentioning that the language $L(A)$ of A is defined as the set of words from Σ^* that δ guides into a final state when starting in q_0. That is, for the complement DFA $\overline{A} = (Q, \delta, q_0, Q \setminus F)$, we get $L(\overline{A}) = \overline{L(A)} = \Sigma^* \setminus L(A)$. It is well-known that high-level specifications such as LTL formulas over finite words [10] or regular expressions can be compiled into corresponding DFAs.

We sometimes use RNNs and DFAs synonymous for their respective languages. For example, we say that R is ε-approximately correct wrt. A if $L(R)$ is ε-approximately correct wrt. $L(A)$.

Angluin's Algorithm. Angluin introduced L*, a classical instance of a learning algorithm in the presence of a minimally adequate teacher (MAT) [2]. We do not detail the algorithm here but only define the interfaces that we need to embed L* into our framework. Given any regular language $L \subseteq \Sigma^*$, the algorithm L* eventually outputs the unique minimal DFA $\mathcal{H}$ such that $L(\mathcal{H}) = L$. The crux is that, while Σ is given, L is a priori unknown and can only be accessed through *membership queries (MQ)* and *equivalence queries (EQ)*:

(MQ) $w \overset{?}{\in} L$ for a given word $w \in \Sigma^*$. Thus, the answer is either yes or no.

(EQ) $L(\mathcal{H}) \overset{?}{=} L$ for a given DFA $\mathcal{H}$. Again, the answer is either yes or no. If the answer is no, one also gets a counterexample word from the symmetric difference $L(\mathcal{H}) \oplus L$.

Essentially, L* asks MQs until it considers that it has a consistent data set to come up with a hypothesis DFA $\mathcal{H}$, which then undergoes an EQ. If the latter succeeds, then the algorithm stops. Otherwise, the counterexample and possibly more membership queries are used to refine the hypothesis. The algorithm provides the

```
Algorithm 1: SMC
  Input: RNN $R$, DFA $A$, $\varepsilon, \gamma \in (0,1)$
for $i = 1, \dots, \log(2/\gamma)/(2\varepsilon^2)$ do
    $w \leftarrow$ sampleWord()
    if $w \in L(R) \setminus L(A)$ then
        return "Counterexample $w$"
end
return "Property satisfied"
```

```
Algorithm 2: AAMC
  Input: RNN $R$ and DFA $A$
$A_R \leftarrow$ Approximation($R$)
if $\exists w \in L(A_R) \setminus L(A)$ then
    return "Counterexample $w$"
else return "Property satisfied"
```

```
Algorithm 3: PDV
  Input: RNN $R$, DFA $A$, $\varepsilon, \gamma \in (0,1)$
Initialize L$^*$
while true do
    $\mathcal{H} \leftarrow$ hypothesis provided by L$^*$
    Check $L(\mathcal{H}) \subseteq L(A)$
    if $L(\mathcal{H}) \subseteq L(A)$ then
        Check $L(R) \subseteq L(\mathcal{H})$ using Alg. 1
        if $L(R) \subseteq L(\mathcal{H})$ then
            return "Property satisfied"
        else Feed counterexample to L$^*$
    else
        Let $w \in L(\mathcal{H}) \setminus L(A)$
        if $w \in L(R)$ then
            return "Counterexample $w$"
        else Feed counterexample $w$ to L$^*$
    end
end
```

following guarantee: If MQs and EQs are answered according to a given regular language $L \subseteq \Sigma^*$, then the algorithm eventually outputs, after polynomially[1] many steps, the unique minimal DFA $\mathcal{H}$ such that $L(\mathcal{H}) = L$.

3 Verification Approaches

Before we present (in Section 4) our method of verifying RNNs, we here describe two simple approaches. The experiments will later compare all three algorithms wrt. their performance.

Statistical Model Checking (SMC). One obvious approach for checking whether the RNN under test R satisfies a given specification A, i.e., to check whether $L(R) \subseteq L(A)$, is by a form of random testing. The idea is to generate a finite test suite $T \subset \Sigma^*$ and to check, for each $w \in T$, whether for $w \in L(R)$ also $w \in L(A)$ holds. If not, each such w is a *counterexample*. On the other hand, if none of the words turns out to be a counterexample, the property holds on R with a certain error probability. The algorithm is sketched as Algorithm 1.

Note that the test suite is sampled according to a probability distribution on Σ^*. Recall that our choice depends on two parameters: a probability distribution on Σ and a "termination" probability, both are described in Section 2.

Theorem 1 (Correctness of SMC). *If Algorithm 1, with $\varepsilon, \gamma \in (0,1)$, terminates with "Counterexample w", then w is mistakenly classified by R as positive. If it terminates with "Property satisfied", then R is ε-approximately correct wrt. A with probability at least $1 - \gamma$.*

[1] in the index of the right congruence associated with L and in the size of the longest counterexample obtained as a reply to an EQ

While the approach works in principle, it has several drawbacks for its practical application. The size of the test suite may be quite huge and it may take a while both finding a counterexample or proving correctness.

Moreover, the correctness result and the algorithm assume that the words to be tested are chosen according to a random distribution that somehow also has to take into account the RNN as well as the property automaton.

It has been reported that this method does not work well in practice [25] and our experiments support these findings.

Automaton Abstraction and Model Checking (AAMC). As model checking is mainly working for finite-state systems, a straightforward idea would be to (a) *approximate* the RNN R by a finite automaton A_R such that $L(R) \approx L(A_R)$ and (b) to check whether $L(A_R) \subseteq L(A)$ using model checking. The algorithmic schema is depicted in Algorithm 2.

Here, we can instantiate Approximation() by the DFA-extraction algorithms from [17] or [25]. In fact, for approximating an RNN by a finite-state system, several approaches have been studied in the literature, which can be, roughly, divided into two approaches: (a) *abstraction* and (b) *automata learning.* In the first approach, the state space of the RNN is mapped to equivalence classes according to certain predicates. The second approach uses automata-learning techniques such as Angluin's L*. The approach [25] is an intertwined version combining both ideas.

Therefore, there are different instances of AAMC, varying in the approximation approach. Note that, for verification as language inclusion, as considered here, it actually suffices to learn an over-approximation A_R such that $L(R) \lesssim L(A_R)$.

While the approach seems promising at first hand, its correctness has two glitches. First, the result "Property satisfied" depends on the quality of the approximation. Second, any returned counterexample w may be *spurious*: w is a counterexample with respect to A_R satisfying A but may not be a counterexample for R satisfying A. If $w \in L(R)$, then it is indeed a counterexample, but if not, it is spurious—an indication that the approximation needs to be refined. If the automaton is obtained using abstraction techniques (such as predicate abstraction) that guarantee over-approximations, well-known principles like CEGAR [7] may be used to refine it. In the automata-learning setting, w may be used as a counterexample for the learning algorithm to improve the approximation. Repeating the latter idea suggests an interplay between automata learning and verification—and this is the idea that we follow in this paper. However, rather than starting from some approximation with a certain quality that is later refined according to the RNN and the property, we perform a direct, *property-directed* approach.

4 Property-Directed Verification of RNNs

We are now ready to present our algorithm for property-directed verification (PDV). The underlying idea is to replace the EQ in Angluin's L* algorithm with a combination of classical model checking and statistical model checking, which are used as an alternative to EQs. This approach, which we call *property-directed verification of RNNs*, is outlined as Algorithm 3 and works as follows.

After initialization of L^* and the corresponding data structure, L^* automatically generates and asks MQs to the given RNN R until it comes up with a first hypothesis DFA $\mathcal{H}$ (Line 3). In particular, the language $L(\mathcal{H})$ is consistent with the MQs asked so far.

At an early stage of the algorithm, $\mathcal{H}$ is generally small. However, it already shares some characteristics with R. So it is worth checking, using standard automata algorithms, whether there is no mismatch yet between $\mathcal{H}$ and A, i.e., whether $L(\mathcal{H}) \subseteq L(A)$ holds (Line 4). Because otherwise (Line 10), a counterexample word $w \in L(\mathcal{H}) \setminus L(A)$ is already a candidate for being a misclassified input for R. If indeed $w \in L(R)$, w is mistakenly considered positive by R so that R violates the specification A. The algorithm then outputs "Counterexample w" (Line 13). If, on the other hand, R happens to agree with A on a negative classification of w, then there is a mismatch between R and the hypothesis $\mathcal{H}$ (Line 14). In that case, w is fed back to L^* to refine $\mathcal{H}$.

Now, let us consider the case that $L(\mathcal{H}) \subseteq L(A)$ holds (Line 5). If, in addition, we can establish $L(R) \subseteq L(\mathcal{H})$, we conclude that $L(R) \subseteq L(A)$ and output "Property satisfied" (Line 8). This inclusion test (Line 6) relies on statistical model checking using given parameters $\varepsilon, \gamma > 0$ (cf. Algorithm 1). If the test passes, we have some statistical guarantee of correctness of R (cf. Theorem 1). Otherwise, we obtain a word $w \in L(R) \setminus L(\mathcal{H})$ witnessing a discrepancy between R and $\mathcal{H}$ that will be exploited to refine $\mathcal{H}$ (Line 9).

Overall, in the event that the algorithm terminates, we have the following theorem (with proof in the appendix) that assures the soundness of a returned counterexample and provides the statistical guarantees on the property satisfaction, depending on the result of the algorithm:

Theorem 2 (Correctness of PDV). *Suppose Algorithm 3 terminates, using SMC for inclusion checking with parameters ε and γ. If it outputs "Counterexample w", then w is mistakenly classified by R as positive. If it outputs "Property satisfied", then R is ε-approximately correct wrt. A with probability at least $1 - \gamma$.*

Although we cannot hope that Algorithm 3 will always terminate, we demonstrate empirically that it is an effective way for the verification of RNNs.

5 Adversarial Robustness Certification

Our method can especially be used for *adversarial robustness certification*, which is parameterized by a distance function $\mathit{dist} : \Sigma^* \times \Sigma^* \to [0, \infty]$ satisfying, for all words $w_1, w_2, w_3 \in \Sigma^*$: (i) $\mathit{dist}(w_1, w_2) = 0$ iff $w_1 = w_2$, (ii) $\mathit{dist}(w_1, w_2) = \mathit{dist}(w_2, w_1)$, and (iii) $\mathit{dist}(w_1, w_3) \leq \mathit{dist}(w_1, w_2) + \mathit{dist}(w_2, w_3)$. Popular distance functions are *Hamming distance* and *Levenshtein distance.* The Hamming distance between $w_1, w_2 \in \Sigma^*$ is the number of positions in which w_1 differs from w_2, provided $|w_1| = |w_2|$ (otherwise, the distance is ∞). The Levenshtein distance (edit distance) between w_1 and w_2 is the minimal number of operations among substitution, insertion, and deletion that are required to transform w_1 into w_2. For $L \subseteq \Sigma^*$ and $r \in \mathbb{N}$, we let $\mathcal{N}_r(L) = \{w' \in \Sigma^* \mid \mathit{dist}(w, w') \leq r$ for some $w \in L\}$ be the *r-neighborhood* of L. If L is regular and *dist* is the Hamming or Levenshtein distance, then $\mathcal{N}_r(L)$ is regular (for efficient constructions of *Levenshtein automata* when L is a singleton, see [24]).

Let R be an RNN, $L \subseteq \Sigma^*$ be a regular language such that $L \subseteq L(R)$, $r \in \mathbb{N}$, and $0 < \varepsilon < 1$. We call R *ε-adversarially robust* (wrt. L and r) if $\Pr(\mathcal{N}_r(L) \backslash L(R)) < \varepsilon$. Accordingly, every word from $\mathcal{N}_r(L) \backslash L(R)$ is an *adversarial example*. Thus, checking adversarial robustness amounts to checking the inclusion $L(\overline{R}) \subseteq \overline{\mathcal{N}_r(L)}$ through one of the above-mentioned algorithms.

Note that, even when L is a finite set, $\mathcal{N}_r(L)$ can be too large for exhaustive exploration so that PDV, in combination with SMC, is particularly promising, as we demonstrate in our experimental evaluation.

From the definitions and Theorem 2, we get:

Lemma 1. *Suppose Algorithm 3, for input $\overline{R}$ and a DFA A recognizing $\overline{\mathcal{N}_r(L)}$, terminates, using SMC for inclusion checking with parameters ε and γ. If it outputs "Counterexample w", then w is an adversarial example. Otherwise, R is ε-adversarially robust (wrt. L and r) with probability at least $1 - \gamma$.*

Similarly, we can handle the case where $L \cap L(R) = \emptyset$. Then, R is *ε-adversarially robust* if $\Pr(L(R) \cap \mathcal{N}_r(L)) < \varepsilon$, and every word in $L(R) \cap \mathcal{N}_r(L)$ is an *adversarial example*. Overall, this case amounts to checking $L(R) \subseteq \overline{\mathcal{N}_r(L)}$.

6 Experimental Evaluation

We now present an experimental evaluation of the three algorithms SMC, AAMC, and PDV, and provide a comparison of their performance on LSTM networks [11] (a variant of RNNs using LSTM units). The algorithms have been implemented[2] in Python 3.6 using PyTorch 19.09 and Numpy library. The experiments of adversarial robustness certification were run on Macbook Pro 13 with the macOS. The other experiments were run on NVIDIA DGX-2 with an Ubuntu OS.

Optimization For Equivalence Queries. In [17], the authors implement AAMC but with an optimization that was originally shown in [2]. This optimization concerns the number of samples required for checking the equivalence between the hypothesis and the taught language. This number depends on ε, γ and the number of previous equivalence queries n and is calculated by $\frac{1}{\varepsilon}\left(\log \frac{1}{\gamma} + \log(2)(n+1)\right)$. We adopt this optimization in AAMC and PDV as well (Algorithm 2 in Line 1 and Algorithm 3 in Line 6).

6.1 Evaluation on Randomly Generated DFAs

Synthetic Benchmarks. To compare the algorithms, we implemented the following procedure, which generates a random DFA A_{rand}, an RNN R that learned

[2] available at `https://github.com/LeaRNNify/Property-directed-verification`

Table 1. Experimental results

Type	*Avg time* (s)	*Avg len*	*#Mistakes*	*Avg MQs*
SMC	92	111	**122**	286063
AAMC	444	**7**	30	3701916
PDV	**21**	11	109	**28318**

$L(A_{\mathsf{rand}})$, and a finite set of specification DFAs: (1) choose a random DFA $A_{\mathsf{rand}} = (Q, \delta, q_0, F)$, with $|Q| \leq 30$, over an alphabet Σ with $|\Sigma| = 5$; (2) randomly sample words from Σ^* as described in Section 2 in order to create a training set and a test set; (3) train an RNN R with hidden dimension $20|Q|$ and $1 + |Q|/10$ layers—if the accuracy of R on the training set is larger than 95%, continue, otherwise restart the procedure; (4) choose randomly up to five sets $F_i \subseteq Q \setminus F$ to define specification DFAs $A_i = (Q, \delta, q_0, F \cup F_i)$. Using this procedure, we created 30 DFAs/RNNs and 138 specifications.

Experimental Results. Given an RNN R and a specification DFA A, we checked whether R satisfies A using Algorithms 1–3, i.e., SMC, AAMC, and PDV, with $\varepsilon, \gamma = 5 \cdot 10^{-4}$.

Table 1 summarizes the executions of the three algorithms on our 138 random instances. The columns of the table are as follows: (i) *Avg time* was counted in seconds and all the algorithms were timed out after 10 minutes; (ii) *Avg len* is the average length of the found counterexamples (if one was found); (iii) *#Mistakes* is the number of random instances for which a mistake was found; (iv) *Avg MQs* is the average number of membership queries asked to the RNN.

Note that not only is PDV faster and finds more errors than AAMC, the average number of states of the final DFA is also much smaller: **26** states with PDV and **319** with AAMC. Furthermore, it asked more than 10 times less MQs to the RNN. Comparing PDV to SMC, it is 4.5 times faster and the average length of counterexamples it found is 10 times smaller, even though with a little fewer mistakes discovered.

Faulty Flows. One of the advantages of extracting DFAs in order to detect mistakes in a given RNN is the possibility to find not only one mistake but a “faulty flow”. For example, Figure 1 shows one hypothesis DFA extracted with PDV, based on which we found a mistake in the corresponding RNN. The counterexample we found was *abcee*. One can see that the word *abce* is a loop in the DFA. Hence, we can suspect that this could be a “faulty flow”. Checking the words $w_n = (abce)^n e$ for $n \in \{1, \ldots, 100\}$, we observed that, for any $n \in \{1, \ldots, 100\}$, the word w_n was in the RNN language but not in the specification.

To automate the reasoning above, we did the following: Given an RNN R, a specification A, the extracted DFA $\mathcal{H}$, and the counterexample w: (1) build the cross product DFA $\mathcal{H} \times \overline{A}$; (2) for every prefix w_1 of the counterexample $w = w_1 w_2$, denote by s_{w_1} the state to which the prefix w_1 leads in $\mathcal{H} \times \overline{A}$—for any loop ℓ starting from s_{w_1}, check if $w_n = w_1 \ell^n w_2$ is a counterexample for $n \in \{1, \ldots, 100\}$; (3) if w_n is a counterexample for more than 20 times, declare a

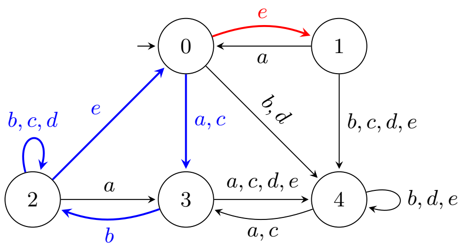

Fig. 1. Faulty Flow in DFA extracted through PDV

"faulty flow". Using this procedure, we managed to find faulty flows in 81/109 of the counterexamples that were found by PDV.

6.2 Adversarial Robustness Certification

We also examined PDV for adversarial robustness certification, following the ideas explained in Section 5, both on synthetic as well as real-world examples.

Synthetic Benchmarks. For a given DFA (representing one of the languages described below), we randomly sampled words from Σ^* by using the DFA and created a training set and a test set. For RNN training, we proceeded like in step (3) for the benchmarks in Section 6.1. Moreover, for certification, we randomly sampled 100 positive words and 100 negative words from the test set. For a given word w, we then let $L = \{w\}$ and considered $\mathcal{N}_r(L)$ where $r = 1, \ldots, 5$.

Given an RNN R, we checked whether R satisfies adversarial robustness using the certification methods PDV, SMC, and *neighborhood-automata generation SMC (NAG-SMC)*, with $\varepsilon, \gamma = 0.01$. In SMC, we randomly modified the input word within a certain distance to generate words in the neighborhood. In NAG-SMC, on the other hand, we first generated a neighborhood automaton of the input word, and sampled words that are accepted by the automaton. Here, we followed the algorithm by Bernardi and Giménez [5], who introduce a method for generating a uniformly random word of length n in a given regular language with mean time bit-complexity $O(n)$.

Figure 2, which is a set of scatter plots, shows the results of the average time of executing the algorithms on the languages that we describe below. The x-axis and y-axis are both time in seconds, and each data point represents one adversarial robustness certification procedure. The length of words are from 50 to 500 and follow the normal distribution.

Simple Regular Ranguages. As a sanity check of our approach, we considered the following two regular languages and distance functions: $L_1 = ((a+b)(a+b))^*$ (also called *modulo-2 language*) with Hamming distance; $L_2 = c(a+b)^*c$ with distance function *dist* such that $dist(w_1, w_2)$ is the Hamming distance if $w_1, w_2 \in L_2$ and $|w_1| = |w_2|$, and $dist(w_1, w_2) = \infty$ otherwise. The size of the Hamming neighbourhood will exponentially grow with the distance.

The accuracies of the trained RNNs reached 100%. All three approaches successfully reported "adversarially robust" for the certified RNNs.

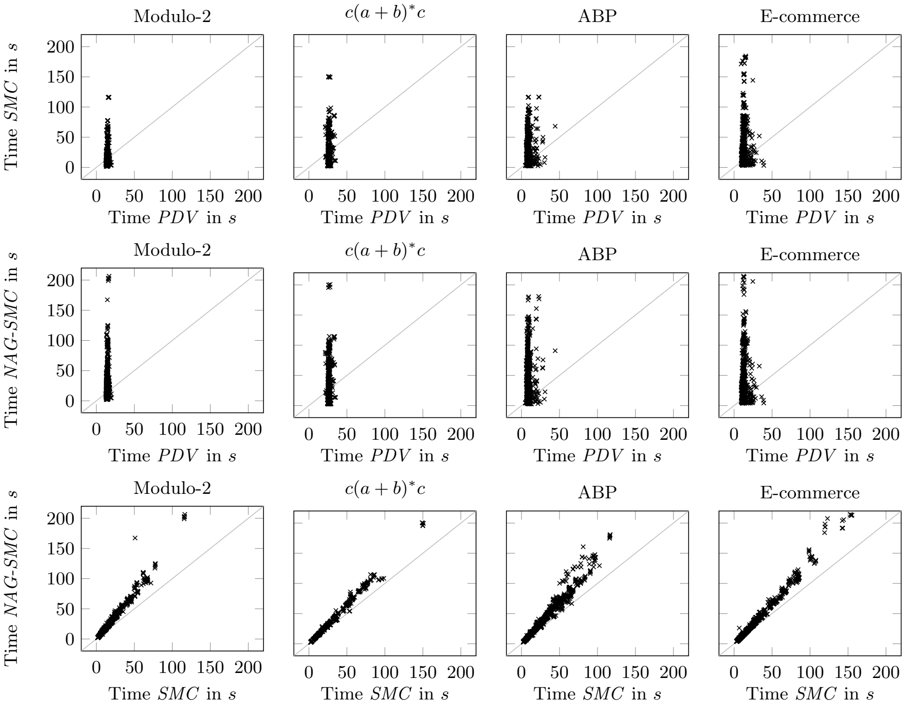

Fig. 2. Comparison of three algorithms on the regular languages

The first two diagrams on the first row of Figure 2 compare the runtimes of PDV and SMC on the two regular-language datasets, resp., whereas the first two diagrams on the second row compare the runtimes of PDV and NAG-SMC. We make two main observations. First, on average, the running time of PDV (avg. 15.70 seconds) is faster than SMC (avg. 24.04 seconds) and NAG-SMC (avg. 32.5 seconds), which shows clearly that combining symbolically checking robustness on the extracted model and statistical approximation checking is more efficient than pure statistical approaches. Second, although SMC and NAG-SMC are able to certify short words (whose length is smaller than 30) faster, when the length of words is greater, they have to spend more time (which is more than 60 seconds) for certification. This is because, for short words, statistical approaches can easily explore the whole neighborhood, but when the neighborhood becomes larger and larger, this becomes infeasible.

The first two diagrams on the third row of Figure 2 compare the running time of SMC and NAG-SMC, respectively. In general, SMC is faster than NAG-SMC, this is mainly because, for sampling random words from the neighborhood, using the algorithm proposed by Bernardi et al. [5] is slower than combining the *random.choice* function in the Python library and the corresponding modification.

Real-World Dataset. We used two real-world examples considered by Mayr and Yovine [17]. The first one is the alternating-bit protocol (ABP) shown in Figure 3. However, we add a special letter *dummy* in the alphabet and a self-loop transition

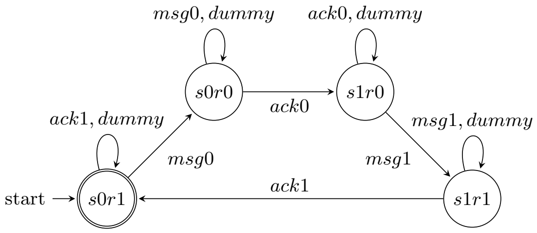

Fig. 3. Automaton for ABP

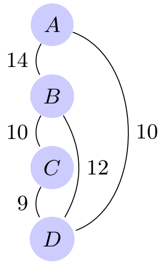

Fig. 4. Temporal Network for contact between 4 people

labeled with *dummy* on every state. We use the number of insertions of the letter *dummy* as the distance function. The second example is a variant of an example from the E-commerce website [19]. There are seven letters in the original automaton. Similarly, we also add *dummy* and self-loop transition in every state (omitted in the figure for simplicity). Again, we use the number of insertions of *dummy* as the distance function.

The accuracies of the trained RNNs also reach 100%. For certification, the three approaches can certify the adversarial robustness for the RNNs as well.

The last two diagrams on the first (resp. second) row of Figure 2 compare the runtime of PDV and SMC (resp. PDV and NAG-SMC) on the ABP and the E-commerce dataset. The data points in the first and second row have a vertical shape. The reason is that the running time of PDV is usually relatively stable (10–20 seconds), while the running time of SMC and NAG-SMC increases linearly with the word length.

The last two diagrams on the third row of Figure 2 compare the runtimes of SMC and NAG-SMC on the two datasets. Here, the data points have a diagonal shape, but for NAG-SMC, when the word length is long (more than 300), it usually spends more time than SMC. This is mainly because it is inefficient to construct the neighborhood automaton and sample random words from the neighborhood.

6.3 RNNs Identifying Contact Sequences

Contact tracing [14] has proven to be increasingly effective in curbing the spread of infectious diseases. In particular, analyzing contact sequences—sequences of individuals who have been in close contact in a certain order—can be crucial in identifying individuals who might be at risk during an epidemic. We, thus, look at RNNs which can potentially aid contact tracing by identifying possible contact sequences. However, in order to deploy such RNNs in practice, one would require them to be verified adequately. One does not want to alert individuals unnecessarily even if they are safe or overlook individuals who could be at risk.

In a real-world setting, one would obtain contact sequences from contact-tracing information available from, for instance, contact-tracing apps. However, such data is often difficult to procure due to privacy issues. Thus, in order to mimic a real life scenario, we use data available from `www.sociopatterns.org`, which contains information about interaction of humans in public places (hospitals, schools, etc.) presented as temporal networks.

Formally, a *temporal network* $G = (V, E)$ [12] is a graph structure consisting of a set of vertices V and a set of labeled edges E, where the labels represent the timestamp during which the edge was active. Figure 4 is a simple temporal network, which can be perceived as contact graph of four workers in an office where edge labels represent the time of meeting between them. A *time-respecting path* $\pi \in V^*$—a sequence of vertices such that there exists a sequence of edges with increasing time labels—depicts a contact sequence in such a network. In the above example, $CDAB$ is a time-respecting path while $ABCD$ is not.

Benchmarks. For our experiment, given a temporal network G, we generated an RNN R recognizing contact sequences as follows:

1. We created training and test data for the RNN by generating (i) valid time-respecting paths (of lengths between 5 and 15) using labeled edges from G, and (ii) invalid time-respecting paths, by considering a valid path and randomly introducing breaks in the path. The number of time-respecting paths in the training set is twice the size of the number of labeled edges in G, while the test set is one-fifth the size of the training set.
2. We trained RNN R with hidden dimension $|V|$ (minimum 100) as well as $\lfloor 2 + |V|/100 \rfloor$ layers on the training data. We considered only those RNNs that could be trained within 5 hours with high accuracy (avg. 99%) on the test data.
3. We used a DFA that accepts all possible paths (disregarding the time labels) in the network as the specification, which would allow us to check whether the RNN learned unwanted edges between vertices.

Using this process, from the seven temporal networks, we generated seven RNNs and seven specification DFAs. We ran SMC, PDV, and AAMC on the generated RNNs, using the same parameters as used for the random instances.

Results. Table 2 notes the length of counterexample, the extracted DFA size (only for PDV and AAMC), and the running time of the algorithms. We make three main observations. First, the counterexamples obtained by PDV and AAMC (avg. length 2), are much more succinct than those by SMC (avg. length 13.1). Small counterexamples help in identifying the underlying error in the RNN, while long and random counterexamples provide much less insight. For example, from the counterexamples obtained from PDV and AAMC, we learned that the RNN overlooked certain edges or identified wrong edges. This result highlights the demerit of SMC, which has also been observed by [25]. Second, the running time of SMC and PDV (avg. 0.48 seconds and 0.41 seconds) is comparable, while that of AAMC is prohibitively large (avg. 655.68 seconds), indicating that model checking on small and rough abstractions of the RNN produces superior results. Third, the extracted DFA size, in case of AAMC (avg. size 124.14), is always larger compared to PDV (avg. size 2), indicating that RNNs are quite difficult

Table 2. Results of model-checking algorithm on RNN identifying contact sequences

		Counter-	*Extracted*	
Case	*Alg.*	*example len.*	*DFA size*	*Time* (s)
Across	SMC	3		0.3
Kenyan	AAMC	2	328	624.76
Household	PDV	2	2	0.22
	SMC	2		0.23
Workplace	AAMC	2	111	604.99
	PDV	2	2	0.77
	SMC	5		0.33
Highschool	AAMC	2	91	627.30
2011	PDV	2	2	0.19
	SMC	7		0.24
Hospital	AAMC	2	36	614.76
	PDV	2	2	0.006
Within	SMC	2		0.28
Kenyan	AAMC	2	178	620.30
Household	PDV	2	2	0.27
	SMC	71		1.51
Conference	AAMC	2	38	876.19
	PDV	2	2	0.33
	SMC	3		0.48
Workplace	AAMC	2	87	621.44
2015	PDV	2	2	1.11

to be approximated by small DFAs and this slows down the model-checking process as well. Again, our experiments confirm that PDV produces succinct counterexamples reasonably fast.

7 Conclusion

We proposed property-directed verification (PDV) as a new verification method for formally verifying RNNs with respect to regular specifications, with adversarial robustness certification as one important application. It is straightforward to extend our ideas to the setting of Moore/Mealy machines supporting the setting of richer classes of RNN classifiers, but this is left as part of future work. Another future work is to investigate the applicability of our approach for RNNs representing more expressive languages, such as context-free ones. Finally, we plan to extend the PDV algorithm for the formal verification of RNN-based agent environment systems, and to compare it with the existing results.

References

1. Akintunde, M.E., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of rnn-based neural agent-environment systems. In: Proceedings of AAAI 2019. pp. 6006–6013. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33016006
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Ayache, S., Eyraud, R., Goudian, N.: Explaining black boxes on sequential data using weighted automata. In: Proceedings of ICGI 2018. Proceedings of Machine Learning Research, vol. 93, pp. 81–103. PMLR (2018)
4. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
5. Bernardi, O., Giménez, O.: A linear algorithm for the random sampling from regular languages. Algorithmica **62**(1-2), 130–145 (2012)
6. Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proc. EMNLP. pp. 1724–1734. ACL (2014)

7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of CAV 2000. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000)
8. Du, X., Li, Y., Xie, X., Ma, L., Liu, Y., Zhao, J.: Marble: Model-based robustness analysis of stateful deep learning systems. In: ASE 2020. pp. 423–435. IEEE (2020)
9. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Proceedings of CAV 2020, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 43–65. Springer (2020)
10. Giacomo, G.D., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: Proceedings of IJCAI 2015. pp. 1558–1564. AAAI Press (2015)
11. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
12. Holme, P.: Temporal networks. In: Encyclopedia of Social Network Analysis and Mining, pp. 2119–2129. Springer (2014)
13. Jacoby, Y., Barrett, C.W., Katz, G.: Verifying recurrent neural networks using invariant inference. CoRR **abs/2004.02462** (2020)
14. Keck, C.: Principles of Public Health Practice. Cengage Learning (2002)
15. Kwiatkowska, M.Z.: Safety Verification for Deep Neural Networks with Provable Guarantees (Invited Paper). In: Proceedings of CONCUR 2019. Leibniz International Proceedings in Informatics (LIPIcs), vol. 140, pp. 1:1–1:5. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019)
16. Mayr, F., Visca, R., Yovine, S.: On-the-fly black-box probably approximately correct checking of recurrent neural networks. In: Proceedings of CD-MAKE 2020. Lecture Notes in Computer Science, vol. 12279, pp. 343–363. Springer (2020)
17. Mayr, F., Yovine, S.: Regular inference on artificial neural networks. In: Holzinger, A., Kieseberg, P., Tjoa, A.M., Weippl, E.R. (eds.) Proceedings of CD-MAKE 2018. LNCS, vol. 11015, pp. 350–369. Springer (2018)
18. Mayr, F., Yovine, S., Visca, R.: Property checking with interpretable error characterization for recurrent neural networks. Mach. Learn. Knowl. Extr. **3**(1), 205–227 (2021)
19. Merten, M.: Active automata learning for real life applications. Ph.D. thesis, Dortmund University of Technology (2013)
20. Okudono, T., Waga, M., Sekiyama, T., Hasuo, I.: Weighted automata extraction from recurrent neural networks via regression on state spaces. In: Proceedings of AAAI 2020. pp. 5306–5314. AAAI Press (2020)
21. Omlin, C.W., Giles, C.L.: Extraction of rules from discrete-time recurrent neural networks. Neural Networks **9**(1), 41–52 (1996)
22. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. Journal of Automata, Languages and Combinatorics **7**(2), 225–246 (2002)
23. Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A.M., Vechev, M.T.: Fast and effective robustness certification for recurrent neural networks. CoRR **abs/2005.13300** (2020)
24. Schulz, K.U., Mihov, S.: Fast string correction with levenshtein automata. Int. J. Document Anal. Recognit. **5**(1), 67–85 (2002)
25. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. In: Proceedings of ICML 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5244–5253. PMLR (2018)