The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)

Verification of RNN-Based Neural Agent-Environment Systems

Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, Edoardo Pirovano
Department of Computing,
Imperial College London, UK

Abstract

We introduce agent-environment systems where the agent is stateful and executing a ReLU recurrent neural network. We define and study their verification problem by providing equivalences of recurrent and feed-forward neural networks on bounded execution traces. We give a sound and complete procedure for their verification against properties specified in a simplified version of LTL on bounded executions. We present an implementation and discuss the experimental results obtained.

1 Introduction

A key obstacle in the deployment of autonomous systems is the inherent difficulty of their verification. This is because the behaviour of autonomous systems is hard to predict; in turn this makes their certification a challenge.

Progress has been made over the past few years in the area of verification of multi-agent systems where a number of methods based on model checking and theorem proving have been put forward (Alechina et al. 2010; Lomuscio, Qu, and Raimondi 2017; Kouvaros, Lomuscio, and Pirovano 2018). Some of this work has been combined with safety analysis and abstraction thereby resulting in the assessment of designs such as autonomous underwater vehicles (Ezekiel et al. 2011).

A key assumption made in the literature on autonomous systems is that their designs have been traditionally implemented by engineers. But parts of AI systems are increasingly realised by machine learning, thereby making present verification approaches not applicable. For example, vision systems currently being tested in autonomous vehicles make use of special classes of feed-forward neural networks to classify the environment at runtime (Redmon et al. 2016). It follows that, to be able to verify and certify forthcoming autonomous systems, methods for the verification of systems based on neural networks are urgently required.

In this paper we develop a novel method for the verification of closed-loop systems of one agent, based on a neural network, interacting with an environment. Specifically, we study systems in which the agent is memoryful and realised via a previously trained recurrent neural network (Hausknecht and Stone 2015). As we discuss below, we are not aware of other methods in the literature addressing the formal verification of systems based on recurrent networks. The present contribution focuses on reachability analysis, that is we intend to give formal guarantees as to whether a state, or a set of states are reached in the system. Typically this is an unwanted state, or a bug, that we intend to ensure is not reached in any possible execution. Similarly, to provide safety guarantees, this analysis can be used to show the system does not enter an unsafe region of the state space.

The rest of the paper is organised as follows. After the discussion of related work below, in Section 2, we introduce the basic concepts related to neural networks that are used throughout the paper. In Section 3 we define Recurrent Neural Agent-Environment Systems and define their verification decision problem. In Section 4 we show that for verification purposes recurrent networks can be simulated by simpler feed-forward networks as long as the analysis is on bounded executions. We exploit this in Section 5 where we provide a complete (up to bounded executions) procedure for the verification of recurrent neural agent-environment systems. We give details of an implementation of the approach in Section 6, where we benchmark the procedures previously introduced.

Related Work. As mentioned above, there is a large literature on verification of agent-based systems. With the exceptions mentioned below, differently from this paper, this body of work deals with agents that are designed by using declarative or procedural programming languages. Therefore their aims and techniques are different from ours.

There has been recent work on the formal verification of feed-forward networks. This was first explored in (Pulina and Tacchella 2010), which addressed sigmoid activation functions, and could be scaled up to approximately 20 neurons. More recently, (Katz et al. 2017; Lomuscio and Maganti 2017; Ehlers 2017; Bunel et al. 2017; Gehr et al. 2018; Ruan, Huang, and Kwiatkowska 2018) considered ReLU activation functions, as we do here, but differently from the present contribution, all previous analyses are limited to feed-forward networks and purely reachability conditions. Instead, our contribution focuses on systems based on recurrent neural networks specified in a restricted version of LTL. Further recent work on verification of neural networks focuses on adversarial examples, e.g., (Huang et al. 2017), and has different aims from those here pursued.

Much closer to the work here presented is (Akintunde et al. 2018), where a verification method for reachability in closed-loop systems controlled by a ReLU feed-forward network is presented. In contrast with this work, we here focus on recurrent neural networks, which require a different theoretical treatment. We take up the model introduced there of a neural agent-environment system, and modify it by introducing agents based on recurrent networks. The implementation here presented uses the module released in (Akintunde et al. 2018) for the verification of feed-forward networks, but extends it by supporting agents defined on recurrent networks as well.

2 Preliminaries

In this section we introduce our notation for neural networks that will be used throughout the rest of the paper. We assume the basic definition of a neural network and related notions (Haykin 1999).

Feed-forward neural networks. Feed-forward neural networks are a simple class of neural networks consisting of multiple *hidden layers* and admitting no cycles. The presentation here loosely follows that in (Akintunde et al. 2018).

Definition 1. An n-layer *feed-forward multilayer neural network* (FFNN) N is a neural network resulting from the composition of n layers, each denoted by $L^{(i)}$, for $1 \leq i \leq n$. Each layer $L^{(i)}$ is defined by a *weight matrix* $W^{(i)}$, a *bias vector* $b^{(i)}$, and an *activation function* $\sigma^{(i)}$. We refer to $L^{(1)}$ as the *input layer* of the network, and $L^{(n)}$ as the *output layer*. Any additional layers which lie between these two layers are referred to as *hidden layers*.

Each layer of the network is composed of *nodes*, in which computations combine the output of nodes in the previous layer to produce an output. This output is then used in the computations of nodes in successive layers.

We only consider *fully-connected* neural networks where all the nodes in each layer have a connection to every node in the adjacent layers (with the exception of the input and output layers $L^{(1)}$ and $L^{(n)}$, which intuitively are only connected to layers $L^{(2)}$ and $L^{(n-1)}$ respectively).

Each node in each hidden layer has an associated *activation function*, which applies a transformation to a linear combination of the *values* of the input (incoming) nodes. This quantity defines the *value* of the node, which is then passed into a subsequent node.

We only consider networks with hidden layers utilising *Rectified Linear Unit (ReLU)* activation functions, defined by $\text{ReLU}(x) \triangleq \max(0, x)$, where x represents the linear combination of values from incoming nodes in the previous layer. Here the value of the node is the output of the ReLU function. ReLU activation functions are widely used and are known to allow FFNNs to generalise well to unseen inputs (Nair and Hinton 2010).

Neural networks are generally used to learn highly non-linear, non-programmatic functions; upon training, the network computes an approximation of such a function by means of the weights and biases learned.

Definition 2 (Function computed by FFNN)**.** Let N be a FFNN. For each layer $L^{(i)}$ of N, let c and d denote respectively the number of inputs and output nodes of layer i. We define the *computed function* for $L^{(i)}$, denoted $f^{(i)} : \mathbb{R}^c \to \mathbb{R}^d$, by $f^{(i)}(x) = \sigma^{(i)}(W^{(i)}x + b^{(i)})$. Further, for an m-layer FFNN N with p input nodes and q output nodes, the computed function for N as a whole, $f : \mathbb{R}^p \to \mathbb{R}^q$, is defined as $f(x) = f^{(m)}(f^{(m-1)}(\ldots(f^{(2)}(x))))$.

Recurrent neural networks. Recurrent neural networks (RNNs) are known to be difficult to train and whose behaviour is difficult to predict. Differently from FFNNs, whose output is always the same given a certain single input, RNNs were designed to process sequences of data, and are equipped with a state that evolves over time. As a result, the behaviour of an RNN depends on the history of their inputs. We formalise this below.

Definition 3. A single-layer *recurrent neural network* (RNN) R with h *hidden units* and input size i and output size o is a neural network associated with the weight matrices $W_{i\to h} \in \mathbb{R}^{i\times h}$, $W_{h\to h} \in \mathbb{R}^{h\times h}$ and $W_{h\to o} \in \mathbb{R}^{h\times o}$, and the two activation functions $\sigma : \mathbb{R}^h \to \mathbb{R}^h$ and $\sigma' : \mathbb{R}^o \to \mathbb{R}^o$.

The weight matrix $W_{i\to h}$ defines the weights of the connections between input units and hidden units, $W_{h\to h}$ defines the weights of the connections between hidden units, and $W_{h\to o}$ defines the weights of the connections between hidden units and output units. We refer to the value of the hidden units as the *state* of the network, due to the self-loop introduced by $W_{h\to h}$. The activation function σ is applied to the state before it is passed onto the next time step, and σ' is applied to the state of the network at a given time step, in order to compute the output of the network at that time step. In what follows, we only consider ReLU activation functions for σ and σ'.

Definition 4 (Function computed by RNN)**.** Let N be an RNN with weight matrices $W_{i\to h}$, $W_{h\to h}$ and $W_{h\to o}$. Let $\bar{x} \in (\mathbb{R}^k)^n$ denote an input sequence of length n where each element of the sequence is a vector of size k, with $\bar{x}_t$ denoting the t-th vector of $\bar{x}$. We define $h_0^{\bar{x}} = \bar{0}$ as a vector of 0s. For each time step $1 \leq t \leq n$, we define:

$$h_t^{\bar{x}} = \sigma(W_{h\to h}h_{t-1}^{\bar{x}} + W_{i\to h}\bar{x}_t).$$

Then, the *output* of the RNN is given by $f(\bar{x}) = \sigma'(W_{h\to o}h_n^{\bar{x}})$.

The ability of RNNs to recall information from previous time steps has traditionally made them useful in language-related tasks such as text prediction (Sutskever, Martens, and Hinton 2011). However, since they define a stateful component, they can be taken as a basis for a stateful agent as we do in the next section.

3 Recurrent Neural Agent-Environment Systems

We introduce recurrent neural agent-environment systems (RNN-AES). RNN-AESs are an extension of the neural agent-environment systems introduced in (Akintunde et al. 2018) which are defined on feed-forward neural networks instead.

An RNN-AES is a closed-loop system comprising an environment and an agent. The environment is stateful and updates its state in response to the actions of the agent. The agent is implemented by an RNN and chooses the next action on the basis of a sequence of (possibly incomplete) observations of the environment. We begin by defining the *environment*.

Definition 5 (Environment)**.** An *environment* is a tuple $E = (S, O, o, t_E)$, where:

- S is a set of states of the environment,
- O is a set of observations of the environment,
- $o : S \to O$ is an environment observation function that given an environment state returns an observation of it that agents can access,
- $t_E : S \times Act \to S$ is a transition function which given the current state of the environment and an *action* performed by the *agent* returns the next state of the environment.

Notice that if we wish to have a fully observable environment, we can take $O = S$ and $o = id$, but the above framework also allows us to have a partially observable environment by taking some $O \neq id$.

We assume that the environment's observation and transition function are linearly-definable. If they are not, they can be linearly approximated to an arbitrary level of precision. See, e.g., (Akintunde et al. 2018; D'Ambrosio, Lodi, and Martello 2010) for a discussion on this.

We now proceed to define a recurrent neural agent, which performs actions on the environment on the basis of its past observations of it, and the current observation.

Definition 6 (Recurrent Neural Agent)**.** A recurrent neural agent, or simply an *agent*, denoted Agt_N, acting on an environment E is defined by an *action* function $act : O^* \to Act$, which given a finite sequence of environment observations from $O \subseteq \mathbb{R}^m$ returns an action from a set $Act = \mathbb{R}^n$ of admissible actions for the agent. The function act is implemented by a ReLU-RNN N with m inputs and n outputs computing a function $f : (\mathbb{R}^m)^* \to \mathbb{R}^n$, i.e. $act(e) = f(e)$.

We now proceed to define a *recurrent neural agent-environment system*, which is a closed-loop system of an environment composed with an agent acting on it.

Definition 7 (RNN-AES)**.** A *recurrent neural agent-environment system* (RNN-AES) is a tuple $AES = (E, Agt_N, I)$ where:

- $E = (S, O, o, t_E)$ is an environment with corresponding state space S, observation space O, observation function o and transition function t_E,
- Agt_N is a recurrent neural agent with corresponding action function $act : O^* \to Act$,
- $I \subseteq S$ is a set of initial states for the environment.

We will assume that the set of initial states for the environment is linearly definable, or has been linearly approximated to a suitable degree of accuracy.

An RNN-AES is a general model that can be used to encode agents implemented by RNNs interacting with an environment. For example, recurrent controllers in power-plant scenarios can be modelled in this framework.

We would like to verify RNN-AESs against temporal specifications. To do so, we first describe how the system evolves at each time step.

Definition 8 (System evolution)**.** Given an RNN-AES system $AES = (E, Agt_N, I)$, we say that AES *evolves* to state $y \in S$ from initial state $x \in S$ after $n \in \mathbb{N}$ steps if $t_E^{(n)}(x) = y$ where $t_E^{(n+1)}(x) = t_E(t_E^{(n)}(x), f(o(t_E^{(0)}(x)) \dots o(t_E^{(n)}(x))))$ for $n \geq 1$ and $t_E^{(0)}(x) = x$ denotes the repeated application of the transition function t_E, and where f denotes the neural action function of the agent Agt_N.

We use Π to denote the set of all infinite paths. For a path $\rho \in \Pi$, we use $\rho(i)$ to denote the ith state in ρ and ρ_i to denote the subset of the path obtained by omitting the first i states.

We now introduce the specification language that we will use to reason about RNN-AESs. Our specifications are inspired by Linear Temporal Logic (Pnueli 1977) and are indexed by a natural number representing the number of time steps up to which the temporal modalities need to be satisfied.

Definition 9 (Specifications)**.** For an environment with state space $S = \mathbb{R}^m$, the specifications ϕ are defined by the following BNF.

$$\begin{aligned}\phi &::= X^k\mathcal{C} \mid \mathcal{C}U^{\leq k}\mathcal{C} \\ \mathcal{C} &::= \mathcal{C} \vee \mathcal{C} \mid x \leq y \mid x \geq y \mid x = y \\ &\qquad \mid x \neq y \mid x < y \mid x > y\end{aligned}$$

for $x, y \in \mathbb{R} \cup \{x_0, \dots, x_m\}$ and $k \in \mathbb{N}$.

The temporal formula $X^k\mathcal{C}$ is read as "after k time steps it is the case that $\mathcal{C}$". The formula $\mathcal{C}_1 U^{\leq k}\mathcal{C}_2$ is read as "$\mathcal{C}_2$ holds within k time steps and $\mathcal{C}_1$ holds up to then". A constraint $\mathcal{C}$ holds at a given time step if the state of the environment at that time step satisfies it. Notice we can have a disjunction of constraints but not a conjunction since our procedure will need the negation of the constraint to be linearly definable.

Observe that we use linear constraints on the state of the environment as the only atomic propositions and do not have nesting of temporal operators. Given a constraint $\mathcal{C}$, we will use $\bar{\mathcal{C}}$ to denote the constraint obtained by negating it (this will be a conjunction of linear constraints).

For example, consider an environment with state space $S = [0, 100]$ giving the position of a cart on a track. We can express the specification that after 5 time steps the cart will be at a distance of at most 10 from the start of the track by the formula $X^5(x_0 \leq 10)$.

We now define the satisfaction relation.

Definition 10 (Satisfaction)**.** Given a path $\rho \in \Pi$ on an RNN-AES and a formula ϕ, the satisfaction relation $\models$ is defined as follows (we omit all but one of the cases for constraints since they are all similar):

$\rho \models \mathcal{C}_1 \vee \mathcal{C}_2$	iff	$\rho \models \mathcal{C}_1$ or $\rho \models \mathcal{C}_2$;
$\rho \models x \leq y$	iff	the state ρ_0 satisfies the constraint $x \leq y$;
$\rho \models X^k \mathcal{C}$	iff	$\rho(k) \models \mathcal{C}$;
$\rho \models \mathcal{C}_1 U^{\leq k} \mathcal{C}_2$	iff	there is some $i \leq k$ such that $\rho(i) \models \mathcal{C}_2$ and $\rho(j) \models \mathcal{C}_1$ for all $j < i$.

We say that an agent environment system AES satisfies a formula ϕ if it is the case that every path originating from an initial state $i \in I$ satisfies ϕ. We denote this by $AES \models \phi$. This is the basis of the model checking problem that we consider in this paper, which we formalise below.

Definition 11 (Model Checking)**.** Given an RNN-AES AES and a formula ϕ, determine if it is the case that $AES \models \phi$.

The remainder of the paper will be dedicated to obtaining a sound and complete verification procedure for this decision problem.

4 Unrolling of RNNs

Agents in RNN-AESs are stateful and implemented via RNNs (see Definition 3). To provide a method for the verification of RNN-AESs against the temporal language introduced in the previous section, we show an equivalence result with FFNNs on finite paths. This enables us to frame verification of RNN-AESs in terms of reachability properties of specific FFNNs. Once this is in place we will be able to exploit existing results on the verification of FFNNs (Katz et al. 2017; Ehlers 2017; Lomuscio and Maganti 2017).

At the essence of our method is the notion of *unrolling*. Specifically, we unroll the recurrent loop in an RNN into an FFNN and show the latter is equivalent to the former on paths whose length is dependent on the unrolling considered.

We consider two similar but distinct ways of unrolling the RNN: Input on Start and Input on Demand.

Input on Start (IOS). In this method, all the input values are initially scaled according to the weights of the $W_{(i\to h)}$ matrix. Then, at the time step when the input is needed it is passed unchanged to the corresponding hidden unit.

Before defining this construction, we introduce some notation. We denote by I_a the $a \times a$ identity matrix, by $O_{a,b}$ the $a \times b$ zero matrix, and by $A \otimes B$ is the Kronecker product (Brewer 1978) of matrices A and B defined by:

$$\begin{aligned}(A \otimes B)_{i,j} =& A_{\lfloor (i-1)/p \rfloor+1, \lfloor (j-1)/q \rfloor +1} \\ &\cdot B_{(i-1)\%p+1,(j-1)\%q+1},\end{aligned}$$

where $\lfloor a \rfloor$ defines the floor of a and $a\%b$ represents the remainder of a/b.

Now, we can proceed to define the construction of our unrolling. This is shown visually in Figure 1a. We formalise the construction in the definition below.

Definition 12. Let R be an RNN with weight matrices $W_{(i\to h)}$, $W_{(h\to h)}$ and $W_{(h\to o)}$. Let $n \in \mathbb{Z}^+$ be the length of a sequence of inputs. Then, the FFNN R_n, called the length n unrolling of R, is defined by assigning to each layer the weights:

$$\begin{aligned}W^{(1)} &= I_s \otimes W_{(i\to h)}, \\ W^{(l)} &= \begin{bmatrix} W_{(h\to h)} & O_{|H|,\,(s-l)\cdot|H|} \\ I_{(s-l)} \otimes I_{|H|} & \end{bmatrix} \\ &\qquad \text{for } l \in \{2,..,n\}, \\ W^{(n+1)} &= W_{(h\to o)}.\end{aligned}$$

Notice that the input layer of the network applies the transformation defined by $W_{(i\to h)}$ to the input data. The hidden layers each take in another element of the transformed input data and combine it with the existing state using the weights in $W_{(h\to h)}$. Finally, the output layer applies the transformation given by $W_{(h\to o)}$.

We now show that given an RNN R, the FFNN R_n constructed by following Definition 12 is an appropriate abstraction. Specifically, the outputs produced by R_n are the same as those produced by R on any input sequence of length n.

Theorem 1. Let $f : (\mathbb{R}^x)^n \to \mathbb{R}^y$ denote the function computed by R and $g : \mathbb{R}^{xn} \to \mathbb{R}^y$ denote the function computed by R_n. Then, for all $\bar{x} \in (\mathbb{R}^x)^n$, it is the case that $f(\bar{x}) = g((\bar{x}_{0,0}, \ldots, \bar{x}_{x,n}))$.

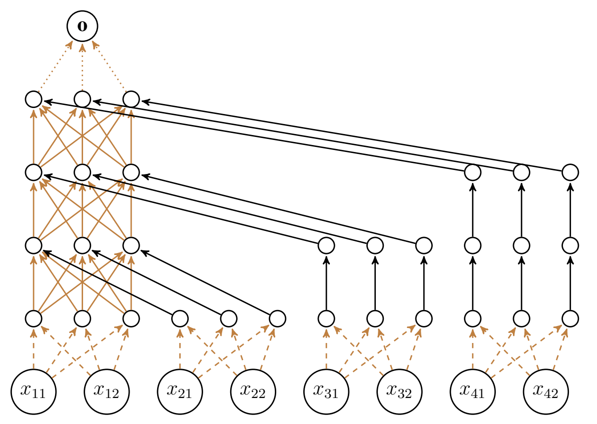
Proof sketch. The proof relies on showing that the output of layer k of the FFNN for $k < n$ is equal to the value of $h_k^{\bar{x}}$ in Definition 4. Then, the output of layer $n+1$ corresponds to the output of the RNN. □

Input on Demand (IOD). This unrolling method differs from the IOS method by passing on the input unchanged to the hidden units, rather than transforming the input at the start. This is illustrated in Figure 1b.

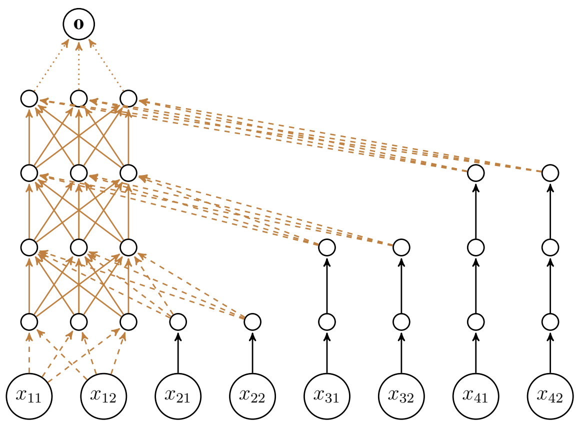
Definition 13. Let R be an RNN with weight matrices $W_{(i\to h)}$, $W_{(h\to h)}$ and $W_{(h\to o)}$. Let $n \in \mathbb{Z}^+$ be the length of sequences that we wish to consider. Then, the FFNN R'_n, called the length n unrolling of R, is defined by assigning to each layer the weights:

$$\begin{aligned}W^{(1)} &= \begin{bmatrix} W_{(i\to h)} & O_{|I|,\,|I|(s-1)} \\ O_{(|I|(s-1)),\,|H|} & I_{|I|(s-1)} \end{bmatrix}, \\ W^{(l)} &= \begin{bmatrix} W_{(h\to h)} & O_{|H|,\,|I|(s-l-1)} \\ W_{(i\to h)} & O_{|I|,\,|I|(s-l-1)} \\ O_{(|I|(s-l-1)),\,|H|} & I_{|I|(s-l-1)} \end{bmatrix} \\ &\qquad \text{for } l \in \{2,..,n\}, \\ W^{(n+1)} &= W_{(h\to o)}.\end{aligned}$$

Notice that in this case the input layer applies the transformation given by $W_{(i\to h)}$ to the first element of the sequence, and passes the rest of the sequence along unchanged. Each hidden layer then applies $W_{(i\to h)}$ to another element of the sequence and combines this with the existing state using $W_{(h\to h)}$. Finally, the output layer applies the transformation given by $W_{(h\to o)}$.



(a) Input on Start



(b) Input on Demand

Figure 1: FFNN constructed from an example RNN with an input sequence of length 4, input size of 2, 3 hidden units and output size 1. Note that the layers are fully connected but 0-weight connections are omitted from the image for clarity. Brown connections represent weights from the RNN, with dashed lines from $W_{(i\to h)}$, solid lines from $W_{(h\to h)}$, and dotted lines from $W_{(h\to o)}$. Black lines represent a weight of 1.

As for IOS, we now show that the IOD construction also produces an appropriate abstraction which is behaviourally equivalent to the original RNN for input sequences of length n.

Theorem 2. Let $f : (\mathbb{R}^x)^n \to \mathbb{R}^y$ denote the function computed by R and $g : \mathbb{R}^{xn} \to \mathbb{R}^y$ denote the function computed by R'_n. Then, for all $\bar{x} \in (\mathbb{R}^x)^n$, it is the case that $f(\bar{x}) = g((\bar{x}_{0,0}, \dots, \bar{x}_{x,n}))$.

Proof. The proof is similar to that of Theorem 1. $\square$

5 Verification of RNN-AESs

Having defined a sound unrolling of RNNs, we now show how to use this to verify RNN-AESs against the specifications previously introduced. Since both unrolling methods are correct, in the following we assume to fix one of the two. Their relative performance will be evaluated later.

The verification procedure defined in Algorithm 1 reduces the model checking problem for RNN-AESs to an instance of a mixed integer linear programming (MILP) problem. A MILP problem concerns finding a solution to a set of linear constraints on integer, binary or real-valued variables. We do not formalise the problem or methods that exist to solve it here as this is tangential to our work. For further information on MILPs, we refer the reader to (Winston 1987). It has been shown previously (Katz et al. 2017; Ehlers 2017; Lomuscio and Maganti 2017) that an l-layer FFNN N can be encoded as a set of linear constraints C_N on continuous variables $\bar{x} \in \mathbb{R}^m$ and $\bar{y} \in \mathbb{R}^n$ and binary variables $\Delta = \cup_{i=2}^{l} \bar{\delta}^{(i)}$ with $\bar{\delta}^{(i)} \in \mathbb{B}^{L^{(i)}}$, such that C_N is satisfiable when substituting in $\bar{x}$ and $\bar{y}$ iff $f(\bar{x}) = \bar{y}$. We assume the existence of Δ in a MILP encoding of an FFNN N, and omit it for brievity in the rest of our discussion.

Our procedure is shown in Algorithm 1. It takes as input an RNN-AES and a specification and returns whether the specification is satisfied on the system.

We use $X[\bar{x}, \bar{y}]$ to denote a set of constraints X with the variable $\bar{x}$ renamed to $\bar{y}$. We use SAT to denote a function checking whether a set of linear constraints passed as its argument is satisfiable.

We assume that the transition function is given by a set of linear constraints C_t on the variables $\bar{x}$ (the current state), $\bar{a}$ (the action performed by the agent) and $\bar{y}$ (the next state). We also assume the observation function is given by the set of linear constraints C_o on $\bar{x}$ and $\bar{o}$. Finally, we assume that the function computed by each R_n is given by a set of linear constraints C_{R_n} on $\bar{o}$ and $\bar{a}_n$.

For a formula $X^k\mathcal{C}$, for each step n from 0 to k the algorithm adds constraints corresponding to the observation function, the unrolling of length n of the RNN and the transition function of the environment. The constraint problem will then be satisfied precisely by states that are possible after k time steps. Then, the procedure checks whether it is possible for $\bar{\mathcal{C}}$ to be satisfied in any of these states, and returns a result accordingly.

The algorithm for $\mathcal{C}_1 U^{\leq k} \mathcal{C}_2$ is similar, except that at each time step n we check whether it is the case that $\mathcal{C}_2$ is always satisfied. If this is the case, we can return `True`. If this is not the case then we continue from the states that did not satisfy $\mathcal{C}_2$. We first check that these all satisfy $\mathcal{C}_1$. If this is not the case, we can return `False`. If it is the case, then we continue to the next time step. If we have reached k time steps and still have not returned a result, then there are paths of length k for which no state satisfies $\mathcal{C}_2$. So, we return `False`.

We now proceed to prove the correctness of this algorithm.

Theorem 3. For every RNN-AES AES and formula ϕ, Algorithm 1 returns `True` iff $AES \models \phi$.

Proof sketch. We first prove the case for $\phi = X^k\mathcal{C}$. Suppose that the algorithm returns `False`. Then it follows, that

Algorithm 1 Verification Procedure

```
    Input: RNN-AES $AES$; formula $\phi$
    Output: True/False
 1: $C \leftarrow C_I$
 2: switch $\phi$ do
 3:     case $X^k\mathcal{C}$
 4:         for $n \leftarrow 0$ until $k$ do
 5:             $C \leftarrow C[\bar{x}, \bar{x}_n] \cup C_o[\bar{x}, \bar{x}_n][\bar{o}, \bar{o}_n]$
 6:             $C \leftarrow C \cup C_{R_n} \cup C_t[\bar{x}, \bar{x}_n][\bar{a}, \bar{a}_n][\bar{y}, \bar{x}]$
 7:         end for
 8:         return $\neg$ SAT($\{\bar{\mathcal{C}}\} \cup C$)
 9:     case $\mathcal{C}_1 U^{\leq k} \mathcal{C}_2$
10:         for $n \leftarrow 0$ until $k$ do
11:             $C \leftarrow C \cup \{\bar{\mathcal{C}_2}\}$
12:             if $\neg$ SAT($C$) then
13:                 return True
14:             end if
15:             if SAT($\{\bar{\mathcal{C}_1}\} \cup C$) then
16:                 return False
17:             end if
18:             $C \leftarrow C[\bar{x}, \bar{x}_n] \cup C_o[\bar{x}, \bar{x}_n][\bar{o}, \bar{o}_n]$
19:             $C \leftarrow C \cup C_{R_n} \cup C_t[\bar{x}, \bar{x}_n][\bar{a}, \bar{a}_n][\bar{y}, \bar{x}]$
20:         end for
21:         return False
```

on Line 8 we found a satisfying assignment for the constraint problem $\{\bar{\mathcal{C}}\} \cup C$. Notice that since this satisfies the constraints added to C in Line 5 and 6, it is a length k path of the system (this follows from the unrolling being valid, as proved in Theorems 1 and 2). Notice also that since the final state satisfies $\bar{\mathcal{C}}$, it gives a counter-example for $AES \models X^k\mathcal{C}$. Hence, we have $AES \not\models \phi$, as desired. Conversely, note that if we returned `True` then no such path can exist (since otherwise it would give a satisfying assignment for the constraints), so $AES \models \phi$.

Now, let $\phi = \mathcal{C}_1 U^{\leq k} \mathcal{C}_2$. Suppose we return `False` on Line 16 on the nth iteration of the loop. Then, we found that $\{\bar{\mathcal{C}_1}\} \cup C$ has a satisfying assignment. It can be checked that the constraints added to C in Line 18 and 19 force this to be a valid path of length n, and the constraints added on Line 11 force that $\mathcal{C}_2$ did not hold in the first n states of the loop. Finally, the constraint $\bar{\mathcal{C}_1}$ means that $\mathcal{C}_1$ does not hold in the nth state. So, this path proves that $AES \not\models \phi$. If we returned `False` on Line 21, then on the kth iteration of the loop, we must have found a satisfying assignment for C on Line 12. Notice this gives a path of length k along which $\mathcal{C}_2$ is never satisfied, proving that $AES \not\models \phi$. Finally, suppose we returned `True` on Line 13. Then, we could not find an assignment for C on the nth iteration of the loop for some $n \leq k$. Notice this means that along every path of length n, we reach a state satisfying $\mathcal{C}_2$, and do not reach a state satisfying $\bar{\mathcal{C}_1}$ before this (otherwise we would have returned on Line 16). This shows that $AES \models \phi$. □

Having proved the validity of our method, we now proceed to present an implementation of it.

6 Implementation and Evaluation

We implemented the methods described in the previous sections into an experimental toolkit, called RNSVERIFY (RNSVerify 2018), thereby automating the unrolling procedures described in Section 4 as well as the verification procedure described in Algorithm 1. RNSVERIFY analyses an RNN-AES AES, which is composed of a recurrent neural agent, a linearly approximated environment, and a linearly defined set of initial states. RNSVERIFY supports specifications from Definition 9. The number of steps to be considered and the choice of the unrolling method are passed to the tool as additional parameters.

RNSVERIFY is written in Python and uses Gurobi ver. 7.5.2 (Gu, Rothberg, and Bixby 2016) as its underlying MILP solver. Our use of Gurobi is motivated by its attractive performance in various benchmarks (Mittelmann 2018).

Upon invocation the agent's RNN is unrolled into an FFNN (as described in Section 4) and the resulting linear constraints for the network are added to a MILP as described in the encoding given in (Akintunde et al. 2018), along with the encoding for the environment and the state transitions, as described in Section 5.

The tool outputs `True` if the property holds in the AES, or `False` if it does not. If the output is `False`, the tool also returns a counter-example for the formula in the form of a trace of states and actions for the agent.

In order to evaluate our implementation and its scalability, we consider the Open AI task Pendulum-v0 (OpenAI 2018). The objective of the task is for an agent to learn how to keep a pendulum upright by applying small rotational forces at each time step. For the agent, we used Q-Learning (Watkins and Dayan 1992) to train a ReLU-RNN with 16 hidden units using the open source deep learning toolkit Keras ver. 2.2.2 (Chollet 2015). The weights of the RNN were initialised as described in (Le, Jaitly, and Hinton 2015). The environment was linearly approximated using a ReLU-FFNN as described in (Akintunde et al. 2018).

In order to utilise a Q-Learning approach, we discretised the action space of the agent such that it may only produce forces contained in the set $\{-1, 1\}$, i.e., only a positive or negative force of equal magnitude. At each time step k, the network takes as input a sequence of states in the form $[(\theta_0, \dot{\theta}_0), \ldots, (\theta_{k-1}, \dot{\theta}_{k-1})]$, where at each time step θ represents the current angle of the pendulum and $\dot{\theta}$ represents its angular velocity, and outputs two Q-values $[q_1, q_2]$. To convert back into action space, we take the index $i_q = \arg\max([q_1, q_2])$ and apply a force of -1 if $i_q = 0$ or a force of 1 if $i_q = 1$ to compute the environment state for the next time step.

Let $(\theta_i, \dot{\theta}_i)$ denote the initial state of the environment and let $(\theta_f, \dot{\theta}_f)$ denote the final state of the environment after n time steps. We fix a set of initial states with $0 \leq \theta_i \leq \pi/64$ and $0 \leq \dot{\theta}_i \leq 0.3$, i.e., the pendulum begins possibly off-centre to the right and with an angular velocity taking it possibly further to the right.

We checked the property $\phi = X^n(\theta_f > -\varepsilon)$ for different positive values of ε and a variable number of steps n. Our results are recorded in Table 1, along with the time needed

		ε			
		$\pi/10$	$\pi/30$	$\pi/50$	$\pi/70$
n	1	0.056s	0.067s	0.011s	0.014s
	2	0.052s	0.179s	0.138s	0.197s
	3	0.372s	0.904s	5.794s	0.552s
	4	2.578s	7.222s	0.378s	0.368s
	5	20.57s	31.07s	0.748s	0.663s
	6	73.97s	3.264s	31.07s	23.99s
	7	54.30s	96.54s	116.8s	207.8s
	8	693.2s	294.9s	239.8s	243.3s

(a) Input on Start

		ε			
		$\pi/10$	$\pi/30$	$\pi/50$	$\pi/70$
n	1	0.004s	0.012s	0.011s	0.014s
	2	0.060s	0.114s	0.244s	0.253s
	3	0.247s	1.068s	6.092s	0.125s
	4	2.176s	5.359s	0.182s	0.198s
	5	10.04s	0.293s	0.317s	0.294s
	6	13.99s	0.367s	0.357s	0.359s
	7	31.93s	0.497s	0.488s	0.478s
	8	0.689s	0.660s	0.696s	0.703s

(b) Input on Demand

Table 1: The results of checking the property $X^n(\theta_f > -\varepsilon)$ after n steps using the IOS and IOD unrolling methods for different values of ε and n. Greyed out cells indicate a `False` result and white ones a `True` result. The time in the cell indicates the time Gurobi took to solve the corresponding MILP problem constructed by RNSVERIFY.

to obtain these results on a machine running Linux kernel 3.16 on an Intel Core i5-3320M CPU with 4GB of RAM.

Notice this property is true precisely if after n time steps it is always the case that the pendulum is not more than ε to the left of the vertical (i.e,. $\theta_f = 0$). When the property is false, the tool provides a trace showing the agent failing to satisfy the property, i.e., causing the pendulum to fall more than ε to the left of the vertical. For instance, when $n = 3$ and $\varepsilon = \pi/70$, the tool returns a trace starting from the initial state $(0, 0)$ where the agent repeatedly applies a force of -1 to bring the pendulum to the final state $(-0.046, -0.472)$. We see from this that the agent does not attempt to apply a force to move the pendulum in an opposite direction in order to keep the pendulum upright, but rather allows it to continue to fall. This highlights a case where the agent does not behave as desired. Following this the agent could be retrained and further verification conducted.

All our test results confirmed the correctness of the implementation. Further, note from the timing results that the IOD unrolling method performs better than IOS in all cases, with the difference being particularly noticeable for large values of n. This is likely due to the fact that, as shown in Table 2, the constraint problems constructed by RNSVERIFY have more variables and constraints in them and, accordingly, take Gurobi longer to solve. The performance of our experimental tool is promising and scales well with the number of steps.

n	Input on Start V	Input on Start C	Input on Demand V	Input on Demand C
1	273	336	273	336
2	736	736	620	766
3	1455	1806	1055	1306
4	2494	3101	1590	1971
5	3917	4876	2237	2776
6	5788	7211	3008	3736
7	8171	10186	3915	4866
8	11130	13881	4970	6181

Table 2: For different values of n, the size of the constraint problem constructed by RNSVERIFY in terms of number of variables (V) and constraints (C) when checking the property $X^n(\theta_f > -\varepsilon)$ using the IOS or IOD unrolling method.

7 Conclusions

As we argued in the Introduction, recent developments of autonomous and robotic systems, in which neural networks are used in parts of the architecture, make the formal verification of the resulting systems very challenging. In this paper we have developed a method for the formal verification of systems composed by a stateful agent implemented by an RNN interacting with an environment. We have introduced a semantics supporting this closed-loop system, defined and solved the resulting verification problem. The method relies on unravelling RNNs into FFNNs and compiling the resulting verification problem into a MILP. We showed the method is sound, complete and effective for controllers of limited complexity.

The method is novel and addresses a presently unsolved problem as no method for verifying RNNs or agents based on them currently exists. In the future we intend to address some limitations of our approach to make it suitable for verifying real-life autonomous systems. Firstly, we intend to improve on the present solutions for the verification step. This is presently implemented in MILP, but there is no reason it should not be carried out in SMT (Barrett et al. 2011), or by combining approaches to improve its performance. Secondly, we intend to develop the toolkit RNSVERIFY further by adding a modelling language for the agents and the environment.

8 Acknowledgements

Alessio Lomuscio is supported by a Royal Academy of Engineering Chair in Emerging Technologies. This work is partly funded by DARPA under the Assured Autonomy programme. Nuri Cingillioglu's help on training recurrent neural networks is gratefully acknowledged. The authors are grateful to Giuseppe De Giacomo for extensive comments on a previous version of this paper.

References

Akintunde, M.; Lomuscio, A.; Maganti, L.; and Pirovano, E. 2018. Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th International Confer-*

ence on Principles of Knowledge Representation and Reasoning (KR18), 184–193. AAAI Press.

Alechina, N.; Dastani, M.; Khan, F.; Logan, B.; and Meyer, J. J. C. 2010. Using theorem proving to verify properties of agent programs. In *Specification and Verification of Multi-agent Systems*. Springer. 1–33.

Barrett, C.; Conway, C.; Deters, M.; Hadarean, L.; Jovanović, D.; King, T.; Reynolds, A.; and Tinelli, C. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV11)*, volume 6806 of *Lecture Notes in Computer Science*, 171–177. Springer.

Brewer, J. 1978. Kronecker products and matrix calculus in system theory. *IEEE Transactions on Circuits and Systems* 25(9):772–781.

Bunel, R.; Turkaslan, I.; Torr, P. H. S.; Kohli, P.; and Kumar, M. P. 2017. Piecewise linear neural network verification: A comparative study. *CoRR* abs/1711.00455v1.

Chollet, F. 2015. Keras. https://keras.io.

D'Ambrosio, C.; Lodi, A.; and Martello, S. 2010. Piecewise linear approximation of functions of two variables in milp models. *Operations Research Letters* 38(1):39–46.

Ehlers, R. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA17)*, volume 10482 of *Lecture Notes in Computer Science*, 269–286. Springer.

Ezekiel, J.; Lomuscio, A.; Molnar, L.; and Veres, S. 2011. Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI11)*, 1659–1664. AAAI Press.

Gehr, T.; Mirman, M.; Drachsler-Cohen, D.; Tsankov, P.; Chaudhuri, S.; and Vechev, M. 2018. AI^2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (S&P18)*, 948–963.

Gu, Z.; Rothberg, E.; and Bixby, R. 2016. Gurobi optimizer reference manual. http://www.gurobi.com.

Hausknecht, M., and Stone, P. 2015. Deep recurrent q-learning for partially observable mdps. In *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (AAAI-SDMIA15)*. AAAI Press.

Haykin, S. S. 1999. *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Huang, X.; Kwiatkowska, M.; Wang, S.; and Wu, M. 2017. Safety verification of deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV17)*, volume 10426 of *Lecture Notes in Computer Science*, 3–29. Springer.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV17)*, volume 10426 of *Lecture Notes in Computer Science*, 97–117. Springer.

Kouvaros, P.; Lomuscio, A.; and Pirovano, E. 2018. Symbolic synthesis of fault-tolerance ratios in parameterised multi-agent systems. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence and 23rd European Conference on Artificial Intelligence (IJCAI-ECAI18)*, 324–330. IJCAI.

Le, Q. V.; Jaitly, N.; and Hinton, G. E. 2015. A simple way to initialize recurrent networks of rectified linear units. *CoRR* abs/1504.00941.

Lomuscio, A., and Maganti, L. 2017. An approach to reachability analysis for feed-forward relu neural networks. *CoRR* abs/1706.07351.

Lomuscio, A.; Qu, H.; and Raimondi, F. 2017. MCMAS: A model checker for the verification of multi-agent systems. *Software Tools for Technology Transfer* 19(1):9–30.

Mittelmann, H. 2018. Benchmarks for Optimization Software. http://plato.asu.edu/bench.html.

Nair, V., and Hinton, G. E. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML10)*, 807–814. Omnipress.

OpenAI. 2018. Pendulum-v0. https://gym.openai.com/envs/Pendulum-v0/.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th International Symposium Foundations of Computer Science (FOCS77)*, 46–57.

Pulina, L., and Tacchella, A. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV10)*, volume 6184 of *Lecture Notes in Computer Science*, 243–257. Springer.

Redmon, J.; Divvala, S. K.; Girshick, R. B.; and Farhadi, A. 2016. You only look once: Unified, real-time object detection. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR16)* 779–788.

RNSVerify. 2018. Recurrent Neural System Verify, http://vas.doc.ic.ac.uk/software/ – also available as supplementary material.

Ruan, W.; Huang, X.; and Kwiatkowska, M. 2018. Reachability analysis of deep neural networks with provable guarantees. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence and 23rd European Conference on Artificial Intelligence (IJCAI-ECAI18)*, 2651–2659. AAAI Press.

Sutskever, I.; Martens, J.; and Hinton, G. E. 2011. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML11)*, 1017–1024. Omnipress.

Watkins, C. J. C. H., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3–4):279–292.

Winston, W. 1987. *Operations research: applications and algorithms*. Duxbury Press.