Proceedings of the Sixteenth International Conference on
Principles of Knowledge Representation and Reasoning (KR 2018)

Reachability Analysis for Neural Agent-Environment Systems

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

Abstract

We develop a novel model for studying agent-environment systems, where the agents are implemented via feed-forward ReLU neural networks. We provide a semantics and develop a method to verify automatically that no unwanted states are reached by the system during its evolution. We study several reachability problems for the system, ranging from one-step reachability, to fixed multi-step and arbitrary-step to study the system evolution. We also study the decision problem of whether an agent, realised via feed-forward ReLU networks will perform an action in a system run. Whenever possible, we give tight complexity bounds to decision problems introduced. We automate the various reachability problems studied by recasting them as mixed-integer linear programming problems. We present an implementation and discuss the experimental results obtained on a range of test cases.

1 Introduction

Over the past ten years, there has been growing interest in trying to verify formally the correctness of AI systems. This has been compounded by recent public calls for the development of "responsible" and "verifiable" AI (Russell, Dewey, and Tegmark 2015). Indeed, since the development of ever more complex and pervasive AI systems including autonomous vehicles, the need for higher guarantees of correctness for the systems has intensified.

Differently from many areas within AI, in multi-agent systems (MAS) there already has been considerable activity aimed at verifying MAS formally. In one line, efficient model checkers for finite state MAS against expressive AI-based specifications, such as those based on epistemic logic, have been developed (Lomuscio, Qu, and Raimondi 2017; Gammie and van der Meyden 2004). Abstraction techniques have also been put forward to verify infinite state MAS (Lomuscio and Michaliszyn 2016) and approaches for parameterised verification for MAS and swarms have been introduced (Kouvaros and Lomuscio 2015). In a different strand of work, theorem proving approaches have also been tailored to MAS (Alechina et al. 2010; Shapiro, Lespérance, and Levesque 2002).

While significant results have been achieved in these lines, their object of study is a system that is given either via a traditional programming language or a MAS-oriented programming language. *None of the present approaches can provide any guarantees on systems synthesised by machine learning methods*. However, systems based on machine learning are increasingly deployed in a variety of applications. These systems are not programmed directly; instead, neural networks are first appropriately trained against data and then employed to conduct a particular task. Present applications of deep neural networks include state-of-the-art systems for automatic vision (Krizhevsky, Sutskever, and Hinton 2012), natural language processing (Sutskever, Vinyals, and Le 2014), and recommender systems (van den Oord, Dieleman, and Schrauwen 2013). If the present pace of development in machine-learning continues, it is expected that machine learning technology will provide key parts in a wide range of AI applications, including robotics, autonomous systems, and AI decision making systems.

These will be closed-loop systems where the actions of a neural agent, e.g., a controller implementing a neural network, affect the environment, which is in turn observed by the agent. We are not aware of any method that can give guarantees on the behaviour of this important class of forthcoming AI systems. This paper aims to make a first contribution on this topic by devising a method to study reachability for systems composed of a neural agent interacting in a closed loop with an environment.

We begin this investigation by addressing reachability as this is a key property to analyse on any system. In a nutshell, the reachability decision problem consists in determining whether a given state is reachable in one or more steps from some initial state of the system. By performing reachability analysis, we can establish whether an unwanted state of the system, e.g., an erroneous state or a bug, is ever reached during the system evolution. Analysing reachability is often of lower complexity than exploring temporal properties, such as those expressible in Linear Temporal Logic (Pnueli 1977). Given the high computational cost of verifying specifications in temporal logic, reachability analysis is often the only realistic property that can be verified on real systems; for example, in multi-threaded systems a key part of the analysis concerns race conditions, which are expressible as reachability properties (Bouajjani et al. 2005).

In this paper, while we give a general agent-environment model, we focus on systems comprising of agents imple-

mented via deep feed-forward neural networks, where the activation function is governed by ReLU functions (Haykin 2011; Nair and Hinton 2010). The rest of the paper is organised as follows. In Section 2, we fix the notation of ReLU-based neural networks and mixed integer linear programs. In Section 3, we present a model of a neural system as a combination of a neural agent and an environment and define its evolution from a set of initial states. In Section 4 we introduce a number of reachability decision problems for these systems and give methods, based on mixed-integer linear programming, to solve them. NSVERIFY, a toolkit implementing these methods is presented in Section 5, where experimental results are also given. While we are not aware of other methods addressing closed-loop systems, very recently some methods have been put forward to study the possible output of ReLU-based networks. We present further experimental results in Section 6, by comparing the performance of NSVERIFY against existing approaches in the literature for the limited case of studying the network output. We exemplify all concepts and the implementation on the OpenAI Gym (Brockman et al. 2016) task PENDULUM-V0.

Related Work. The literature on verification of MAS addresses systems where the model is given either declaratively or procedurally. While we are inspired by this work, instead we here study a class of agent-environment systems where agents are based on neural computation.

Much of the literature on feed-forward neural networks is concerned with training and does not address the formal verification question. Currently, techniques used for evaluating the correctness of networks rely on test datasets which can provide statistical guarantees at most and are thus incomplete. The few exceptions that we are aware of are discussed below. We note that all of these only address the issue of whether a network can output a value, i.e. in an open-loop system. The method we put forward also allows this but is more general since it also addresses closed-loop systems.

(Kurd and Kelly 2003) advocates the use of safety specifications to validate neural networks. The work here presented on reachability partially falls within the types 3 and 4 of safety which they discuss. While the broad direction of their work is in line with what is pursued here, no actual verification method is discussed.

The research line on adversarial examples, e.g. (Bastani et al. 2016), can be seen as special case of reachability where the input set is constrained with respect to a specific input; thus the formulation here proposed is more general even for the case of open-loop system. Related to this, a method for finding adversarial inputs using a layer-by-layer approach and employing *satisfiability modulo theories* (SMT) solvers was recently proposed (Huang et al. 2017). This technique supports any activation function, not just ReLU as we do here. However, because the focus is on adversarial inputs, as before, the method seems to not immediately generalise to solving reachability on feed-forward networks.

A methodology for the analysis of ReLU feed-forward networks, conducted independently from this research and originally around the same time has recently appeared (Katz et al. 2017). While their method is based on SMT-solving, we only use linear programming here. Moreover, while we here address the composition of a neural agent with an environment, (Katz et al. 2017) is concerned with stand-alone neural networks only as discussed above. A comparison limited to neural networks only is presented in Section 6.

A further approach to combining SMT and linear programming (LP) techniques to assess feed-forward networks has also recently appeared (Ehlers 2017). As above, no formal correspondence is made between reachability and their encoding, and closed-loop systems are not analysed. The work presented in (Cheng, Nührenberg, and Ruess 2017) uses a mixed-integer linear programming approach to solve the verification problem for a single network, but like other tools, does not consider reachability properties of closed-loop systems. Finally, in (Bunel et al. 2017), a comparative study of the methods currently used for the verification of neural networks with piecewise-linear activation functions. Along with this, the authors propose a method based on the Branch-and-Bound (Lawler and Wood 1966) framework. As with other approaches above, closed-loop systems of agents and environments are not considered. All these methods are compared to the one we here present, to the extent that this comparison can be made, in Section 6.

2 Preliminaries

In this section we present concepts essential to the rest of the paper. In particular, we introduce the class of neural networks that we will study and the linear programming techniques that we will use. For more details on neural networks and linear programming we refer to (Haykin 1999; Dantzig 1998).

Feed-forward neural networks. Recall that feed-forward neural networks (FFNN) are one of the simplest classes of neural networks consisting of multiple *hidden layers* and admitting no cycles (Zell et al. 1994).

We now fix some notation that will be used throughout the paper A *feed-forward (multilayer) neural network* N is made up of a number of layers; we denote the ith *layer* of N as $L^{(i)}$; each layer $L^{(i)}$ has a *weight matrix* $W^{(i)}$, a *bias vector* $b^{(i)}$, and an *activation function* as $\sigma^{(i)}$. Each *layer* of the network consists of multiple *nodes*, which are single computation units that combine input from the values of nodes in the previous layer, and produce an output. This output is to be used in the computations of nodes in successive layers.

For a network with n layers, 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*. Note that we only consider *fully-connected* neural networks here, meaning that all 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 will define the *value* of the node, which can

then be passed into a subsequent node.

We consider only networks with hidden layers utilising *Rectified Linear Unit (ReLU)* activation functions, defined $\text{ReLU}(x) = \max(0, x)$, where x represents the linear combination of values from incoming nodes in the previous layer. The value of the node will be 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 1 (Function computed by FFNN)**.** Let N be an 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 $f(x) = f^{(m)}(f^{(m-1)}(\ldots(f^{(2)}(x))))$.

Linear Programming. Linear programming (LP) is an optimisation technique where we seek to maximise a linear objective function subject to a set of linear constraints on the input values. Efficient algorithms exist to solve linear programming problems (Winston 1987). For the purposes of this paper, we consider mixed integer linear programs, which contain both real and integer variables.

Definition 2 (Mixed Integer Linear Programs)**.** A function $f(x_1, \ldots, x_n)$ is said to be *linear* if for some $c \in \mathbb{R}^N$, we have $f(x_1, \ldots, x_n) = \sum_{i=1}^{N} c_i x_i$. For any linear function $f(x_1, \ldots, x_n)$, and any $b \in \mathbb{R}$, the expressions $f(x_1, \ldots, x_n) = b$, $f(x_1, \ldots, x_n) \leq b$ and $f(x_1, \ldots, x_n) \geq b$ are said to be *linear constraints*. A *mixed integer linear program* (MILP) is an optimisation problem where the objective function is linear and the constraints on the variables of the objective function are linear. MILP problems allow for real, binary and integer decision variables.

Definition 3 (Linearly Definable Set)**.** Let $S \subseteq \mathbb{R}^n$. We say that S is *linearly definable* if there exists a finite set of linear constraints C_S such that $S = \{x \in \mathbb{R}^n \mid x \text{ satisfies every constraint in } C_S\}$. We define one such C_S to be the *constraint set* of S.

Notation. For a linearly definable set S with constraint set C_S, we take v_S to be the LP variables used in defining C_S. We use the term *linearly definable function* to refer to any function which can be encoded as a set of mixed integer linear constraints. This includes piecewise linear functions, logical Boolean functions ($\wedge$, $\vee$, $\neg$), max, min, the absolute value function $|\cdot|$, as well as indicator and conditional constructs. In the following we will refer to feed-forward neural networks which exclusively use ReLU activation functions throughout all hidden layers as *ReLU-FFNNs*.

3 Neural Agent-Environment Systems

In this section we introduce the notion of a *neural agent-environment system*. We will then present a restriction of these general systems called *linearly-definable neural agent-environment systems* and show how to obtain such a linearly-definable system from a general one.

At the core of our model is the concept of an autonomous agent situated in an environment performing actions onto this environment and observing and reacting to their effects (Wooldridge 2009). We here limit ourselves to memoryless agents that react purely to the current state of the environment. In traditional agent-based systems, the agent's decision-making mechanism, which determines which action to perform given the observations and the present state, is a program, often in an agent-based programming language. Instead, we here consider decision-making mechanisms synthesised from data and implemented via ReLU-FFNNs. For ease of presentation, we assume that the agent has full observability of the environment, i.e. that the input to the agent's ReLU-FFNN is the full state of the environment. This restriction can easily be removed by introducing an additional function from the environment to the agent's input, but this is not pursued here for ease of presentation.

General Neural Agent-Environment Systems

A neural agent-environment system consists of a closed-loop system comprising an agent and an environment. The environment is stateful and updates its state in response to the action of the agent. The agent is stateless and chooses an action on the basis of the state of the environment, which is fully observable. We begin by defining the *environment*.

Definition 4 (Environment)**.** We define an *environment* as a tuple $E = (S, t_E)$, where:

- S is a set of states of the environment,
- $t_E : S \times Act \to S$ 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.

Throughout the paper we will consider and build upon the example below.

Example 1. Consider the OpenAI Gym (Brockman et al. 2016) task PENDULUM-V0. The system is composed of a pendulum and an agent which can apply a force to the pendulum. The agent can observe the current angle of the pendulum (where an angle of zero indicates that it is perfectly vertical) along with the pendulum's angular velocity. Based on this, the agent chooses an action, which consists of a torque (rotational force) to be applied to the pendulum. The aim of the agent is to get the pendulum to an upright position and maintain it there. We would traditionally consider the pendulum as controlled by a program with pre-defined states and transitions. Instead we consider an agent running a ReLU-FFNN trained to perform this task (see Example 2).

We formalise the environment for this problem as $E = (S, t_E)$ where:

- S is the set of tuples $(\theta, \dot{\theta}) \in [-\pi, \pi] \times [-8, 8]$ where:
  - θ represents the pendulum's angle, and
  - $\dot{\theta}$ represents pendulum's angular velocity.

- $t_E : S \times Act \to S$ is the transition function mapping the state-action pair $((\theta, \dot{\theta}), a)$ to a new state $(\theta', \dot{\theta}')$ (see below for a description of the agent component), where:
  - $\theta' = \theta + \dot{\theta}' \cdot \mathrm{d}t$, and
  - $\dot{\theta}' = \dot{\theta} + (-(3g/2l)\sin(\theta + \pi) + 3(\max(\min(a, 2), -2))/(ml^2)) \cdot \mathrm{d}t$,

  where $g = 10, m = l = 1$ and $\mathrm{d}t = 1/20$. This function is obtained by restricting a to the range $[-2, 2]$ (which is the maximum rotational force that may be applied), and then using the equations of motion for the physical dynamics governing the pendulum.

We now proceed to define the notion of a neural *agent*. To achieve this, we will henceforth assume that the states and actions can be defined by real vectors.

Definition 5 (Neural Agent)**.** An *agent*, denoted Agt, acting on an environment E is defined by an *action* function $act : S \to Act$, which given an environment state from $S \subseteq \mathbb{R}^m$ returns an action from a set $Act = \mathbb{R}^n$ of admissible actions for the agent. We henceforth assume that the function act is computed by a ReLU-FFNN 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 refer to the agent as a *neural agent* Agt_N.

Example 2. Recall Example 1. Agt_N is the neural agent responsible for keeping the pendulum upright. The agent's action function act can be given by a neural network, which has been trained to apply a suitable force $a \in \mathbb{R}$ given some environment state $(\theta, \dot{\theta}) \in S$. We do not present such a neural network here, but refer to Section 5 for more details. Notice we assume that the neural network has been previously trained and all the weights and biases are fixed.

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

Definition 6 (Agent-environment system)**.** An *agent-environment system* (AES) is a tuple $AES = (E, Agt, I)$ where:

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

Observe that the environment is stateful and the agent is stateless. Given a state $s \in S$ and action determined by the output of act, the next state of the environment is $s' = t_E(s, act(s))$.

If the agent is a *neural agent* (see Definition 5), we call the resulting system a *Neural agent-environment System* (NS).

Example 3. Extending the previously defined pendulum example, consider a neural agent-environment system $NS = (E, Agt_N, I)$, with:

- E denoting the environment with corresponding transition function defined in Example 1,
- Agt_N denoting the neural agent defined in Example 2,
- $I = \{(0, 0.1)\}$ giving a single initial state with the pendulum at zero angle (vertical) and beginning to rotate clockwise with a small initial angular velocity of 0.1.

Given an agent-environment system, we wish to define its evolution. We formalise this notion below.

Definition 7 (System evolution)**.** Given a neural agent-environment system $NS = (E, Agt_N, I)$, we say that NS *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 $t_E^{(n+1)}(x) = t_E(t_E^{(n)}(x), f(t_E^{(n)}(x)))$ 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.

Example 4. Recall the pendulum problem from Example 3 with corresponding neural agent-environment system $NS = (E, Agt_N, I)$. Assume the system is in the initial state $x = (\theta, \dot{\theta}) = (0, 0.1)$ with t_E defined as in Example 1 and consider the trivial agent action function $act(s) = 0.5$ for all $s \in S$, which applies a constant force of 0.5. It can be computed that NS evolves to state $y = (\theta', \dot{\theta}') = (0.0264, 0.3391)$ in $n = 3$ steps since $t^{(3)}(x) = y$.

Linearly-definable Neural Agent-Environment Systems

Note that thus far our environment E is very general and can use any real-valued transition function. We will now define the notion of a *linearly-definable environment* and *linearly-definable neural agent-environment system* to allow us to solve reachability problems on agent-environment systems. We will later discuss how we can approximate a more general environment to such a linearly-definable one.

Recall the notion of an environment $E = (S, t_E)$ (Definition 4). If the transition function t_E is linearly-definable (Definition 3), we say that E is a *Linearly-definable Environment* (LE).

We now consider systems resulting from the composition of one agent with an environment.

Definition 8 (Linearly-definable Neural Agent-Environment System (LNS))**.** A *Linearly-definable Neural Agent-Environment System* $LNS = (LE, Agt_N, I)$ is a tuple where:

- $LE = (S, t_{LE})$ is a linearly-definable environment on states S with a linearly-definable transition function t_{LE}.
- Agt_N is a neural agent with its action function act characterised by a neural network N with m inputs ranging on the states S of the LE environment LE and n outputs.
- $I \subseteq S$ is a linearly-definable set of initial states.

As before (see Definition 6), the transition function is determined by $t_{LE}(s, (act(s)))$ for $s \in S$.

Example 5. Recall the neural agent-environment system $NS = (E, Agt_N, I)$ from Example 3. We can assume the existence of a piecewise-linear approximation t_{LE} for the non-linear transition function t_E (Eriksson, Estep, and Johnson 2004). The resulting environment $LE = (S, t_{LE})$ is

linearly-definable. So, the system $NS = (LE, Agt_N, I)$ is an LNS, since $I = \{(0, 0.1)\}$ is also linearly-definable.

As a further example, where the environment is inherently linear, consider a 1kg cart on a frictionless track of length 10. The state S of the system is given by the set of tuples $(p, u) \in [0, 10] \times \mathbb{R}$, which denotes the position of the cart and its current velocity. The agent Agt_N chooses a force a (in Newtons) in the range $[-1, 1]$ to apply to the cart for 1 second. We define the transition function as follows:

$$t_{LE}\left((p, u), a\right) = \begin{cases} (p + u + a/2, u + a) & \text{if } 0 \leq p' \leq 10 \\ (10, 0) & \text{if } p' > 10 \\ (0, 0) & \text{if } p' < 0, \end{cases}$$

where $p' = p + u + a/2$. Observe that t_{LE} is linearly-definable, and so we can construct the linearly-definable environment $LE = (S, t_{LE})$. Assume the initial state set $I = \{(0, 0)\}$. The resulting system $LNS = (LE, Agt_N, I)$ is a linearly-definable neural agent-environment system.

Neurally-Approximated Environments

The restriction imposed by a linearly-definable neural agent-environment system lies in the requirement that environments need to be linearly-definable. In realistic settings, however, agents interact with non-linear environments.

To accommodate this we approximate the environment by training a ReLU-FFNN to mimic the environment. This FFNN can be trained to approximate the environment that it models to arbitrary fidelity. We now formally define this *environment approximation*.

Definition 9 (Environment Approximators)**.** Let $S \subseteq \mathbb{R}^m$ and $A \subseteq \mathbb{R}^n$ be a set of states and actions respectively. Let $e : S \times A \to S$ be an arbitrary function. Let N be a RELU-FFNN with computed function $f : \mathbb{R}^{m+n} \to \mathbb{R}^m$. We define f as an *environment approximator* for e with error ϵ if for some error metric m, $m(f, e) \leq \epsilon$.

Several choices can be made in choosing the error metric m. For example, we can take the average difference between expected and actual value on a representative sample domain. Note that by the universal approximation theorem for FFNNs (Hornik, Stinchcombe, and White 1989), we can approximate any continuous function to arbitrary precision using a multilayer FFNN. Thus, the environment approximations can be as close as desired to the original non-linear environment.

Observation 1. Since we can linearly encode ReLU-FFNNs (we formalise this later in Definition 10), our neural approximation of the environment is linearly definable.

As a consequence of Observation 1, we will henceforth only consider linearly-definable environments where we assume that, if necessary, they have been obtained by training a ReLU-FFNN to suitably approximate a non-linear one.

4 Reachability Analysis via MILP Solving

In this section we introduce a variety of decision problems related to reachability in neural agent-environment systems. Reachability analysis is of fundamental importance in determining whether or not a system may encounter an error state. It can be used to determine whether a system satisfies safety properties by checking a fault is never encountered during any system run. We refer to (Hudak, Simonak, and Korecko 2010) for more details.

We solve these decision problems via compilation into mixed-integer linear programming (MILP) problem instances. For ease of presentation, throughout this section we fix a $LNS = (LE, Agt_N, I)$ and assume that the action function of Agt_N is given by an m-layer ReLU-FFNN denoted N. Before proceeding, we give a linear encoding of N.

Definition 10 (Linear Encoding for a ReLU-FFNN)**.** Let N be an m-layer FFNN with computed function f. Suppose $\bar{x}^{(i-1)}$ and $\bar{x}^{(i)}$ are vectors of real (LP) variables representing the input and output of layer i respectively and $\bar{\delta}^{(i)}$ is a vector of binary (LP) variables. We use the value of $\bar{\delta}^{(i)}$ to indicate whether the ReLU unit is in the "active" (equal to its input) or "inactive" (equal to zero) phase. Specifically, for some $k \in \mathbb{N}$, The unit is active if $\bar{\delta}^{(i)}_k = 0$, and is inactive if $\bar{\delta}^{(i)}_k = 1$. Then, the set of *linear constraints encoding layer* i is defined as:

$$\begin{aligned} C_i = \{&\bar{x}^{(i)}_j \geq W^{(i)}_j \bar{x}^{(i-1)} + b^{(i)}_j, \\ &\bar{x}^{(i)}_j \leq W^{(i)}_j \bar{x}^{(i-1)} + b^{(i)}_j + M\bar{\delta}^{(i)}_j, \\ &\bar{x}^{(i)}_j \geq 0, \bar{x}^{(i)}_j \leq M(1 - \bar{\delta}^{(i)}_j) \mid j = 1 \ldots |L^{(i)}|\}, \end{aligned}$$

where M is larger than the largest possible magnitude of $W^{(i)}_j \bar{x}^{(i-1)} + b^{(i)}_j$. For more details on the choice of a M in this context, we refer to (Cheng, Nührenberg, and Ruess 2017). For our purposes, we simply fix a large enough constant in line with the constraint above.

Let $C_N = \cup^m_{i=2} C_i$, which we will refer to as the set of *linear constraints encoding the network*.

We now proceed to show that the set of linear constraints above precisely captures the computation of the network.

Lemma 1. The constraint problem C_N is satisfiable when substituting $\bar{x}$ for $\bar{x}^{(1)}$ and $\bar{y}$ for $\bar{x}^{(m)}$ iff $f(\bar{x}) = \bar{y}$.

Proof. ($\Leftarrow$) Suppose $f(\bar{x}) = \bar{y}$. Then, set each $\bar{x}^{(i)}$ to $f^{(i)}(\bar{x})$, and set:

$$\bar{\delta}^{(i)}_j \triangleq \begin{cases} 0 \text{ if } \bar{x}^{(m)}_j > 0 \\ 1 \text{ otherwise} \end{cases}$$

Then, we observe that the constraints in C_N are satisfied by our definition of the neural network. The converse is similar. □

Notice that if we have a non-linear environment, we can also apply this linear encoding to a neural approximation of the environment in order to obtain a set of linear constraints capturing the evolution of the environment's state. We use C_{LE} to denote this set of linear constraints on input vector $\bar{x}$ and output vector $\bar{y}$.

Finally, we use C_I to denote the set of linear constraints encoding the initial states on a vector $\bar{v}_I$.

Single-Step State Reachability

To begin with, we define a simple notion of single-step reachability through which we will determine whether it is the case that a set of target states can be reached by the agent-environment system in a single step. We formalise this decision problem below.

Definition 11 (SSR Decision Problem)**.** Let $O \subseteq S$ be a linearly definable set of target states. The *single-step state reachability* (SSR) decision problem involves determining whether it is the case that $\exists \bar{i} \in I$ such that $t^{(1)}_{LE}(\bar{i}) \in O$.

We now proceed to give an encoding of this decision problem into the feasibility of a linear program.

Definition 12 (SSR Linear Encoding)**.** Let $n \in \mathbb{N}$ and $O \subseteq S$ be a linearly definable set of target stages defined by the set of constraints C_O on the vector $\bar{v}_O$. The linear program R_O is defined by the set of constraints:

$$C_I \cup \{\bar{v}_I = \bar{x}^{(1)}\} \cup C_N \cup \{\bar{x}^{(m)} = \bar{x}\} \cup C_{LE} \cup \{\bar{y} = \bar{v}_O\} \cup C_O.$$

Intuitively this linear encoding captures all the computation of the system by taking the union of the constraints defining the initial states, evaluation of the network, evolution of the environment and output states. It also includes additional constraints to match up the corresponding variables. We now proceed to prove that this linear encoding precisely characterises the first step of the system's evolution.

Theorem 2. The linear program R_O is feasible if and only if $\exists \bar{i} \in I$ such that $t^{(1)}_{LE}(\bar{i}) \in O$.

Proof sketch. ($\Leftarrow$) Suppose that $\exists \bar{i} \in I$ such that $t^{(1)}_{LE}(\bar{i}) \in O$. Then, consider the assignment that maps each $\bar{x}^{(k)}$ to $f^{(k)}(\bar{i})$ and $\bar{y}$ to $t^{(1)}_{LE}(\bar{i})$. This gives a satisfying assignment, showing that the linear program is feasible. The converse is similar. □

From this result we can derive a sound and complete decision procedure for SSR by constructing the corresponding linear encoding and checking its feasibility using existing procedures for solving linear programs.

Multi-Step State Reachability

We now extend the method above to solve the more general decision problem of establishing whether a set of states is reachable in n steps. We formalise this below.

Definition 13 (MSR Decision Problem)**.** Let $n \in \mathbb{N}$ and $O \subseteq S$ be a linearly definable set of target states. The *multi-step state reachability* (MSR) decision problem involves determining whether it is the case that $\exists \bar{i} \in I$ such that $t^{(n)}_{LE}(\bar{i}) \in O$.

To solve MSR we create n copies of the linear constraints encoding the computation of the network controlling the agent and the environment and compose these into one linear program, thereby encoding the computation of the first n steps. This encoding is formalised below.

Definition 14 (MSR Linear Encoding)**.** Let $n \in \mathbb{N}$ and $O \subseteq S$ be a linearly definable set of target stages defined by the set of constraints C_O on the vector $\bar{v}_O$. Now, denote by $C_{N,k}$ the relabelling of the constraints in C_N so that each $\bar{x}^{(i)}$ is renamed to $\bar{x}^{(i,k)}$. Similarly, denote by $C_{LE,k}$ the relabelling of the constraints in C_{LE} so that $\bar{x}$ is renamed to $\bar{x}^{(m,k)}$ and $\bar{y}$ is renamed to $\bar{x}^{(1,k+1)}$. The linear program R^O_n is defined by the set of constraints:

$$C_I \cup \{\bar{v}_I = \bar{x}^{(1,1)}\} \cup (\cup^n_{k=1}(C_{N,k} \cup C_{LE,k})) \cup \{\bar{x}^{(1,n+1)} = \bar{v}_O\} \cup C_O.$$

We now prove that the linear program R^O_n fully characterises the states reachable in precisely n steps.

Theorem 3. The linear program R^O_n is feasible if and only if $\exists \bar{i} \in I$ such that $t^{(n)}_{LE}(\bar{i}) \in O$.

Proof sketch. ($\Leftarrow$) Suppose that $\exists \bar{i} \in I$ such that $t^{(n)}_{LE}(\bar{i}) \in O$. Then, consider the assignment that maps each $\bar{x}^{(m,k)}$ to $f^{(m)}(t^{(k)}_{LE}(\bar{i}))$. This gives a satisfying assignment, showing the linear program is feasible. The converse is similar. □

As in the single-step, we can derive a sound and complete decision procedure for the MSR decision problem by encoding it into the linear program R^O_n, and then using existing techniques to verify the feasibility of this linear program.

Arbitrary-Step State Reachability

In view of generalising the results above, we now aim to characterise all states that are reachable in an arbitrary but finite number of time steps from a set of initial states.

Definition 15 (ASR Decision Problem)**.** Let $O \subseteq S$ be a linearly definable set of target states. The *arbitrary-step state reachability* (ASR) decision problem involves determining whether it is the case that $\exists n \in \mathbb{N}$ and $\exists \bar{i} \in I$ such that $t^{(n)}_{LE}(\bar{i}) \in O$.

As a stepping stone to solving this decision problem, first note that by dropping the constraints C_O in Definition 14 corresponding to the target states, we could instead use the linear program to compute all states reachable in n steps (this will be the set of all values for $\bar{v}_O$ in the feasible set of the linear program) from the linearly-definable set of initial states I. We denote this set by R_n. We also define:

$$\mathcal{R}_n \triangleq \cup^n_{m=1} R_m$$

Notice that $\mathcal{R}_n$ characterises precisely those states that are reachable in n or fewer steps. We now define a notion of a *fixed-point* which gives a sufficient number of steps to consider in order to explore every possible state of our system.

Definition 16 (Fixed-point)**.** We say that n is a *fixed-point* for an LNS if it is the case that $\mathcal{R}_n = \mathcal{R}_{n+1}$.

So, if n is a fixed-point, then $\mathcal{R}_n$ already contains all states reachable from I. This leads us to the following theorem.

Theorem 4. Suppose k is a fixed-point for an LNS and $O \subseteq S$ is a set of target states. Then $\exists n \in \mathbb{N}$ and $\exists \bar{i} \in I$ such that $t^{(n)}_{LE}(\bar{i}) \in O$ iff $\exists n \leq k$ and $\exists \bar{i} \in I$ such that $t^{(n)}_{LE}(\bar{i}) \in O$.

Proof. ($\Rightarrow$) Suppose $t^{(n)}_{LE}(\bar{i}) \in O$ for some $n \in \mathbb{N}$ and $\bar{i} \in I$. Then, $t^{(n)}_{LE}(\bar{i}) \in \mathcal{R}_n$ so, by k being a cut-off, $t^{(n)}_{LE}(\bar{i}) \in \mathcal{R}_k$. The result follows.

($\Leftarrow$) This is immediate. □

This theorem gives us a partial decision procedure for solving the ASR decision problem by calculating $\mathcal{R}_n$ from I for increasing values of n until we find a fixed-point k. Then, by Theorem 4, considering the first k steps of the system's evolution is sufficient to determine all reachable states. In this case we can use the MSR procedure from the previous section. This procedure for ASR is sound but incomplete, as a fixed-point may not exist.

Multi-Step Action Executability

The final decision problem we define concerns determining whether the agent will ever perform a certain action (i.e. whether the neural network controlling the agent may ever produce a certain output). We formalise and solve this problem in the multi-step case, with the restriction to single-step and extension to arbitrary-step reachability being similar to those already presented for state reachability.

Definition 17 (MAE Decision Problem)**.** Let $n \in \mathbb{N}$ and $O \subseteq A$ be a linearly definable set of target actions. The multi-step action executability (MAE) decision problem involves determining whether it is the case that $\exists \bar{i} \in I$ such that $f(t^{(n)}_{LE}(\bar{i})) \in O$, i.e. whether n time steps after starting from $\bar{i}$ the action of the agent is in O.

As for the problems above we first present the relevant MILP encoding.

Definition 18 (MAE Linear Encoding)**.** Let $n \in \mathbb{N}$ and $O \subseteq A$ be a linearly definable set of target actions defined by the set of constraints C_O on vector $\bar{v}_O$. The linear program E^O_n is defined by the set of linear constraints:

$$C_I \cup \{\bar{v}_I = \bar{x}^{(1,1)}\} \cup (\cup_{k=1}^{n}(C_{N,k} \cup C_{LE,k})) \\ \cup C_{N,n+1} \cup \{\bar{x}^{(m,n+1)} = \bar{v}_O\} \cup C_O.$$

Now, we prove that this is a precise characterisation of the actions that are feasible after n time steps.

Theorem 5. The linear program E^O_n is feasible if and only if $\exists \bar{i} \in I$ such that $f(t^{(n)}_{LE}(\bar{i})) \in O$.

Proof sketch. The proof is very similar to that of Theorem 3. □

This theorem gives us a sound and complete decision procedure for the MAE decision problem by computing the corresponding set of linear constraints and then using existing linear programming techniques to verify the feasibility of this linear program.

Complexity We briefly characterise the complexity of the decision problems we have introduced.

Theorem 6. The SSR, MSR and MAE decision problems are NP-complete.

Proof sketch. To see the SSR decision problem is in NP, notice that an input $i \in I$ is a witness that can be checked in polynomial time by feeding the value through the network and checking the linear constraints. To see that it is NP-hard, notice that in the supplementary material of (Katz et al. 2017), an instance of the 3-SAT problem is encoded in a ReLU network along with some constraints on the input and output. We can use the same encoding in the ReLU network of a neural agent and a trivial environment that simply copies the action of the agent into its state. Then, using the same input constraints along with the output constraints relabelled to describe the environment gives an encoding of an instance of the 3-SAT problem into SSR. Thus, SSR is NP-complete.

The proofs for MSR and MAE are similar. □

This lays the foundations for the construction of a toolkit for verifying general neural agent-environment systems. For a given agent implemented as a ReLU-FFNN, we derive a further FFNN approximating the environment to an arbitrary level of precision, and then study the resulting system via linear programming. In the next section we give details of a toolkit constructed in this manner.

5 Implementation and Evaluation

We implemented the methods described in the previous section in a toolkit, called NSVERIFY, that solves the single and multi-step state reachability decision problems and the multi-step action executability problems described in Section 4. The code is released as open-source (NSVerify 2018).

NSVERIFY takes as input a neural agent and a neurally-approximated environment. We experimented with agents' FFNNs trained using Keras, an open-source deep learning toolkit, but other choices are possible.

To answer reachability and executability queries, the toolkit also takes as input a description (in terms of linear constraints) of the input and output sets, and the desired number of steps n to be considered. Upon invocation, it builds the set of linear constraints necessary based on Definition 12, Definition 14 or Definition 18 (depending on which problem we are solving). The constraints are then passed to our MILP back-end, Gurobi ver 7.5.1 (Gu, Rothberg, and Bixby 2016), which is used to determine whether the generated linear program admits a solution. Following the output from Gurobi, a result is then presented to the user as either a **SAT** result, indicating that an output state is reachable (or, an output action executable) from the input sates or an **UNSAT** result representing unreachability of the output states (or, impossibility of the actions). In case of **SAT** a trace showing the states and actions corresponding to the solution are also given.

To evaluate the correctness and the performance of NSVERIFY, we analysed the PENDULUM-V0 task described in Example 4. Since the environment is non-linear we approximated it by producing a neural network (as described in Definition 9). We could do this by training a single network that tries to recreate all the behaviour of the environment. To facilitate the training process, however, we trained several

networks implementing only the non-linear parts of the environment and used a linear combination of them to produce the final result. Specifically, we neurally approximated the $\sin\theta$ and $\cos\theta$ functions.

The results reported in the rest of this section and the next were obtained on a machine running Linux kernel 4.4 on an Intel Core i7-6700 CPU with 16GB of RAM.

Single-Step Experiments

We begin by illustrating the toolkit on single-step reachability properties. We denote by $(\theta_i, \dot{\theta}_i)$ the initial state of the environment and by $(\theta_f, \dot{\theta}_f)$ the final state of the environment after one transition, i.e. $t^{(1)}((\theta_i, \dot{\theta}_i))$.

In the first experiment we restrict the initial states to $-\pi/8 \leq \theta_i \leq -\pi/16$ and $-0.18 \leq \dot{\theta}_i \leq -0.10$. This corresponds to states where the pendulum is leaning to the left and the angular velocity is such that it will be taken even further left at the next time step if the agent does not act appropriately. We wish to check if it is possible to reach a state with $\dot{\theta}_f \leq -0.20$. This corresponds to the angular velocity further increasing in absolute terms following the agent's action, thus leading to the pendulum being even more likely to fall to the left.

By using NSVERIFY we found that the property is satisfiable: from the initial state $(-0.049, -0.165)$, the agent applies a torque of -0.018 resulting in a final state of $(-0.059, -0.200)$. This highlights a possible bug in the synthesised controller, or at least a situation where the agent does not behave optimally; intuitively a positive torque should be applied instead. This information could be used to produce more training examples for the neural network controlling the agent to improve its response.

As a further experiment we considered the analogous scenario with positive values, i.e. the initial states are such that $\pi/16 \leq \theta_i \leq \pi/8$ and $0.10 \leq \dot{\theta}_i \leq 0.18$. We checked whether a target state with $\dot{\theta}_n \geq 0.20$ could be reached. NSVERIFY reported that the property is unsatisfiable, showing that the network behaves as desired on the right side.

In each case, the linear program constructed by NSVERIFY contains 1214 constraints on 975 variables (729 continuous and 246 binary). This linear program takes Gurobi around 50ms to solve.

It follows that by performing two simple experiments we were able to deduce that the agent controller does not behave symmetrically, nor optimally. This may be evidence of bias in the training data or unwanted features in the training algorithm.

Multi-Step Experiments

We now report on experiments involving multi-step reachability. In addition to its intrinsic interest this also enables us to evaluate the scalability of the approach. As before, 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.

		ε			
		$\pi/70$	$\pi/100$	$\pi/200$	$\pi/500$
n	1	0.06s	0.06s	0.06s	0.06s
	2	0.26s	0.16s	0.16s	0.16s
	3	0.68s	1.20s	0.35s	0.34s
	4	1.54s	2.17s	1.69s	1.46s
	5	8.19s	8.26s	7.42s	3.01s
	6	20.29s	16.91s	17.06s	18.44s
	7	38.49s	32.51s	69.95s	29.11s
	8	77.42s	83.29s	149.81s	99.77s

Table 1: The result of checking the property $\theta_f \geq \varepsilon$ after n steps for different values of ε and n. Greyed out cells indicate an **UNSAT** result and white ones a **SAT** result. The time in the cell indicates the time Gurobi took to solve the corresponding LP program constructed by NSVERIFY.

We checked the property $\theta_f \geq \varepsilon$ for different small positive values of ε and a variable number of steps n. Intuitively, this property is unsatisfiable after a sufficient number of steps if the agent is capable of bringing the pendulum close to the vertical (i.e. $\theta_f = 0$) when given sufficient time. Our results are recorded in Table 1.

When the result is **SAT**, it follows that there is a trace of length n after which the pendulum angle remains larger than ε (i.e. the agent has failed to bring the pendulum close to the centre). The toolkit returns this trace; for example, when $\varepsilon = \pi/100$ and $n = 2$ we get an initial state of $(0.049, 0.115)$ at which the agent applies a torque of -1.54 to get a state of $(0.045, -0.075)$. Then, it applies a torque of -0.93 to reach a final state of $(0.037, -0.176)$. So, the agent is correctly applying negative torque to try and get the pendulum vertical, but it does not have enough time to get it within our target.

When the result is **UNSAT**, it follows that after n steps the angle of the pendulum has been brought below ε by the agent. The toolkit reported **UNSAT** after a sufficiently large number of steps, with this number increasing when ε gets closer to 0. This is in line with our intuition as the closer to vertical we require the pendulum to be, the longer it will take the agent to achieve this.

When conducting the experiments the number of constraints and variables in the LP generated by NSVERIFY increased linearly with the number of steps considered. In particular, when considering n steps the resulting problem contained approximately $1214n$ constraints on $973n$ variables ($727n$ continuous and $246n$ binary).

In terms of limitations of the approach, we note that in the experiments the environment is, as explained earlier, a neural approximation; as noted, this can be made arbitrarily precise. Note that the impact of this approximation increases when considering a larger number of steps, thereby requiring increasingly finer environment approximations.

A further source of error arises from rounding errors in how MILP solvers handle real numbers. While this theoretically can lead to invalid results, it is difficult to avoid in any

MILP problem. Indeed we have not encountered issues in the experiments conducted.

Lastly, note that our choice in using Gurobi stems from its attractive performance (Mittelmann 2018) and not from any structural limitation of the approach. We expect the results reported to improve as even more efficient solvers emerge.

All results and experiments here described can be independently verified by running the tool provided in the package (NSVerify 2018).

6 Comparison with Other Toolkits

As discussed in the previous section, NSVERIFY enables the verification of multi-step reachability properties for closed-loop neural agent-environment systems. We are not aware of another tool that enables the verification of such systems.

However, other tools have been released addressing the verification of properties for a single ReLU-FFNN. This problem can be seen as a special instance of the multi-step action feasibility problem for an LNS that we considered in Section 4, where the environment is not considered, i.e. n is fixed to 0. While the alternative toolkits are not comparable on the verification of neural agent-environment systems, in order to evaluate the efficiency of our proposed linear program encoding, we report the results obtained by comparing the performance of NSVERIFY to other ReLU-FFNN verification toolkits on the special case of $n = 0$.

Specifically we compare NSVERIFY against BAB ver 1.0, based on the Branch-and-Bound algorithm (Bunel et al. 2017), and MIP ver 1.0, a re-implementation made by the authors of (Bunel et al. 2017) of the technique originally presented in (Lomuscio and Maganti 2017; Cheng, Nührenberg, and Ruess 2017) based on MILP solving. We also report results obtained by using the SMT-based toolkit RELUPLEX ver 1.0 (Katz et al. 2017), which extends the simplex algorithm to handle ReLU non-linearities, and PLANET ver 1.0 (Ehlers 2017), which uses a combination of SMT and MILP techniques. All these tools, including the one here produced, which derives from (Lomuscio and Maganti 2017), appear to have been developed in parallel and with no interaction between the various research groups.

To conduct the benchmarks we used the TWINSTREAM dataset, released in (Bunel et al. 2017). The networks of the dataset are comprised of two identical streams with the same architecture, weights, and inputs. The benchmark itself consists of 81 properties, with unique network configurations for each property. The final layer of the network computes the difference between the outputs of the two streams, and a positive bias term is finally added. The output of the network is equal to the value of the bias. In the tests we attempted to prove the property that the output of the network is positive, i.e. all cases involve looking for an **UNSAT** result, since all properties are True by construction. We restrict our attention to ReLU networks only in common with the other toolkits.

The results of the comparison are recorded in Table 2. We discuss the relative performance by using three different metrics: the success rate, which represents the percentage of correct results produced by each tool without timing out or running out of memory, the average runtime, and the *number of wins*, representing the number of times a tool managed to solve a property in the quickest time. Each run of TWINSTREAM was conducted with a time-out of two hours on the same machine described above.

Tool	Success rate	Average runtime	Number of wins
NSVERIFY	62.96%	2106.8s	16
RELUPLEX	53.08%	2135.5s	4
MIP	69.14%	2430.2s	7
BAB	29.63%	5212.5s	5
PLANET	64.20%	2807.9s	29

Table 2: Results on the TWINSTREAM dataset.

From Table 2, we see that NSVERIFY has the quickest runtime on average, PLANET most frequently produces a result in the quickest time (it has the highest number of wins) and MIP appears to be the most accurate, producing the highest percentage of correct results. The results suggest that the encodings used in NSVERIFY are, in terms of efficiency, in line with the other tools. It should be noted that further optimisations are possible in all tools considered.

All experimental results from this section can be independently reproduced by using the scripts provided in the package (NSVerify 2018).

7 Conclusions

As argued in the introduction, while methods have been put forward to verify MAS given either declaratively or procedurally, no technique presently exists for the verification of MAS, or any AI system, based on neural networks and interacting with their environment. These systems are being introduced in a variety of applications, including autonomous vehicles; it is therefore essential that novel verification techniques are developed to address these neural architectures.

To move towards this aim, in this paper we have introduced a formal model of a stateless agent, implementing a ReLU-FFNN, interacting with a generic stateful environment. We have shown how several reachability problems can be successfully encoded into mixed integer linear programming problems following a linear encoding of the neurally-approximated environment. We have studied the resulting decision problems and found them to be NP-complete.

In addition to the theoretical evaluation, we have also reported on an experimental evaluation. Specifically, we have introduced NSVERIFY, a novel toolkit for the verification of neural agent-environment systems. We used the toolkit to verify key properties of a simple neural agent-environment system and identify bugs in an agent that was trained with state-of-the-art techniques in machine learning. We are not aware of other methods capable of investigating bugs in neural closed-loop systems.

To evaluate the underlying efficiency of the MILP encoding, we ran benchmarks against all available tools on the single-step action executability problem. While the results are necessarily limited to this specific decision problem, they show that NSVERIFY offers competitive performance compared to other state-of-the-art FFNN verification tools. It

should be noted that further optimisations are certainly possible on all toolkits and that some methods may be more efficient than others depending on the specific scenario.

In future work we intend to address systems resulting from different neural architectures, e.g., recurrent neural networks, as well as different techniques to solve the resulting problem.

8 Acknowledgements

Alessio Lomuscio is supported by a Royal Academy of Engineering Chair in Emerging Technologies. This work is partly funded by DARPA via the Assured Autonomy programme.

References

Alechina, N.; Dastani, M.; Khan, F.; Logan, B.; and Meyer, J. J. C. 2010. *Using Theorem Proving to Verify Properties of Agent Programs*. Springer. 1–33.

Bastani, O.; Ioannou, Y.; Lampropoulos, L.; Vytiniotis, D.; Nori, A. V.; and Criminisi, A. 2016. Measuring neural net robustness with constraints. In *Proc. NIPS16*, 2613–2621.

Bouajjani, A.; Esparza, J.; Schwoon, S.; and Strejček, J. 2005. Reachability analysis of multithreaded software with asynchronous communication. In *Proc. FSTTCS05*, 348–359.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. *CoRR* abs/1606.01540.

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.

Cheng, C.; Nührenberg, G.; and Ruess, H. 2017. Maximum resilience of artificial neural networks. In *Proc. ATVA17*, 251 – 268.

Dantzig, G. B. 1998. *Linear Programming and Extensions*. Princeton University Press.

Ehlers, R. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *Proc. ATVA17*, 269–286.

Eriksson, K.; Estep, D.; and Johnson, C. 2004. *Piecewise Linear Approximation*. 741–753.

Gammie, P., and van der Meyden, R. 2004. MCK: Model checking the logic of knowledge. In *Proc. CAV04*, 479–483.

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

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

Haykin, S. S. 2011. *Neural Networks and Learning Machines*. Pearson Education.

Hornik, K.; Stinchcombe, M.; and White, H. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2(5):359–366.

Huang, X.; Kwiatkowska, M.; Wang, S.; and Wu, M. 2017. Safety verification of deep neural networks. In *Proc. CAV17*, 3–29.

Hudak, S.; Simonak, S.; and Korecko, S. 2010. *Reachability Analysis of Time-Critical Systems*. INTECH Open Access Publisher.

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 *Proc. CAV17*, 97–117.

Kouvaros, P., and Lomuscio, A. 2015. Verifying emergent properties of swarms. In *Proc. IJCAI15*, 1083–1089.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS12*. 1097–1105.

Kurd, Z., and Kelly, T. 2003. Establishing safety criteria for artificial neural networks. In *Proc. KES03*, 163–169.

Lawler, E. L., and Wood, D. E. 1966. Branch-and-bound methods: A survey. *Operations Research* 14(4):699–719.

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

Lomuscio, A., and Michaliszyn, J. 2016. Verification of multi-agent systems via predicate abstraction against ATLK specifications. In *Proc. AAMAS16*, 662–670.

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 *Proc. ICML10*, 807–814.

NSVerify. 2018. Neural System Verify http://vas.doc.ic.ac.uk/software/.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. FOCS77*, 46–57.

Russell, S. J.; Dewey, D.; and Tegmark, M. 2015. Research priorities for robust and beneficial artificial intelligence. *AI Magazine* 36(4).

Shapiro, S.; Lespérance, Y.; and Levesque, H. J. 2002. The cognitive agents specification language and verification environment for multiagent systems. In *Proc. AAMAS02*, 19–26.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Proc. NIPS14*. 3104–3112.

van den Oord, A.; Dieleman, S.; and Schrauwen, B. 2013. Deep content-based music recommendation. In *Proc. NIPS13*. 2643–2651.

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

Wooldridge, M. 2009. *An introduction to MultiAgent systems*. Wiley, second edition.

Zell, A.; Mache, N.; Hübner, R.; Mamier, G.; Vogt, M.; Schmalzl, M.; and Herrmann, K.-U. 1994. *SNNS (Stuttgart Neural Network Simulator)*. In *Neural Network Simulation Environments*. 165–186.