Preface

This volume contains the proceedings of FoMLAS'2020: the 3rd Workshop on Formal Methods for ML-Enabled Autonomous Systems, held on July 21-24, 2020 in Los Angeles (virtually). Each submission was reviewed by 2 program committee members. The committee decided to accept 13 papers.

The FoMLAS workshop is the annual international workshop on formal methods and machine learning. The main goal of the workshop is to facilitate discussions on how formal methods can be used to increase predictability, explainability, and accountability of ML-enabled autonomous system. The workshop program featured one invited talk by Chih-Hong Cheng, and a VNNLIB discussion led by Armando Tacchella.

July 19, 2020
Los Angeles, CA (virtually)

Aws Albarghouthi,
Guy Katz,
Nina Narodytska

Program Committee

Aws Albarghouthi	University of Wisconsin-Madison
Clark Barrett	Stanford University
Chih-Hong Cheng	DENSO AUTOMOTIVE Deutschland GmbH
Arie Gurfinkel	University of Waterloo
Xiaowei Huang	University of Liverpool
Suman Jana	Columbia University
Jean-Baptiste Jeannin	University of Michigan
Susmit Jha	SRI International
Guy Katz	The Hebrew University of Jerusalem
Alessio Lomuscio	Imperial College London
Nina Narodytska	VMware Research
Luca Pulina	University of Sassari
Gagandeep Singh	ETH
Armando Tacchella	Università di Genova
Aleksandar Zeljić	Stanford University
Zhen Zhang	Utah State University

Additional Reviewers

Le, Nham

Author Index

Keyword Index

Table of Contents

Formal Verification of Neural Agents in Non-deterministic Environments*

Michael E. Akintunde, Elena Botoeva, Panagiotis Kouvaros, Alessio Lomuscio

Department of Computing, Imperial College London
London, United Kingdom
{michael.akintunde13,e.botoeva,p.kouvaros,a.lomuscio}@imperial.ac.uk

Abstract. We introduce a model for agent-environment systems where the agents are implemented via feed-forward ReLU neural networks and the environment is non-deterministic. We study the verification problem of such systems against CTL properties. We show that verifying these systems against reachability properties is undecidable. We introduce a bounded fragment of CTL, show its usefulness in identifying shallow bugs in the system, and prove that the verification problem against specifications in bounded CTL is in coNEXPTIME and PSPACE-hard. We present a novel parallel algorithm for MILP-based verification of agent-environment systems, present an implementation, and report the experimental results obtained against a variant of the VerticalCAS use-case.

Keywords: Verification · Neural Systems · Learning-enabled Multi-agent Systems.

1 Introduction

Forthcoming autonomous and robotic systems, including autonomous vehicles, are expected to use machine learning (ML) methods for some of their components. Differently from more conventional AI systems that are programmed directly by engineers, components based on ML are synthesised from data and implemented via neural networks. In an autonomous system these components could execute functions such as perception [35, 26] and control [22, 20]. Employing ML components has considerable attractions in terms of performance (e.g., image classifiers), and, sometimes, ease of realisation (e.g., non-linear controllers). However, it also raises concerns in terms of overall system safety. Indeed, it is known that neural networks, as presently used, are susceptible to adversarial perturbations of known inputs and produce outputs which are difficult to draw immediate conclusions from [37].

If ML components are to be used in safety-critical systems, including various forthcoming autonomous systems, it is essential that they are verified and validated before deployment; standard practice for conventional software. In some

* A shorter version of this article appeared in the Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS20). Auckland, New Zealand. IFAAMAS Press.

areas of AI, notably multi-agent systems (MAS), considerable research has already addressed the automatic verification of AI systems [15, 28]. These concern the validation of either MAS models [25, 33], or MAS programs [4, 10] against expressive AI-inspired specifications, such as those expressible in epistemic and strategy logic. However, with the exceptions discussed below, there is little work addressing the verification of AI systems synthesised from data and implemented via neural networks. This paper makes a contribution in this direction.

Specifically, we formalise and analyse a closed-loop system composed of a reactive neural agent, synthesised from data and implemented by a feed-forward ReLU-activated neural network (ReLU-FFNN), interacting with a non-deterministic environment. Intuitively, the system follows the usual agent-environment loop of observations (of the environment by the agent) and actions (by the agent onto the environment). To model the complexity and partial observability of rich environments, we assume that the neural agent is interacting with a non-deterministic environment, where non-deterministic updates of the environment's state disallow the agent from fully controlling and fully observing the environment's state. Under these assumptions, differently from all related work, the system's evolution is not linear but branching in the future.

We study the verification problem of these systems against a branching time temporal logic. As is known, scalability is a concern in verification and is also an issue in the case of neural systems. To alleviate these difficulties, we are here concerned with a method that is aimed at finding shallow bugs in the system execution, i.e., malfunctions that are realised within a few steps from the system's initialisation. This kind of analysis has been shown to be of particular importance in applications, see, e.g., bounded model checking (BMC) [7], as, experimentally, bugs are often realised after a limited number of steps. Given this, we focus on a bounded version of CTL, i.e., a language expressing temporal properties realisable in a limited number of execution steps. This allows us to reason about applications where the agents ought to bring about a state of affairs within a finite number of steps, or to verify whether a system remains within safety bounds within a number of steps. This enables us to retain decidability even if we consider infinite domains over the reals for the system's state variables, whereas the verification problem for plain CTL is undecidable, as we show. To further alleviate the difficulty of the verification problem, we also introduce a novel algorithm that checks for the occurrence of bugs in parallel over the execution paths. As we show, in the case of bounded safety specifications, this enables us to return a bug to the user as soon as a violation is identified on any of the branching paths that are explored in parallel. This gives considerable advantages in applications, as we show in an avionics application.

A key feature of the parallel verification procedure that we introduce lies in its completeness: we can determine with precision when a potentially infinite set of states (up to a number of steps from the systems's initialisation) satisfies a temporal formula. While this results in a heavier computational cost than some incomplete approaches, there are obvious benefits in precise verification, notably the lack of false positives and false negatives. To the best of our knowledge this

is the first sound and complete verification framework for closed-loop neural systems that accounts for non-deterministic, branching temporal evolutions.

The rest of the paper is organised as follows. After discussing related work, in Section 2 we formally define systems composed by a neural agent, implemented by a ReLU-FFNN, interacting with non-deterministic environments. We analyse the resulting models built on branching executions and define a bounded version of the branching temporal logic CTL to express specifications of these systems. After defining the verification problem, Section 3 introduces monolithic and compositional verification algorithms with a complexity study. In this context we show results ranging from undecidability for unbounded reachability, to coNExpTime upper bound for bounded CTL. We present a toolkit for the practical verification of these systems in Section 4, implementing said procedure, providing additional functionalities, and reporting the experimental results obtained. We conclude in Section 5.

Related Work. In [2] a closed-loop neural agent-environment system was put forward and analysed. Like the present contribution the agent was modelled via a ReLU-FFNN. However, differently from here, a simple deterministic environment was considered. As a consequence, the system executions were linear and only bounded reachability properties were analysed. [1] extended this work to neural agents formalised via recurrent ReLU-activated neural networks and verified the resulting linear system executions against bounded LTL properties. In contrast, the model put forward here can account for complex, partially observable environments resulting in branching traces, and the strictly more expressive specification language allows for existential and universal quantification over paths. In addition, while the papers above focus on sequential verification procedures, we here develop a parallel approach specifically tailored at identifying shallow bugs efficiently. This requires novel verification algorithms and mixed-integer linear programming [40] (MILP) encodings.

A number of other proposals have also addressed the issue of closed loop systems. For example, [21] presents an approach based on hybrid systems to analyse a control-plant where neural networks are synthesised controllers. Their approach is incomparable with the one here pursued, since they target sigmoidal activation functions (while we focus on ReLU activation functions). Also their verification procedure is not complete, while completeness is a key objective here. Similarly, [23, 41, 11, 19] present work addressing closed loop systems with learned controllers and focus on reachable set estimation and, hence, incomplete techniques for such systems.

Lastly, there has been recent activity on complete approaches for verifying standalone ReLU-FFNNs [27, 24, 12, 32, 6, 38]. The systems considered in these approaches are not closed-loop and do not incorporate the environment. This makes the problems considered there different from those analysed here; for instance no temporal evolution can be considered for neural network-controlled agents interacting with an environment.

More broadly, this line of work is related to long standing efforts in BMC [3, 34] that are tailored to finding malfunctions easily accessible from the initial

states. While our approach is technically different from BMC, it shares with it the characteristic of being more efficient than full exploration methods when only a fraction of the model needs to be explored.

2 Neural agent-environment systems

In this section we introduce systems with a neural agent operating on a non-deterministic environment (NANES). These are an extension to non-deterministic environments of the deterministic neural agent-environment systems put forward in [2].

In contrast to traditional models of agency, where the agent's behaviour is given in an agent-based programming language, a NANES accounts for the recent shift to synthesise the agents' behaviour from data [22]; we consider agent protocol functions implemented via feed-forward ReLU neural networks[1] (ReLU-FFNNs) [18]. Differently from [2], following the dynamism and unpredictability of the environments where autonomous agents are typically deployed [29], a NANES models interactions of an agent with a partially observable environment. In this setting an agent cannot observe the full environment state, and therefore cannot deterministically predict the effect of any of its actions.

We now proceed to a formal description of NANES components: a neural agent and a non-deterministic environment. To this end, we fix a set $S \subseteq \mathbb{R}^m$ of environment states and a set $Act \subseteq \mathbb{R}^n$ of actions, for $m, n \in \mathbb{N}$. We assume that the agent is stateless and that its protocol (also known as action policy) has already been synthesised, e.g., via reinforcement learning [36], and is implemented via a ReLU-FFNN or via a piecewise-linear (PWL) combination of them.

Definition 1 (Neural Agents). *Let S be a set of environment states. A* neural agent *(or simply an* agent*) Ag acting on an environment is defined as the tuple $Ag = (Act, prot)$, where:*

- *Act is a set of actions;*
- *$prot : S \to Act$ is a* protocol function *that determines the action the agent will perform given the current state of the environment. Specifically, given ReLU-FFNNs $N_1, \ldots, N_h$ computing functions $f_{N_1}, \ldots, f_{N_h}$, $h \geq 1$, $prot$ is a PWL combination of the latter.*

When $h = 1$, $prot(s)$ can be defined, e.g., as $f_{N_1}(s)$ for $s \in S$.

The environment is stateful and non-deterministically updates its state in response to the actions of the agent.

Definition 2 (Non-deterministic Environments). *An* environment *is a tuple $E = (S, t_E)$, where:*

[1] Specifically, we consider fully-connected feed-forward neural networks where hidden layers are activated by the widely used Rectified Linear Unit (ReLU) activation function [30], which are known to allow FFNNs to generalise well to unseen inputs, defined as $\mathrm{ReLU}(x) \coloneqq \max(0, x)$.

- $S \subseteq \mathbb{R}^m$ *is a set of states.*
- $t_E : S \times Act \to 2^S$ *is a transition function which determines a* finite *set of next possible environment states given its current state and the agent's action.*

Having a finite set of successor environment states allows us to model systems where agents learn a non-deterministic policy to be used in mission-critical decision-making scenarios where the effects of each action are roughly equally optimal in performance, for example, in medical diagnosis systems [14]. Each successor state must therefore be considered with equal importance.

Given the above we can now define a closed-loop system comprising of an agent interacting with an environment.

Definition 3 (NANES). *A* Neural Agent operating on a Non-Deterministic Environment System (NANES) *is a tuple* $\mathcal{S} = (Ag, E, I)$ *where* $Ag = (Act, prot)$ *is a neural agent,* $E = (S, t_E)$ *is an environment, and* $I \subseteq S$ *is a set of initial states for the environment.*

Hereafter we assume the environment's transition function is PWL and its set of initial states is expressible as a set of linear constraints over integer and real-valued variables. Note this does not prevent NANES from modelling a wide class of non-linear environments as these can be approximated to arbitrary precision [9].

With each NANES $\mathcal{S}$ we can associate a model $\mathcal{M}_\mathcal{S}$ capturing its evolutions that is used to interpret temporal specifications.

Definition 4 (Model). *Given a NANES system* $\mathcal{S} = (Ag, E, I)$*, its associated* temporal model $\mathcal{M}_\mathcal{S}$ *is a pair* (R, T) *where* R *is the set of environment states* reachable *from* I *via* T*,* $I \subseteq R \subseteq S$*, and* $T \subseteq R \times R$ *is the successor relation defined by* $(s, s') \in T$ *iff* $s' \in t_E(s, prot(s))$*.*

In the rest of the paper, we assume to have fixed a NANES $\mathcal{S}$ and the associated model $\mathcal{M}_\mathcal{S}$. An $\mathcal{M}_\mathcal{S}$*-path*, or simply *path*, is an infinite sequence of states $s_1 s_2 \ldots$ where $s_i \in R$ and s_{i+1} is a successor of s_i, i.e. $(s_i, s_{i+1}) \in T$, for each $i \geq 1$. Given a path ρ we use $\rho(i)$ to denote the i-th state in ρ. For an environment state $s = (a_1, \ldots, a_m)$, we write $\mathsf{paths}(s)$ to denote the set of all paths originating from s and we use $s.d$ to denote its d-th component a_d.

We verify NANES against properties expressed in a bounded variant of the temporal logic CTL [8]. Inspired by Real-Time Computation Tree Logic (RTCTL) [13], formulae of bounded CTL build upon temporal modalities indexed with natural numbers denoting the temporal depth up to which the formula is evaluated.

Definition 5 (Bounded CTL). *Given a set of environment states* $S \subseteq \mathbb{R}^m$*, the* bounded CTL specification language over linear inequalities*,* $bCTL_{\mathbb{R}^<}$*, is defined by the following BNF:*

$$\varphi ::= \alpha \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid EX^k \varphi \mid AX^k \varphi,$$
$$\alpha ::= c_1(d_1) + \cdots + c_l(d_l)\, op\, c,$$

where $op \in \{<, >\}$*,* $d_i \in \{1, \ldots, m\}$*,* $c_i, c \in \mathbb{R}$*, and* $k \in \mathbb{N}$*.*

Here atomic propositions α are linear constraints on the components of a state. For instance, the atomic proposition $(d_1) + (d_2) < 2$ states that "the sum of the d_1-st and d_2-nd components is less than 2." The temporal formula $EX^k\varphi$ stands for "there is a path such that φ holds after k time steps", whereas $AX^k\varphi$ stands for "in all paths φ holds after k time steps". Moreover, bounded until $E(\varphi U^k\psi)$ ("there is a path such that ψ holds within k time steps, and where φ holds up until then") can be defined by the abbreviations $E(\varphi U^1\psi) \triangleq \psi \vee (\varphi \wedge EX^1\psi)$, and $E(\varphi U^k\psi) \triangleq \psi \vee (\varphi \wedge EX^1E(\varphi U^{k-1}\psi))$ for $k > 1$, and analogously with $A(\varphi U^k\psi)$.

Although $\text{bCTL}_{\mathbb{R}<}$ does not include any form of negation, it still allows us to express arbitrary CTL formulae of bounded temporal depth since it supports all Boolean and temporal operators with their duals. Note also that although $\text{bCTL}_{\mathbb{R}<}$ does not support non-strict inequalities, one can in practice remedy this at the expense of completeness through the use of slack variables [40] to create an approximation of a theoretically closed feasble set, which is common practice when modelled using MILP. Consider for instance an atomic proposition of the form $e \geq c$. It can be replaced with the constraints $e > c - \varepsilon$ and $\varepsilon > 0$, where ε is a small slack variable.

We now define the logic CTL built from the atoms of $\text{bCTL}_{\mathbb{R}<}$.

Definition 6 (CTL). *The* branching-time logic $CTL_{\mathbb{R}<}$ *is defined by the following BNF:*

$$\varphi ::= \alpha \mid \neg\varphi \mid \varphi \vee \varphi \mid AX\varphi \mid AF\varphi \mid E(\varphi U\varphi),$$

where α is an atomic proposition in $bCTL_{\mathbb{R}<}$.

Comparing $\text{bCTL}_{\mathbb{R}<}$ to $\text{CTL}_{\mathbb{R}<}$, on the one hand $AX^k\varphi$ is expressible as $AX(\cdots(AX\varphi)\cdots)$ and $EX^k\varphi$ is expressible as $\neg AX(\cdots(AX\neg\varphi)\cdots)$, where AX is applied k times. On the other hand, $\text{CTL}_{\mathbb{R}<}$ includes the AF ("in all paths eventually") and EU (unbounded until) modalities capable of expressing arbitrary reachability, whereas $\text{bCTL}_{\mathbb{R}<}$ admits bounded specifications only. Note that, while $\text{bCTL}_{\mathbb{R}<}$ is clearly less expressive than $\text{CTL}_{\mathbb{R}<}$, it still captures properties of interest. Notably, *bounded safety* is expressible in $\text{bCTL}_{\mathbb{R}<}$ as $AG^k safe \triangleq AX^1 safe \wedge \cdots \wedge AX^k safe$ stating that every state on every path is safe within the first k steps.

We interpret $\text{bCTL}_{\mathbb{R}<}$ formulae on a temporal model as follows.

Definition 7 (Satisfaction).

For a model $\mathcal{M}_{\mathcal{S}}$, an environment state s, and a $bCTL_{\mathbb{R}<}$ formula φ, the satisfaction *of φ at s in $\mathcal{M}_{\mathcal{S}}$, denoted $(\mathcal{M}_{\mathcal{S}}, s) \models \varphi$, or simply $s \models \varphi$ when $\mathcal{M}_{\mathcal{S}}$ is clear from the context, is inductively defined as follows:*

$$\begin{aligned}
&s \models c_1(d_1) + \cdots + c_l(d_l)\, op\, c && \text{iff } \textstyle(\sum_{i=1}^{l} c_i \cdot s.d_i)\, op\, c;\\
&s \models \varphi \vee \psi && \text{iff } s \models \varphi \text{ or } s \models \psi;\\
&s \models \varphi \wedge \psi && \text{iff } s \models \varphi \text{ and } s \models \psi;\\
&s \models EX^k\varphi && \text{iff there is } \rho \in \mathsf{paths}(s) \text{ such that } \rho(k) \models \varphi;\\
&s \models AX^k\varphi && \text{iff for all } \rho \in \mathsf{paths}(s) \text{ we have } \rho(k) \models \varphi.
\end{aligned}$$

We assume the usual definition of satisfaction for $\text{CTL}_{\mathbb{R}<}$; this can be given as standard by using the atomic case from Definition 7.

A specification φ is said to be *satisfied by* $\mathcal{S}$ if $(\mathcal{M}_{\mathcal{S}}, s) \models \varphi$ for all initial states $s \in I$. We denote this by $\mathcal{S} \models \varphi$. It follows that, for example, to check bounded safety we need to verify that from all (possibly infinitely many) initial states no state (out of possibly infinitely many) within the first k evolutions is an unsafe state. This is the basis of the verification problem that we define below.

Definition 8 (Verification problem). *Given a NANES $\mathcal{S}$ and a formula φ, determine whether $\mathcal{S} \models \varphi$.*

In the next section we study the decidability and complexity of the verification problem here introduced.

3 The Verification Problem

In this section we study the verification problem for a NANES against CTL and $\text{bCTL}_{\mathbb{R}<}$ specifications. First, we show that verifying against CTL formulae is undecidable, already for deterministic environments and simple reachability properties. In the rest of the section, we focus on bounded CTL, where we develop a decision procedure for the verification problem based on producing a single MILP and checking its feasibility. Then we devise a parallelisable version of the procedure that produces multiple MILPs and that can be particularly efficient at finding counter-examples for bounded safety properties. Following this, we analyse the computational complexity of the verification problem against $\text{bCTL}_{\mathbb{R}<}$ formulae.

3.1 Unbounded CTL

In this subsection we show undecidability of the verification problem for deterministic NANES against simple reachability properties, where a deterministic NANES is a tuple $(Ag = (Act, prot), E = (S, t_E), I)$, where $|t_E(s, a)| = 1$ for all $s \in S$ and $a \in Act$. The undecidability result for arbitrary NANES and full CTL follows.

Theorem 1. *Verifying deterministic NANES against formulae of the form $AF\alpha$ is undecidable.*

Proof (Proof Sketch). We can show the result by reduction from the Halting problem of a deterministic Turing machine (DTM) M on an input string ω_0, whose tape alphabet consists of symbols 0, 1 and 2, with one halting state (the accepting state).

The idea of the reduction is to construct a NANES $\mathcal{S} = (Ag, E, I)$ such that each state of $\mathcal{S}$ encodes the current configuration of the DTM, i.e., the current state of M, the symbol under the head, and the contents of the tape to the left and right of the head as two real numbers (the former one is read from right to

left). I consists of a single state and encodes q_0 (the initial state of M) and ω_0. The run of M on its input can be simulated by appropriately updating the state using the environment transition function (while the agent does not need to do anything). Conversely, it is possible to shift all the logic to the agent's protocol function with a trivial environment.

Finally, we verify $\mathcal{S}$ against the reachability specification φ of the form *AF accept*, where *accept* encodes that M is in the accepting state. Then $\mathcal{S} \models \varphi$ iff M halts on ω_0. It can also be checked that the required environment transition function and the agent's protocol function can be implemented as piecewise-linear functions.

We observe that the above result holds even for strongly restricted NANES where either the protocol or the transition function is linear (but not both at the same time). As a corollary, we obtain undecidability of the verification problem against full CTL.

Corollary 1. *Verifying NANES against $CTL_{\mathbb{R}<}$ formulae is undecidable.*

3.2 Bounded CTL

We now proceed to investigate the verification problem for the bounded CTL specification language. We start by showing an auxiliary result that allows us to assume without loss of generality that the cardinality of $t_E(s,a)$ is the same for each state s and action a.

Lemma 1. *Given a NANES $\mathcal{S} = ((Act, prot), (S, t_E), I)$ and specification $\varphi \in bCTL_{\mathbb{R}<}$, there is:*

- *a NANES $\mathcal{S}' = ((Act, prot'), (S', t'_E), I')$ such that $|t'_E(s_1, a_1)| = |t'_E(s_2, a_2)|$ for all $s_1, s_2 \in S'$ and $a_1, a_2 \in Act$,*
- *a specification $\varphi' \in bCTL_{\mathbb{R}<}$ such that $\mathcal{S} \models \varphi$ iff $\mathcal{S}' \models \varphi'$.*

Proof (Proof Sketch). Consider $b = \max_{s \in S, a \in Act} |t_E(s,a)|$. Define the components of $\mathcal{S}'$ such that $|t'_E(s,a)| = b$ for all $s \in S'$, $a \in Act$, and $\mathcal{S} \models \varphi$ iff $\mathcal{S}' \models \varphi'$. S' and I' are defined by $S' = S \times \{0,1\}$ and $I' = I \times \{1\}$. The added dimension indicates whether a state is valid (1) or not (0). The agent's protocol function $prot'$ is defined as $prot'((s,f)) = prot(s)$ for each $s \in S$, $f \in \{0,1\}$. The transition function $t'_E((s,f),a)$ returns $t_E(s,a) \times \{1\} \cup \{(s_1,0), \ldots, (s_{b-l},0)\}$ where $|t_E(s,a)| = l$ and $s_1, \ldots, s_{b-l}$ are pairwise distinct states from S. The formula φ' is a copy of φ with atomic propositions α replaced with $\alpha \wedge ((m+1) > 0.9)$, where $S = \mathbb{R}^m$.

In the rest of this section we assume that $|t_E(s,a)| = b$ for all s and a, and that t_E is given as b piecewise-linear (PWL) functions $t_i : \mathbb{R}^{m+n} \to \mathbb{R}^m$. We remark that b represents the maximum branching factor of the environment, and does not constrain each state-action pair to exactly b successor states. Note that this assumption is used when devising the verification procedure presented below.

The procedure that we put forward recasts the verification problem to MILP. It is well known that a PWL function can be MILP-encoded using the "Big-M" method [16]. For instance, the pairs (x, y), where $y = \text{ReLU}(x)$ and $x \in [l, u]$ can be found as solutions to the following set of MILP constraints that use the binary variable δ, real-valued variables x and y and constants l and u:

$$y \geq 0, y \geq x, y \leq u \cdot \delta, y \leq x - l \cdot (1 - \delta)$$

Here, when $\delta = 1$, the constraints imply that $y = x$ and $x \geq 0$, and when $\delta = 0$, the constraints imply that $y = 0$ and $x \leq 0$. Since the function computed by a ReLU-activated neural network can be obtained via successive compositions of the ReLU function and linear transformations, its MILP encoding can be obtained via the composition of constraints of the above form with appropriate linear constraints. The resulting overall MILP is of linear size in the size of the network. Further details of the Big-M encoding of the ReLU function can be found in [38, 27].

Given a MILP program π, we use $vars(\pi)$ to denote the set of variables in π. Denote by $\mathfrak{a}$ the *assignment function* $\mathfrak{a} : vars(\pi) \to \mathbb{R}$, which defines the specific (binary, integer or real) value assigned to a MILP program variable. We write $\mathfrak{a} \models \pi$ if $\mathfrak{a}$ *satisfies* π, i.e., if $\mathfrak{a}(\delta) \in \{0, 1\}$ for each binary variable δ, $\mathfrak{a}(\iota) \in \mathbb{N}$ for each integer variable ι, and all constraints in π are satisfied. Hereafter, we will denote by boldface font tuples of MILP variables (of length m for $S \subseteq \mathbb{R}^m$ the set of environment states) representing an environment state and call them *state variables*.

Monolithic Encoding. We now give a recursive encoding of the verification problem into a single MILP. As a stepping stone, we first encode the computation of a successor environment state as a composition of the protocol function *prot* and of the transition functions t_i. By assumption, *prot* and each t_i is a PWL function, and so the predicate $\boldsymbol{y} = t_i(\boldsymbol{x}, prot(\boldsymbol{x}))$ is expressible as a set of MILP constraints by means of the Big-M method, which we denote by $C_i(\boldsymbol{x}, \boldsymbol{y})$ (note that $\boldsymbol{x} \cup \boldsymbol{y} \subset vars(C_i(\boldsymbol{x}, \boldsymbol{y}))$). Solutions of $C_i(\boldsymbol{x}, \boldsymbol{y})$ represent pairs of consecutive environment states [2]:

Lemma 2. *Let $C_i(\boldsymbol{x}, \boldsymbol{y})$ be a MILP program corresponding to $\boldsymbol{y} = t_i(\boldsymbol{x}, prot(\boldsymbol{x}))$. Given two states s and s' in $\mathcal{M}_\mathcal{S}$, we have that $s' = t_i(s, prot(s))$ iff there is an assignment $\mathfrak{a}$ to $vars(C_i(\boldsymbol{x}, \boldsymbol{y}))$ such that $s = \mathfrak{a}(\boldsymbol{x})$, $s' = \mathfrak{a}(\boldsymbol{y})$, and $\mathfrak{a} \models C_i(\boldsymbol{x}, \boldsymbol{y})$.*

Denote by $\text{bCTL}_{\mathbb{R}\leq}$ the bounded CTL language over atomic propositions α where $op \in \{\leq, \geq\}$ (i.e., linear constraints over non-strict inequalities). As a second step, given a NANES $\mathcal{S}$ and a formula $\varphi \in \text{bCTL}_{\mathbb{R}\leq}$, we construct a MILP program $\pi_{\mathcal{S},\varphi}$, whose feasibility corresponds to the existence of a state in $\mathcal{M}_\mathcal{S}$ that satisfies φ. For ease of presentation, and without loss of generality, we assume that φ may contain only the temporal modalities EX^1 and AX^1,

$$
\begin{array}{rcl}
\pi_{\mathcal{S},\alpha}(\boldsymbol{x}) & = & \{C_\alpha(\boldsymbol{x})\}, \\
 & & \quad \text{where } C_\alpha(\boldsymbol{x}) \text{ is defined as } c_1 x_{d_1} + \cdots + c_l x_{d_l}\, op\, c \text{ for} \\
 & & \quad \alpha = c_1(d_1) + \cdots + c_l(d_l)\, op\, c \text{ and } \boldsymbol{x} = (x_1, \ldots, x_m), \\
\pi_{\mathcal{S},\varphi_1 \vee \varphi_2}(\boldsymbol{x}) & = & (\delta = 1) \Rightarrow (\pi_{\mathcal{S},\varphi_1}(\boldsymbol{x_1}) \cup \{\boldsymbol{x} = \boldsymbol{x}_1\}) \cup \\
 & & (\delta = 0) \Rightarrow (\pi_{\mathcal{S},\varphi_2}(\boldsymbol{x_2}) \cup \{\boldsymbol{x} = \boldsymbol{x}_2\}), \\
 & & \quad \text{where the binary variables } \delta, \delta_1, \ldots, \delta_b, \text{ the state variables} \\
 & & \quad \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_b, \boldsymbol{y}, \text{ and all auxiliary variables in } C_i(\boldsymbol{x}, \boldsymbol{y}_i) \\
 & & \quad \text{are fresh}, \\
\pi_{\mathcal{S},\varphi_1 \wedge \varphi_2}(\boldsymbol{x}) & = & \pi_{\mathcal{S},\varphi_1}(\boldsymbol{x}) \cup \pi_{\mathcal{S},\varphi_2}(\boldsymbol{x}), \\
\pi_{\mathcal{S},EX\varphi}(\boldsymbol{x}) & = & \bigcup_{i=1}^{b} (\delta_i = 1) \Rightarrow (C_i(\boldsymbol{x}, \boldsymbol{y}_i) \cup \{\boldsymbol{y} = \boldsymbol{y}_i\}) \cup \{\delta_1 + \cdots + \delta_b = 1\} \cup \pi_\varphi(\boldsymbol{y}), \\
\pi_{\mathcal{S},AX\varphi}(\boldsymbol{x}) & = & C_1(\boldsymbol{x}, \boldsymbol{y}_1) \cup \cdots \cup C_b(\boldsymbol{x}, \boldsymbol{y}_b) \;\cup\; \pi_{\mathcal{S},\varphi}(\boldsymbol{y}_1) \cup \cdots \cup \pi_{\mathcal{S},\varphi}(\boldsymbol{y}_b).
\end{array}
$$

Fig. 1. Monolithic encoding $\pi_{\mathcal{S},\varphi}$ for $\varphi \in \mathrm{bCTL}_{\mathbb{R}^{\leq}}$.

for which we write EX and AX, respectively[2]. We make use of the *indicator constraints* of the form $(\delta = v) \Rightarrow c$, for a binary variable δ, binary value $v \in \{0, 1\}$ and a linear constraint c, meaning that whenever the value of δ is v, the constraint c should hold. In particular, indicator constraints can be used to naturally express disjunctive cases. For instance, the disjunction $x = 3 \vee x = 5$ can be encoded using two auxiliary binary variables δ_i, $i \in \{1, 2\}$, and the following set of MILP constraints:

$$(\delta_1 = 1) \Rightarrow x = 3,\ (\delta_2 = 1) \Rightarrow x = 5,\ \delta_1 + \delta_2 = 1.$$

Here, the constraint $\delta_1 + \delta_2 = 1$ ensures that at least one of the two clauses is satisfied (note that, in general, the above encoding does not rule out an assignment where both clauses are satisfied at the same time). Given a binary variable δ and a set of constraints π, we hereafter abbreviate $\{(\delta = v) \Rightarrow c \mid c \in \pi\}$ to $(\delta = v) \Rightarrow \pi$.

We now define the *monolithic* encoding $\pi_{\mathcal{S},\varphi}$.

Definition 9. *Given a NANES $\mathcal{S}$ and a formula $\varphi \in bCTL_{\mathbb{R}^{\leq}}$, their* monolithic MILP encoding $\pi_{\mathcal{S},\varphi}$ *is defined as the MILP program $\pi_{\mathcal{S},\varphi}(\boldsymbol{x})$, where $\boldsymbol{x}$ is a tuple of fresh state variables, and $\pi_{\mathcal{S},\varphi}(\boldsymbol{x})$ is built inductively using the rules in Figure 1.*

In the encoding in Figure 1, the base case $\pi_{\mathcal{S},\alpha}(\boldsymbol{x})$ for an atom α produces the MILP program consisting of a single linear constraint corresponding to α and using variables in $\boldsymbol{x}$. Each inductive case depends on the state variables $\boldsymbol{x}$ but might in turn generate programs for subformulas which depend on freshly created state variables different to $\boldsymbol{x}$ (such as $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{y}$, etc). All other auxiliary variables employed in the encoding are also fresh, preventing undesirable interactions between unrelated branches of the program.

[2] For $Q \in \{A, E\}$, the formula $QX^k\varphi$ is equivalent to $QX^1(\cdots(QX^1\varphi)\cdots)$ where QX^1 is applied k times to φ and which grows linearly in k assuming unary encoding of numbers.

- Disjunctions use a binary variable δ and two sets of indicator constraints. In a feasible assignment $\mathfrak{a}$, when δ is 1, φ_1 is satisfied and the variables $\boldsymbol{x}$ take the values of the variables $\boldsymbol{x}_1$, while when δ is 0, φ_2 is satisfied and $\boldsymbol{x}$ takes the values of $\boldsymbol{x}_2$.
- We encode conjunction as the union of the constraints for each of the conjuncts, which all must be satisfied at the same time.
- We encode EX via b-ary disjunction: there are b possible next states and each disjunct chooses one of them by ensuring that the relevant $C_i(\boldsymbol{x}, \boldsymbol{y}_i)$ is satisfied. The variables for the successor state $\boldsymbol{y}$ are assigned accordingly to this choice; moreover, the subprogram for φ depends on them. Notably, only one copy of $\pi_{\mathcal{S},\varphi}$ is required.
- To satisfy $AX\varphi$, all b possible successor states should satisfy φ, and so we take the union of all $C_i(\boldsymbol{x}, \boldsymbol{y}_i)$ and of b copies of $\pi_{\mathcal{S},\varphi}$, each depending on one of the successor state variables $\boldsymbol{y}_i$.

Note that the size of $\pi_{\mathcal{S},\varphi}$ may grow exponentially due to b repetitions of $\pi_{\mathcal{S},\psi}$ in $\pi_{\mathcal{S},AX\psi}(\boldsymbol{x})$; for $\varphi = AX^k\alpha$, the size of $\pi_{\mathcal{S},\varphi}$ is $O(k \cdot b^k \cdot |\mathcal{S}|)$. The same estimate works in the general case for the temporal bound k of φ. On the other hand, when φ contains no AX operator, the size of $\pi_{\mathcal{S},\varphi}$ remains polynomial $O(k \cdot b \cdot |\mathcal{S}|)$.

We can prove that $\pi_{\mathcal{S},\varphi}$ is as intended.

Lemma 3. *Given a NANES $\mathcal{S}$, a formula $\varphi \in bCTL_{\mathbb{R}^{\leq}}$ and a state s in $\mathcal{M}_{\mathcal{S}}$, the following are equivalent:*

1. $s \models \varphi$.
2. *There exists an assignment $\mathfrak{a}$ to $vars(\pi_{\mathcal{S},\varphi}(\boldsymbol{x}))$ such that $\mathfrak{a} \models \pi_{\mathcal{S},\varphi}(\boldsymbol{x})$ and $s = \mathfrak{a}(\boldsymbol{x})$.*

Finally, we can exploit Lemma 3 to devise a procedure that solves the verification problem by restricting $\boldsymbol{x}$ to the initial states of $\mathcal{S}$. The procedure is given by Algorithm 1. Its soundness and completeness is shown by the following.

Theorem 2. *Given a NANES $\mathcal{S}$ and a formula $\varphi \in bCTL_{\mathbb{R}^{<}}$, Algorithm 1 returns* False iff $\mathcal{S} \not\models \varphi$.

Proof. Suppose that Algorithm 1 returns *False*. It follows that $\pi_{\mathcal{S},\neg\varphi\wedge\varphi_I}(\boldsymbol{x})$ is feasible. So, by Lemma 3, there exists an assignment $\mathfrak{a}$ to $vars(\pi_{\mathcal{S},\neg\varphi\wedge\varphi_I}(\boldsymbol{x}))$ such that $\mathfrak{a} \models \pi_{\mathcal{S},\neg\varphi\wedge\varphi_I}(\boldsymbol{x})$. Moreover, for $s = \mathfrak{a}(\boldsymbol{x})$, we have that $s \models \varphi_I$ and $s \models \neg\varphi$. It follows that $s \in I$ and $s \not\models \varphi$, and consequently, $\mathcal{S} \not\models \varphi$. Conversely, if there exists $s \in I$ such that $s \not\models \varphi$, we obtain that there is an assignment satisfying $\pi_{\mathcal{S},\neg\varphi\wedge\varphi_I}(\boldsymbol{x})$, and therefore Algorithm 1 returns *False*.

Recall that strict inequalities are not supported in the MILP solver. Therefore note that we only pass $\neg\varphi$ (the negation of the specification), and the initial state φ_I (expressed only in terms of non-strict intequalities) to the MILP solver in *negation normal form* (NNF). In this process, negation is eliminated by pushing it down and through the atoms resulting in all strict inequalities of atoms of the original specification φ being converted to non-strict inequalities.

Algorithm 1 The monolithic MILP verification procedure.

```
1: procedure MONO-VERIFY(S, φ)
2:     Input: NANES S = (Ag, E, I); formula φ ∈ bCTL_{R<}
3:     Output: True/False
4:     φ_I ← Boolean formula representing I
5:     φ' ← NNF(¬φ ∧ φ_I)
6:     π_{S,φ'} ← MILP associated with S and φ'
7:     feasible ← MILP_SOLVER(π_{S,φ'})
8:     return ¬feasible
```

Compositional Encoding. Observe that due to its handling of disjunctions, the previously introduced encoding $\pi_{\mathcal{S},\varphi}$ might result in excessively large programs whose feasibility is a computationally expensive task. We now propose a different encoding that instead of delegating disjunction to the MILP solver (the $\varphi_1 \vee \varphi_2$ and $EX\varphi$ cases) creates a separate program for each disjunct. More specifically, for a formula φ, we define a set $\Pi_{\mathcal{S},\varphi}$ of MILP programs with the property that there exists a state s in $\mathcal{M}_{\mathcal{S}}$ such that $s \models \varphi$ iff at least one of the programs in $\Pi_{\mathcal{S},\varphi}$ is feasible. A specific feature of this encoding lies in its parallelisability. Due to this, it is particularly amenable to efficiently finding bugs that can be reached within a few steps along some of the paths from the initial states, similarly to BMC [7]. We demonstrate this experimentally in the next section after having introduced the encoding here.

Below, given a set C of linear constraints, we write $[C]$ to denote the respective MILP program. Given sets $A = \{[A_1], \ldots, [A_p]\}$ and $B = \{[B_1], \ldots, [B_q]\}$ of MILP programs, we write $A \times B$ to denote the *product* of A and B computed as $\{[A_i \cup B_j] \mid i = 1, .., p,\ j = 1, .., q\}$.

Definition 10. *Given a NANES $\mathcal{S}$ and a formula $\varphi \in bCTL_{\mathbb{R}\leq}$, their* compositional MILP encoding *$\Pi_{\mathcal{S},\varphi}$ is defined as the set of MILP programs $\Pi_{\mathcal{S},\varphi}(\boldsymbol{x})$, where $\boldsymbol{x}$ is a tuple of fresh state variables, and $\Pi_{\mathcal{S},\varphi}(\boldsymbol{x})$ is built inductively using the rules in Figure 2.*

$$
\begin{array}{lcl}
\Pi_{\mathcal{S},\alpha}(\boldsymbol{x}) & = & \{[C_\alpha(\boldsymbol{x})]\}, \\
\Pi_{\mathcal{S},\varphi_1 \vee \varphi_2}(\boldsymbol{x}) & = & \Pi_{\mathcal{S},\varphi_1}(\boldsymbol{x}) \cup \Pi_{\mathcal{S},\varphi_2}(\boldsymbol{x}), \\
\Pi_{\mathcal{S},\varphi_1 \wedge \varphi_2}(\boldsymbol{x}) & = & \Pi_{\mathcal{S},\varphi_1}(\boldsymbol{x}) \times \Pi_{\mathcal{S},\varphi_2}(\boldsymbol{x}), \\
\Pi_{\mathcal{S},EX\varphi}(\boldsymbol{x}) & = & \bigcup_{i=1}^{b} \{[C_i(\boldsymbol{x},\boldsymbol{y})]\} \times \Pi_{\mathcal{S},\varphi}(\boldsymbol{y}), \\
 & & \text{where the state variables } \boldsymbol{y} \text{ and all remaining variables in } C_i(\boldsymbol{x},\boldsymbol{y}) \text{ are fresh,} \\
\Pi_{\mathcal{S},AX\varphi}(\boldsymbol{x}) & = & \bigtimes_{i=1}^{b} \{[C_i(\boldsymbol{x},\boldsymbol{y}_i)]\} \times \Pi_{\mathcal{S},\varphi}(\boldsymbol{y}_i), \\
 & & \text{where the state variables } \boldsymbol{y}_1, \ldots, \boldsymbol{y}_b \\
 & & \text{and all remaining variables in } C_i(\boldsymbol{x},\boldsymbol{y}_i) \\
 & & \text{are fresh.}
\end{array}
$$

Fig. 2. Compositional encoding $\Pi_{\mathcal{S},\varphi}$ for $\varphi \in \text{bCTL}_{\mathbb{R}\leq}$.

Algorithm 2 The compositional MILP verification procedure.

```
1: procedure COMP-VERIFY(𝒮, φ)
2:     Input: NANES 𝒮 = (Ag, E, I); formula φ ∈ bCTL_{ℝ<}
3:     Output: True/False
4:     feasible ← False
5:     φ_I ← Boolean formula representing I
6:     φ' ← NNF(¬φ ∧ φ_I)
7:     Π_{𝒮,φ'} ← Set of MILPs associated with 𝒮 and φ'
8:     for π in Π_{𝒮,φ'} do
9:         aux ← MILP_SOLVER(π)
10:        if aux is True then
11:            feasible ← True
12:            break
13:    return ¬feasible
```

Following the monolithic encoding in Figure 1, in Figure 2 $C_\alpha(\boldsymbol{x})$ is the linear constraint corresponding to the atomic proposition α defined over $\boldsymbol{x}$. We use the same convention regarding the state and auxiliary variables of subprograms. In $\Pi_{\mathcal{S},\varphi}$ every program π represents one of the encodings of φ.

- For disjunction we take the union of the two sets of encodings.
- Every encoding of $\varphi_1 \wedge \varphi_2$ consists of an encoding of φ_1 and of an encoding of φ_2, therefore we take the product of the two sets.
- Every encoding of $EX\varphi$ is an encoding of φ extended with the constraints $C_i(\boldsymbol{x}, \boldsymbol{y})$ for a single i.
- Every encoding of $AX\varphi$ consists of b (possibly different) encodings of φ extended with the constraints $C_i(\boldsymbol{x}, \boldsymbol{y}_i)$ for $i = 1, \ldots, b$.

The set $\Pi_{\mathcal{S},\varphi}$ grows exponentially with the temporal depth of φ; however each program in the set can be smaller than the monolithic MILP $\pi_{\mathcal{S},\varphi}$.

Similarly to Lemma 3 we can prove that $\Pi_{\mathcal{S},\varphi}$ is as intended.

Lemma 4. *Given a NANES $\mathcal{S}$, a formula $\varphi \in bCTL_{\mathbb{R}\leq}$ and a state s in $\mathcal{M}_{\mathcal{S}}$, the following are equivalent:*

1. *$s \models \varphi$.*
2. *There is a MILP $\pi(\boldsymbol{x}) \in \Pi_{\mathcal{S},\varphi}(\boldsymbol{x})$ and an assignment $\mathfrak{a}$ to $vars(\pi(\boldsymbol{x}))$ such that $s = \mathfrak{a}(\boldsymbol{x})$ and $\mathfrak{a} \models \pi(\boldsymbol{x})$.*

Based on Lemma 4, we can devise a verification procedure that searches for a feasible MILP in the set of MILPs generated by the encoding of Figure 2. This procedure is presented in Algorithm 2. Importantly, it naturally lends itself to parallelisation when checking feasibility of the generated programs. In turn this enables us to check in parallel for a possible falsification of formulas in which the temporal operator is universally quantified as in $AX^k\alpha$. As we will see in the next section, this will become particularly useful when verifying bounded safety.

Computational Complexity. Finally we study the complexity of the verification problem for $\text{bCTL}_{\mathbb{R}<}$. The upper bound follows from the monolithic

verification procedure and the lower bound can be obtained by reduction from the validity problem of QBF.

Theorem 3. *Verifying NANES against* $bCTL_{\mathbb{R}<}$ *is in co*NExpTime *and is* PSpace*-hard in combined complexity.*

We also show that the complexity of the verification problem is reduced to coNP for the bounded safety fragment of $\text{bCTL}_{\mathbb{R}<}$.

Corollary 2. *Verifying NANES against bounded safety properties is coNP-complete in combined complexity.*

Proof (Proof Sketch). The upper bound follows from the fact that we can check whether a property $\varphi = AG^k \mathit{safe}$ is not satisfied by $\mathcal{S}$ by guessing an initial state s and a path ρ of length k originating from s, and by verifying that $\rho(i) \not\models \mathit{safe}$ for some $i = 1, \ldots, k$. If such an initial state s exists, then there exists an initial state s' with the same properties of polynomial size. This follows from the encoding into MILP and the fact that if a MILP instance is feasible, there is a solution of polynomial size. The lower bound can be adapted from the NP lower bound of the satisfiability problem of neural networks properties [24], and holds already for one-step formulae.

4 Implementation and Experiments

We have implemented the verification procedures described in the previous section in an open source toolkit called NANESVerify [31]. The tool takes as input a $\text{bCTL}_{\mathbb{R}<}$ specification φ and a NANES $\mathcal{S}$ in the form of ReLU-FFNNs implementing the agent, piecewise linear (PWL) functions (possibly given as ReLU-FFNNs) implementing the environment and a set I of the initial states in the form of a hyper-rectangle which can be encoded as $x_1 \geq l_1 \wedge x_1 \leq u_1 \wedge \cdots \wedge x_m \geq l_m \wedge x_m \leq u_m$ for hyper-rectangle $[l_1, u_1] \times \cdots \times [l_m, u_m]$ and state variables $\boldsymbol{x} = (x_1, \ldots, x_m)$. The top-level call to the tool returns `True` if φ is satisfied on $\mathcal{S}$, and returns `False` if φ fails for some initial state of $\mathcal{S}$. In the latter case, a trace in the form of state-action pairs is produced, giving an example run of the system which failed to satisfy the specification.

The user can specify a parameter to determine whether the monolithic or compositional procedure with parallel or sequential execution is to be used. When using sequential execution, NANESVerify follows the respective procedures from Algorithms 1 and 2. For the compositional procedure with parallel execution, NANESVerify performs the computation in line 9 of Algorithm 2 asynchronously across eight worker processes running a separate Gurobi instance for each MILP. The main process finishes either when a MILP-solving job terminates with a feasible solution (finding a counter-example), all jobs gave infeasible results, or no result was returned within a given time limit.

We used Python to implement the tool and relied on Gurobi ver. 8.1 [17] as a back-end to resolve the feasibility of the generated MILP problems. When

constructing the Big-M encodings of the neural networks, the lower and upper bounds for each neuron are determined using symbolic linear relaxation [39] starting from the bounds of the input nodes given by I. For other MILP variables encountered, we propagate their bounds throughout the encoding using interval arithmetic. For the compositional encoding, we delegate disjunctions at the level of atomic propositions to the MILP solver, which avoids the unnecessary blow-up of the number of MILPs generated and can still be efficiently handled by the solver.

Aircraft Collision Avoidance System Example. To validate the toolkit we use a scenario involving two aircraft, the *ownship* and the *intruder*, where the ownship is equipped with a collision avoidance system referred to as VerticalCAS [23]. The intruder is assumed to follow a constant horizontal trajectory. VerticalCAS once every second issues vertical climbrate advisories to the ownship pilot, to avoid a *near mid-air collision (NMAC)*, a region where the ownship and intruder are separated by less than 100ft vertically and 500ft horizontally. The possible advisories are:

1) COC: Clear Of Conflict.
2) DNC: Do Not Climb.
3) DND: Do Not Descend.
4) DES1500: Descend at least 1500 ft/s.
5) CL1500: Climb at least 1500 ft/s.
6) SDES1500: Strengthen Descent to at least 1500 ft/s.
7) SCL1500: Strengthen Climb to at least 1500 ft/s.
8) SDES2500: Strengthen Descent to at least 2500 ft/s.
9) SCL2500: Strengthen Climb to at least 2500 ft/s.

The advisories instruct the pilot to accelerate until the vertical climbrate of the ownship complies with the advisory. For some advisories, e.g. DND, the pilot can choose any acceleration in $[g/4, g/3]$, where g represents the gravitational constant 32.2 ft/s^2. We hereafter denote by $[m]$ the set $\{1, .., m\}$. The set of tuples $S = (h, \dot{h}_0, \tau, \text{adv}) \in [-3000, 3000] \times [-2500, 2500] \times [0, 40] \times [9]$ describe an ownship–intruder *encounter*, where:

1) h (ft): Intruder altitude relative to ownship.
2) $\dot{h}_0$ (ft/s): Ownship vertical climbrate.
3) τ (s): Time to loss of horizontal separation.
4) adv: The previous advisory issued by VerticalCAS.

The vertical geometry of the encounter is given by h and $\dot{h}_0$, and τ reports the seconds until the ownship (black) and intruder (red) are no longer horizontally separated, illustrated in Fig. 3. The VerticalCAS system is composed of nine ReLU-FFNNs $F = \left\{ \left(f_{N_i} : \mathbb{R}^3 \to \mathbb{R}^9 \right) : i \in [9] \right\}$, one for each advisory, with three inputs $(h, \dot{h}_0, \tau)$, five fully-connected hidden layers of 20 units each, and nine outputs representing the score of each possible advisory.

NANES Encoding. We model VerticalCAS as a neural agent with protocol function $prot(s) = \arg\max(\mathsf{apply}(\mathsf{select}(s), s))$ on input state $s = (h, \dot{h}_0, \tau, \text{adv}) \in S$, producing an action $a \in Act = [9]$ corresponding to the highest-scoring advisory, where:

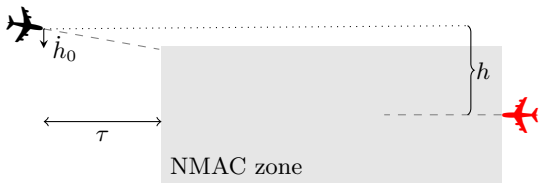


Fig. 3. VerticalCAS encounter geometry

	Comp-Par				Comp-Seq				Monolithic			
k	-19.5	-22.5	-25.5	-28.5	-19.5	-22.5	-25.5	-28.5	-19.5	-22.5	-25.5	-28.5
1	0.629	0.608	0.649	0.652	0.728	0.819	0.737	0.750	0.039	0.039	0.041	0.042
2	2.901	2.730	1.092	1.429	5.392	5.594	0.623	0.618	4.399	6.450	1.444	3.323
3	10.67	1.716	1.918	1.824	26.06	0.986	0.961	0.964	23.33	14.58	12.79	13.59
4	39.58	40.91	2.474	2.570	109.9	108.7	1.404	1.417	–	–	377.5	29.96
5	145.6	156.3	159.8	3.830	433.5	481.2	512.4	2.244	–	–	–	751.1
6	797.4	544.8	573.5	568.8	2174	1639	1826	1859	–	–	–	–

Table 1. Verification times for a VerticalCAS system against the property φ^k for different values of k and $\dot{h}_0$. Greyed-out cells indicate a `False` result, otherwise a `True` result. We use dashes '–' to indicate a two hour timeout.

- $\mathsf{select}\colon S \to F$ selects the neural network corresponding to the previous advisory adv, defined $\mathsf{select}(s) = f_{\text{adv}}$,
- $\mathsf{apply}\colon F \times \mathbb{R}^4 \to \mathbb{R}^9$ computes the output of a neural network given a state, defined as $\mathsf{apply}(f, s) = f(h, \dot{h}_0, \tau)$,
- $\arg\max\colon \mathbb{R}^9 \to [9]$ returns the index of the score with highest value from a neural network's output.

Since each of the above functions and the ReLU-FFNNs are PWL, the composition *prot* is also PWL.

We model the ownship pilot's non-deterministic behaviour in the *environment* of $\mathcal{S}$. Thus, the environment transition function t_E "chooses" an acceleration and determines the next state of the environment through the state transition dynamics. As described in [23], the acceleration chosen by the pilot is assumed to be from a continuous interval, but we bound the number of possible successor states of t_E, by discretising the set of possible accelerations into b equally spaced cells. Here we choose $b = 3$. Take for example advisory DND; the set of next possible accelerations are $\{g/4, 7g/24, g/3\}$.

Assume a boolean predicate $\mathsf{compliant}\colon S \times Act \to \mathbb{B}$ which returns `True` iff the current vertical climbrate of the ownship is compliant with the advisory issued by the agent. Non-zero accelerations are chosen only if $\mathsf{compliant}$ does not hold, otherwise the pilot maintains a constant climbrate, i.e., $\ddot{h}_0^{(i)} = 0$ for $i \in [b]$. Given the current state $s \in S$, the issued advisory $\text{adv}' = \mathit{prot}(s)$, and the set of b accelerations $\{\ddot{h}_0^{(i)} : i \in [b]\}$ corresponding to the advisory adv', we define

each of the transition functions $t_1, \ldots, t_b$ for t_E as:

$$t_i\left(\begin{bmatrix} h \\ \dot{h}_0 \\ \tau \\ \text{adv} \end{bmatrix}, \text{adv}'\right) = \begin{bmatrix} h - \dot{h}_0 \Delta\tau - 0.5\ddot{h}_0^{(i)}\Delta\tau^2 \\ \dot{h}_0 + \ddot{h}_0^{(i)}\Delta\tau \\ \tau - \Delta\tau \\ \text{adv}' \end{bmatrix},$$

where $\Delta\tau = 1$ and $i \in [b]$.

Experimental Results. We tested NANESVERIFY on the following safety specification:

$$\varphi^k = AX^k\left((1) > 100 \vee (1) < -100\right)$$

for various values of k. The formula φ^k is satisfied if from every initial state in I, all possible evolutions of the system remain *safe* after k time steps, i.e., there does not exist a state in I which, after k time steps, can lead to the ownship entering the unsafe region ($|h| \leq 100$), which may potentially lead to an NMAC for small values of τ (recall that in $\text{bCTL}_{\mathbb{R}<}$, the term (1) represents the first component of the state s and so refers to $s.1 = h$). We consider the verification problem with the set of initial states

$$I = [-133, -129] \times \{-19.5, -22.5, -25.5, -28.5\} \times \{25\} \times \{\text{COC}\}.$$

This is a potentially risky encounter with the intruder initially below the ownship, but with the ownship descending towards the intruder.

All results were obtained on a machine with an Intel Core i7-6700 3.40GHz CPU with 16GB of RAM, running a 64-bit version of Ubuntu 16.04. The results for the monolithic procedure are denoted MONOLITHIC, and the results for the compositional procedure with parallel and sequential execution are denoted COMP-PAR and COMP-SEQ, respectively. In Table 1, we report the performance of the tool in terms of the amount of time (in seconds) to resolve the specification φ^k for $k \in \{1, \ldots, 6\}$ with initial climbrates $\dot{h}_0 \in \{-19.5, -22.5, -25.5, -28.5\}$ for each of the execution modes. For all cases we use a fixed timeout of two hours. We empirically observe a linear relationship between k and the size of each MILP program.

In Table 1 we see a climbrate of -28.5 ft/s resulting in a period where the ownship enters the unsafe region for four time steps. For smaller descent rates, the time spent in the unsafe region decreases, until for $\dot{h}_0 = -19.5$ where the ownship remains safe for the entire period. Upon analysing the trace produced by NANESVERIFY for $(\dot{h}_0, k) = (-22.5, 3)$, the agent produces advisory CL1500 at each time step, causing the pilot to accelerate at $g/4$ ft/s^2 in an attempt to climb to avoid colliding with the intruder. The descent rate was not reduced quickly enough to avoid the unsafe state $(h, \dot{h}_0, \tau, \text{adv}) = (-97.719, 1.65, 22, \text{CL1500})$ being reached by the third timestep.

Overall, COMP-PAR is the most performant method for resolving the specification φ^k. We observe that 3^k MILPs are generated for each k when using a compositional encoding; it becomes increasingly necessary to spread the computational load across the available worker processes especially when checking for infeasibility. The speed-up is most noticeable for $\dot{h}_0 = -19.5$.

We expect that Comp-Par is in general more performant when checking for feasibility. For Comp-Seq, we observed that the first MILP checked in the for-loop of Algorithm 2 was feasible, causing the loop to return early, giving quicker feasibility checks compared to Comp-Par. We observe that Monolithic is overall the least performant encoding, with several cases of timeouts when checking for infeasibility of the generated MILPs for $k \geq 4$. Although for VerticalCAS, the unsafe region was entered and eventually escaped, the performance of our compositional procedure exemplifies the tractability of finding shallow bugs in a faulty system.

We are unable to present a comparison with other tools because, as far as we are aware, no other tool supports branching models and CTL specifications as we do here, although the neural network encoding can be interchanged with any state-of-the-art MILP-based approach, e.g. [5]. We use double-precision floating point numbers for representing real values. For the MILP solver that we use for our back-end, Gurobi, we use the default tolerance level of 10^{-6}, which represents the amount of numerical error allowed on a constraint while still considering it "satisfied". We rely on Gurobi for dealing with any further numerical issues. Note also that our encoding is more efficient than [2], which does not use symbolic linear relaxation for their neural network encoding nor interval arithmetic-based bounds propagation for MILP variables.

5 Conclusions

As we argued in Section 1, forthcoming autonomous systems will make greater use of machine learning methods; therefore there is an urgent need to develop techniques aimed at providing guarantees on the resulting behaviour of such systems. While the benefits of formal methods have long been recognised, and they have found large adoption in safety-critical systems as well as in industrial-scale software, there have been few efforts to introduce verification techniques for systems driven by neural networks.

In this paper we defined a system composed of a neural agent driven by deep feed-forward neural networks interacting with a non-deterministic environment. The resulting system displays branching evolutions. We defined and studied the resulting verification problem. While the problem is undecidable for full reachability, we isolated a fragment of the temporal language and showed that its corresponding verification problem is in coNExpTime. We developed and reported on a toolkit which includes a novel parallel algorithm to verify temporal properties of the complex environment defined in the VerticalCAS scenario. As demonstrated, while the parallel algorithm remains complete, it offers considerable advantages over its sequential counterpart when searching for counterexamples to bounded safety specifications in concrete examples.

In future work we plan to extend the framework to multiple agents operating in an environment.

6 Acknowledgements

This work is partly funded by DARPA under the Assured Autonomy programme (FA8750-18-C-0095). Alessio Lomuscio is supported by a Royal Academy of Engineering Chair in Emerging Technologies.

References

1. Akintunde, M.E., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of RNN-based neural agent-environment systems. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI19). pp. 6006–6013. AAAI Press (2019)
2. Akintunde, M.E., Lomuscio, A., Maganti, L., Pirovano, E.: Reachability analysis for neural agent-environment systems. In: Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18). pp. 184–193. AAAI Press (2018)
3. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003)
4. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Autonomous Agents and Multi-Agent Systems **12**(2), 239–256 (2006)
5. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of neural networks via dependency analysis. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI20). AAAI Press (2020)
6. Bunel, R.R., Turkaslan, I., Torr, P., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS18), pp. 4790–4799. Curran Associates, Inc. (2018)
7. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1), 7–34 (2001)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
9. D'Ambrosio, C., Lodi, A., Martello, S.: Piecewise linear approximation of functions of two variables in milp models. Operations Research Letters **38**(1), 39–46 (2010)
10. Doan, T.T., Yao, Y., Alechina, N., Logan, B.: Verifying heterogeneous multi-agent programs. In: Proceedings of the 13th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS14). pp. 149–156 (2014)
11. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC19). pp. 157–168. ACM (2019)
12. Ehlers, R.: 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). Lecture Notes in Computer Science, vol. 10482, pp. 269–286. Springer (2017)
13. Emerson, E.A., Mok, A.K., Sistla, A.P., Srinivasan, J.: Quantitative temporal reasoning. Real-Time Systems **4**(4), 331–352 (1992)
14. Fard, M.M., Pineau, J.: Mdps with non-deterministic policies. In: Proceedings of the 22nd Conference on Neural Information Processing Systems (NIPS09). pp. 1065–1072. Curran Associates, Inc. (2009)

15. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Proceedings of 16th International Conference on Computer Aided Verification (CAV04). Lecture Notes in Computer Science, vol. 3114, pp. 479–483. Springer (2004)
16. Griva, I., Nash, S., Sofer, A.: Linear and nonlinear optimization, vol. 108. Siam (2009)
17. Gu, Z., Rothberg, E., Bixby, R.: Gurobi optimizer reference manual. http://www.gurobi.com (2016)
18. Haykin, S.S.: Neural Networks: A Comprehensive Foundation. Prentice Hall (1999)
19. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: ReachNN: Reachability analysis of neural-network controlled systems. ACM Transactions on Embedded Computing Systems (TECS) **18**(106), 1–22 (2019)
20. Hunt, K., Sbarbaro, D., Zbikowski, R., Gawthrop, P.: Neural networks for control systems–A survey. Automatica **28**(6), 1083–1112 (1992)
21. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC19). pp. 169–178 (2019)
22. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proceedings of the 35th Digital Avionics Systems Conference (DASC16). pp. 1–10 (2016)
23. Julian, K.D., Kochenderfer, M.J.: A reachability method for verifying dynamical systems with deep neural network controllers. CoRR **abs/1903.00520** (2019)
24. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification (CAV17). Lecture Notes in Computer Science, vol. 10426, pp. 97–117. Springer (2017)
25. Kouvaros, P., Lomuscio, A.: Parameterised verification for multi-agent systems. Artificial Intelligence **234**, 152–189 (2016)
26. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of the 26th Conference on Neural Information Processing Systems (NIPS12), pp. 1097–1105. Curran Associates, Inc. (2012)
27. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks. CoRR **abs/1706.07351** (2017)
28. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. Software Tools for Technology Transfer **19**(1), 9–30 (2017)
29. Maes, P.: Modeling adaptive autonomous agents. Artificial life **1**(1–2), 135–162 (1993)
30. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML10). pp. 807–814. Omnipress (2010)
31. NANESVerify: Neural Agent operating on a Non-deterministic Environment System Verify, https://vas.doc.ic.ac.uk/software/neural/ (2020)
32. Narodytska, N.: Formal analysis of deep binarized neural networks. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, (IJCAI18). pp. 5692–5696 (2018)
33. Penczek, W., Lomuscio, A.: Verifying epistemic properties of multi-agent systems via bounded model checking. In: Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-agent systems (AAMAS03). pp. 209–216. IFAAMAS (2003)

34. Penczek, W., Woźna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of CTL. Fundamenta Informaticae **51**(1-2), 135–156 (2002)
35. Redmon, J., Divvala, S.K., Girshick, R.B., Farhadi, A.: You only look once: Unified, real-time object detection. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR16) pp. 779–788 (2016)
36. Sutton, R.S., Barto, A.G.: Reinforcement Learning – An Introduction. MIT Press (1998)
37. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. In: Proceedings of the 2nd International Conference on Learning Representations (ICLR14) (2014)
38. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: Proceedings of the 7th International Conference on Learning Representations (ICLR19) (2019)
39. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Proceedings of the 32nd Conference on Neural Information Processing Systems (NIPS18). pp. 6367–6377. Curran Associates, Inc. (2018)
40. Winston, W.: Operations research: applications and algorithms. Duxbury Press (1987)
41. Xiang, W., Tran, H., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In: 2018 Annual American Control Conference (ACC). pp. 1574–1579. AACC (2018)

A Comparison of Neural Network Tools for the Verification of Linear Specifications of ReLU Networks

Ziggy Attala, Ana Cavalcanti, and Jim Woodcock

University of York, UK

Abstract. There is an increasing interest in using deep neural networks (DNNs) in robotic controllers and safety-critical systems for which verification is paramount. We have selected six modern neural network tools that can verify linear specifications: NNV, ERAN, NeuralVerification.jl, Marabou, SHERLOCK, and Reluplex. We evaluate these tools using two benchmark networks: Network 1 of the ACAS Xu dataset, and a custom example that shares key features with an implementation of a real robot for use on a factory floor. Our primary insight is that, for the purposes of verifying linear specifications of ReLU networks, Marabou is the most efficient and usable tool. On the other hand, Marabou's functionality is limited to proving or disproving properties, while other tools can compute unsafe input and output ranges for the properties. While there have been detailed comparisons of the majority of the algorithms these tools implement, this is, as far as we know, the first comparison of the tools. For that, we use three benchmark problems.

Keywords: Formal Verification · NNV · ERAN · NeuralVerification.jl · Reluplex · Marabou · SHERLOCK

1 Introduction

Deep neural networks are gaining popularity for use in multiple sectors [23], including automotive [23] and robotics [15]. There is, however, no widely accepted tool for verifying a general neural network; this paper delivers insights into emerging tools. These insights inform verifiers on which tools to use in various situations and provide guidance for the improvement and development of tools, and on the utility and applicability of the algorithms and techniques they implement. In addition, our work suggests how the tools can be used in combination.

We have selected four criteria to compare the tools: usability, scalability, precision and applicability. Usability is concerned with the facilities for the definition of a property to verify and of the neural network. Scalability explores how the tools deal with larger and more complex networks. Precision refers to whether the results are precise enough to provide proof of the property. Finally, applicability refers to the type of networks that can be verified.

We explore neural networks for control systems as these are the most relevant in robotics. Examples include a robotic arm network [31] and an airborne collision avoidance system [16]. We are interested in verifying properties such as the robotic arm does not reach an unsafe zone [31] or, if the intruder aircraft is sufficiently far away, the network advises 'clear of conflict' [16].

We focus out comparison on linear properties. They can capture a wide range of specifications, including control safety, adversarial robustness, and robotic controller properties. Furthermore, the vast majority of techniques are concerned with linear properties [13, 16, 18, 25–29]. So, we focus our comparison on linear properties.

In addition, our comparison is based on networks with ReLU activation functions. They are widely used, powerful, and easily trained [22], and every verification method and tool, as far as we know, is applicable to ReLU networks. Its piecewise linear nature is the basis for the feasibility of most techniques.

We are aware of six other comparison works; but, as far as we know, we present the first comparison of tools. There is information available about: several algorithms in [19], but experimental results are not presented; two SMT-like techniques in [10]; multiple convex relaxation methods in [24]; and interval bound propagation methods in [14]. A survey benchmarking on varying problem sets is in [30]. The work most similar to ours is [20]. The volume of experimental results is comparable to that here, including property 10 on the *ACAS* network. They are, however, focused on explaining the methods in a pedagogical manner.

The tools we have selected are those that contain functionality to verify generalised linear specifications: *Matlab Toolbox for Neural Network Verification* (NNV) [1], *ETH Robustness Analyzer for Neural Networks* (ERAN) [2], *NeuralVerification.jl* [3],*Reluplex* [4], *Marabou* [5], and *SHERLOCK* [6]. We evaluate them on three benchmark properties of two networks: two properties of a small neural network (TN) based on the implementation of a real robot, and property one of Network 1, 1 of the ACAS Xu neural networks.

TN is is a feed-forward and deep neural network with two hidden layers of 32 nodes each, with two input and one output node. This network shares key features with control networks: low input and output dimensionality, and feed-forward ReLU structure. It also has a low number of hidden layers and class invariant input zones, which can be represented as linear properties.

Network 1, 1 of ACAS Xu refers to one of the networks of the Airborne Collision Avoidance Systems for Unmanned Aircraft (ACAS Xu) first presented by Katz

[1] `//github.com/verivital/nnv`
[2] `//github.com/eth-sri/eran`
[3] `//github.com/sisl/NeuralVerification.jl`
[4] `//github.com/guykatzz/ReluplexCav2017`
[5] `//github.com/NeuralNetworkVerification/Marabou`
[6] `//github.com/souradeep-111/sherlock`

et al. [16] in 2017. The ACAS Xu networks are a set of 45 neural networks critical for ensuring unmanned aircraft avoid aerial collisions. It is desirable to establish properties to guarantee the behaviour of these networks under certain input domains, so ten properties have been identified.

In Section 2 we provide an introduction to each of the tools evaluated in this work. In Section 3 we present our experiments in relation to the four evaluation criteria set out earlier. In Section 4 we discuss these results, and Section 5 concludes and provides suggestions for future directions.

2 Preliminaries

In this section we provide an introduction to all the tools evaluated.

NNV, ERAN, and SHERLOCK are output range analysers [1, 3, 9]. Given an input range, they compute an output range, and if it falls within the safe output zone of the property the network is determined safe. This computation can either be complete or incomplete, giving the output range or an over-estimation of the output range. NNV and ERAN use sets to compute the output range.

NNV utilises *polytopes*, *hyper-rectangles*, *halfspaces*, and *star sets*. A polytope is a generalisation in higher dimensions of a three-dimensional polyhedron. And a hyper-rectangle is one of the simplest classes of polytopes [21]. This is defined as all points between two upper and lower vectors: the rightmost and leftmost corners of the rectangle given as $\underline{x} = (\underline{x}^1, ..., \underline{x}^n)$ and $\bar{x} = (\bar{x}^1, ..., \bar{x}^n)$ [21]. A halfspace is the set of points that satisfy: $a \cdot x \leq b$ where a is an n-dimensional vector [21]. Finally, a star set is an efficient representation of high-dimensional polytopes. This forms the basis for the methods implemented in NNV.

Any bounded convex polytope can be represented as a star set. This is a tuple $\langle c, V, P \rangle$ where $c \in \mathbb{R}$ is the centre, $V = \{v_1, v_2, ..., v_m\}$ is a set of m vectors in $\mathbb{R}^n$ called basis vectors, and $P : \mathbb{R}^m \rightarrow \{\top, \bot\}$ is a predicate. The set of states represented by the star is: $\{x | x = c + \Sigma_{i=1}^{m}(a_i v_i)$ and $P(a_1, ..., a_m) = \top\}$

NNV has implemented multiple methods for output range analysis: complete output range verification in *exact-star* and *exact-poly*, incomplete output range analysis in *approx-zono*, *approx-star*, *abs-dom* (which uses polytopes) and *approx-hr* (which uses hyper rectangles). We consider them all here.

ERAN has three modes of operation. L_{inf} and geometric analysis are applicable to image domain specifications, so are not considered here. Linear specifications are defined using zonotopes, a centrally symmetric polytope [21]. ERAN defines zonotopes using affine arithmetic, an extension of interval arithmetic. For analysing zonotopes, ERAN has two methods: a hybrid analysis *RefineZono*, and an incomplete output range analysis *DeepZono*. We consider both.

A variable $x \in \mathbb{R}$ can be defined in interval arithmetic through $[a, b]$, where a and b are floating point numbers and $a \leq x \leq b$ [12]. Affine arithmetic extends interval arithmetic by associating an affine form $\hat{x} = x_0 + x_1\epsilon_1 + ... + x_n\epsilon_n$ with

each quantity, where x_0 is the central value and $x_1, ..x_n$ are known as the partial deviations, associated with the noise symbols ϵ [12]. A zonotope defined in affine form associates an affine expression $\hat{x}$ with each of its dimensions. As the affine expressions are related, their error terms are shared.

SHERLOCK utilises conjunctions of linear inequalities as input [9]. SHERLOCK combines gradient-based local search with MILP solving: it solves a series of MILP feasibility problems with local search steps. SHERLOCK, however, is only applicable to networks with a single output node [11]. The implementation we evaluate is the implementation `https://github.com/souradeep-111/sherlock`, as it has proved impossible to build the newer `https://github.com/souradeep-111/sherlock_2`, even with support from the authors.

All the above tools do not require an output set to be defined as they are output range analysers. Defining an unsafe zone after computation can be useful for the analysis of the generated output range sets, but is not necessary.

Reluplex and Marabou are SMT solvers for neural networks [2, 8, 16, 17]. The implementation of Reluplex we have evaluated was the proof of concept implementation in [8]. They find an activation, a single value within the bounds of the problem given. As they compute a single point, the property is inverted, and it is proved if no activation that satisfies the inverse property is found. The input and output constraints are defined using multiple linear inequalities.

NeuralVerification.jl implements 17 different verification methods, both SMT solvers and output range analysers. Inputs are polytopes and hyper-rectangles, and as output it uses halfspaces and polytope complements [4]. Polytope complements define outputs for SMT solvers, and halfspaces are used for the output specification for output-range analysers. However, we have been able to run only five of these methods: SHERLOCK (the algorithm and tool are referred to by the same name), BaB (an SMT solver), Duality, ExactReach, and Ai2. Of these only BaB and SHERLOCK ran on both properties of TN.

In the next section, we address our evaluation criteria.

3 Evaluation Experiments

We evaluate the tools based on four criteria: usability (Section 3.1), scalability (Section 3.2), precision (Section 3.3) and applicability (Section 3.4).

3.1 Usability

Specification of Properties In the following, we describe how an I/O property of a neural network can be defined in each tool.

NNV I/O properties in NNV are defined by the input-set object used by their reach methods. Defining an unsafe output set, however, is a simpler way of evaluating the generated output reachable set. Each set is defined as a custom

MATLAB object. The simplest way to build these objects is to build a hyper-rectangle based on a lower and upper bound vector, and then convert this using NNV's built in methods. If an output set is defined, NNV is also able to generate an intersection with the generated reachable set with the safe output set.

ERAN The input property is defined as a zonotope in affine form. This allows any convex polytope to be defined through shared error terms, but for linear properties we define a simple hyper-rectangle input in affine interval form.

NeuralVerification.jl Properties are defined through the creation of input and output set objects, in a similar manner to NNV. NeuralVerification.jl also uses an output set with reachability methods.

There are four implemented types of set used as I/O definitions, these are: hyper-rectangles, halfspaces, hpolytope (halfspace polytopes) and polytope complements. Hyper-rectangles, halfspaces and hpolytopes are the most commonly used; polytope complements are used for the output sets for solvers. This is because the output set needs to be the negation of the safe zone, and a PolytopeComplement of a closed set is necessarily an open set, and non-convex.

Reluplex There is no distinct method to define properties; the code that defines the method has to be modified to deal with the desired input.

Marabou This implementation requires properties to be defined using a set of inequalities, relating to the input and output nodes. The input nodes are defined as x_n, and the output nodes as y_n, where n is the index of the node.

SHERLOCK The input is defined through input constraints as a hyper-rectangle. There is no way to define output zones, and properties are verified through analysing the computed output range.

Neural Network File Formats The file formats used to define neural networks varies, but the key information which all file formats support is: the weight matrix and the bias vector for each layer. We primarily focus on human readable neural network file formats, since only ERAN is able to natively support non-text based file formats such as ONNX [7], Tensorflow .pb and .meta files [1]. NNV is also able to support these, but through an extension NNVMT [6].

NNet Format The nnet file format was created in 2016 to define the ACAS networks in a human-readable format [5]. It contains normalization information and the structure of the network, including input and output dimensions. The primary drawback of this format is that it is only applicable to *feed forward fully connected ReLU networks* [5]. The nnet file format is utilised by Reluplex, Marabou and NeuralVerification.jl.

ERAN Text Data Formats ERAN's text-based formats are TensorFlows .tf and PyTorch's .pyt. Unlike nnet files, they can define wider types of activation function, including *tanh*, *Sigmoid*, and *Affine* functions. They can also define convolutional networks. Neither file format, however, is readable because the input and output dimension of the network has to be inferred from the weight matrices themselves. Also these file formats do not contain a definition of the structure of the hidden layers, which has to be inferred from the matrices. Only .pyt files contain inherent normalization information.

SHERLOCK Format SHERLOCK utilises a custom file format that defines solely a feed-forward ReLU network. The format defines a network neuron by neuron for small networks, this can provide readable information about the internal structure of the network, because it flattens the weight matrix for each layer and delimits the weights via new line characters. For larger networks, however, this is not a readable file format.

NNV NNV builds networks layer by layer, given the activation function, weight matrix and bias vector for each layer. This information can be loaded through any file format supported by MATLAB, for example, mat, csv and txt files. There is, however, no standard neural network file format support without the use of NNVMT.

A property can be defined by inequalities [2,8,9], or convex sets [1,3,4]. A neural network is defined through associating a weight matrix, a bias vector, and an activation function for each layer, which can be defined node by node, [9], layer by layer [3], or through nnet, pyt or tf files [1, 2, 8, 20].

3.2 Scalability

We have installed and built the tools on Linux version: 'Ubuntu 18.04.2 LTS', apart from NNV which was installed on MATLAB 2019a, Windows version[7]. We have used a i5-8265U CPU with 8GB RAM, with the timeout for each experiment set to 2 hours. We have repeated each experiment that terminated 10 times.

For the NeuralVerification.jl implementations, we have obtained impossible results using Reluplex. In addition, we have experienced problems executing ExactReach and Ai2 on both benchmark networks.

RefineZono is a hybrid method; results are for RefineZono with its complete part at one second, the default value. We have also ran RefineZono with a 1000 second timeout, however, it was still unable to obtain tight enough bounds to verify property one of ACAS Xu. Finally, we have ran RefineZono using its full complete mode, although this timed out. The result of RefineZono with milp timeout 1000 is discussed in the precision section.

[7] `www.mathworks.com/products/matlab.html`

Table 1. Scalability Results (Seconds, 2dp)

	TN P1:					TN P2:				$AX11_1$:			
Tool:	Method:	Mean	Min	Max	RES	Mean	Min	Max	RES	Mean	Min	Max	RES
NNV	Exact-Star	95.18	77.33	122.57	SAT	73.4	55.72	93.57	SAT	T/O	T/O	T/O	T/O
	Exact-Poly	*	*	*	ERR	*	*	*	ERR	T/O	T/O	T/O	T/O
	Approx-Star	4.83	3.94	6.81	AMB	4.67	4.67	5.78	AMB	42.4	30.83	55.38	AMB
	Zono	0	0	0.01	AMB	0.01	0	0.01	AMB	0.03	0.02	0.04	AMB
	Abs-dom	4.94	3.88	6.81	AMB	4.71	3.7	5.81	AMB	40.86	19.55	49.09	AMB
	Approx-Hr	0.01	0	0.05	AMB	0.01	0	0.02	AMB	0.01	0.01	0.02	AMB
ERAN	DeepZono	0.06	0.06	0.07	AMB	0.06	0.06	0.06	AMB	0.15	0.09	0.16	AMB
	RefineZono	3.42	3.24	3.6	SAT	3.85	3.67	4.04	SAT	38.42	37.21	39.47	AMB*
SHERLOCK	SHERLOCK	*	*	*	ERR	0.9	0.72	1.08	SAT	N/A	N/A	N/A	N/A
NeuralVerification.jl	SHERLOCK	28.64	28.22	29.19	SAT	43.81	41.68	46.12	SAT	N/A	N/A	N/A	N/A
	BaB	T/O	T/O	T/O	T/O	T/O	T/O	T/O	T/O	N/A	N/A	N/A	N/A
Marabou	Marabou	1.21	0.75	1.52	SAT	0.44	0.43	0.48	SAT	130.84	130.22	134.22	SAT
Reluplex	Reluplex	*	*	*	ERR	*	*	*	ERR	1348.9	1119	2620	SAT

Table 1 displays the results of the experiments on the three benchmarks. The 'RES' column displays the result of the method: 'SAT', the property is satisfied; 'AMB', the result of the method was ambiguous, that is, the bounds were not tight enough to prove the property; 'ERR', the method did not execute on the property and the time is displayed as *; 'T/O' represents a time out result.

From Table 1, it can be seen that Marabou is the only tool able to verify all three properties, NNV and ERAN are able to verify TN, but only using complete methods, and SHERLOCK is able to verify P2, but produces errors on TN P1.

3.3 Precision

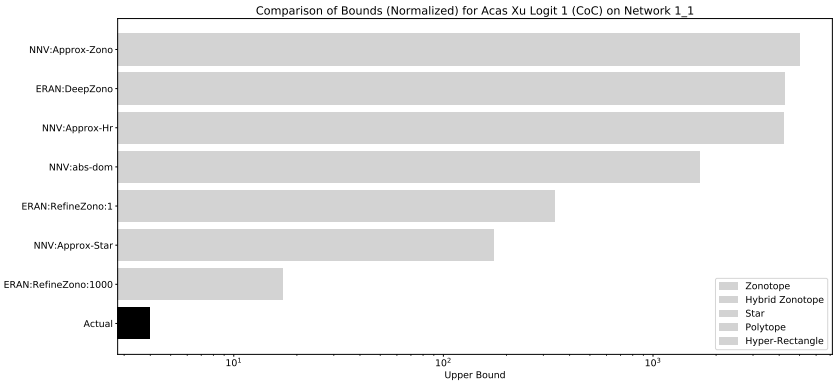


Fig. 1. ACAS Xu P1 Bounds Comparison

In this section we evaluate the quality of the output bounds produced by the incomplete methods implemented by the tools. These bounds may not be tight enough to prove the property in question.

Figures 1 and 2 show the positive bounds generated by the methods of the tools, using logarithmic scaling. Some techniques generate negative bounds, but we record only the positive bounds, as all networks have ReLU output layers. In addition, the bounds given by complete methods are displayed for reference.

One of the main observations we can make from this is that the type of convex set used by a method significantly impacts the bounds generated. Ordering them from least to most precise we have: hyper-rectangles, zonotopes, polytopes then star sets. One exception is that hyper-rectangles are slightly more precise than zonotope methods on ACAS Xu. RefineZono also involves an additional MILP component as well as a zonotope abstract domain formulation.

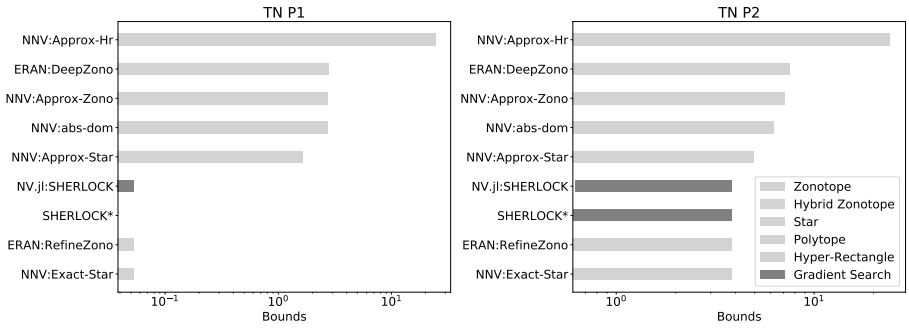


Fig. 2. TN P1 and P2 Bounds Comparison

3.4 Applicability

Marabou and Reluplex are applicable solely to feed-foward fully connected ReLU networks, as the modified Simplex algorithm they are based on does not allow for other types of non-linearity. SHERLOCK is also solely applicable to ReLU networks, and contains the additional limitation of only one output node.

ERAN implements abstract domain methods, and so is the most widely applicable tool. It is applicable to ReLU, Sigmoid, and Tanh activation functions; it is also applicable to feedforward, convolutional, and residual layers.

NNV's methods vary as to their applicability. The exact methods are only applicable to ReLU or linear activation functions, while their approximate reachabil-

ity modes are, in addition, applicable to sigmoid and tanh activation functions. Both types of method are also applicable to convolutional layer types.

Of the methods we considered of NeuralVerification.jl, those applicable to only ReLU networks are: BaB, Reluplex, ExactReach, ConvDual and SHERLOCK. BaB is also only applicable to one output. The implementation of Ai2 is applicable only to ReLU networks, however, Ai2 is applicable to further types of networks [13]. Duality is applicable to any monotone activation function.

4 Evaluation

Marabou is the most successful tool for the verification of ReLU networks, overall, in accordance with our criteria. It is the most usable, scalable and precise, but is one of the most limited tools in terms of applicability.

The most usable method for defining properties is Marabou's. It is the only tool with a dedicated file format for the specification of a complete property. ERAN has a file for specifying a zonotope in affine form, but this is only the input to the network not a complete property definition.

Further conversion tools would be useful for usability, such as, a converter from linear inequality to set representation, converters between various representations of sets, and a converter between generator-form zonotopes and affine-form zonotopes, as used by ERAN and NNV, respectively. These facilities could facilitate obtaining complementary results from multiple tools.

In terms of defining networks, nnet files are the most usable. This is for two main reasons: they display the network structure efficiently, and they contain normalization information. In some situations, however, viewing the exact weightings for each node with a node-by-node file structure may be useful. This can give further information into the weighting assigned to each feature of the data, although this is only feasible with small, simple networks.

The most scalable tool for complete verification is Marabou, obtaining a runtime, on TN, 2.8x faster than ERAN, the next fastest complete tool. It has a runtime 78x faster than NNV's complete analysis on TN, and NNV timed out in the verification of ACAS Xu. The only tools for complete verification that terminated for ACAS Xu Property 1 was Reluplex and Marabou.

The runtime of these tools was influenced by the bounds of the property chosen about the network. This is very minor in most tools, but Marabou experienced a difference of 2.71x from TN property 1 to property 2. For comparison, the largest variation other than Marabou was SHERLOCK with a variation of 1.52x.

The most precise tool for incomplete analysis is ERAN (utilising RefineZono); this, however, is also the least scalable incomplete method. Only Marabou, ERAN and SHERLOCK are precise enough to verify TN properties and only Marabou and Reluplex are precise enough to verify the Acas Xu property.

In terms of verifying ReLU neural networks, all tools are applicable to all networks, but BaB, from NeuralVerification.jl and SHERLOCK are unable to verify networks with more than one output node. This is a limitation from the algorithms defining their methods. ERAN and NNV are the only tools applicable to types of neural network beyond feed-forward fully connected networks.

NNV, while not quite the most scalable, precise or applicable, has the highest degree of functionality of all tools presented. Not only is it an output range analyser, it has methods to generate the unsafe input zones of a problem, functionality no other method is able to perform. One advantage of this is that the unsafe input zone can be used as an adversarial input generator for robust training [28]. Furthermore, NNV contains output visualisation and intersection methods, functionality also unique to NNV. This enables a clear representation of the decision logic of a neural network.

5 Conclusion

Based on the examples we have considered, in terms of usability, the best tool is Marabou, and the worst is SHERLOCK. For scalability, the best is NNV and the worst is Reluplex. For precision, the best is Marabou and the worst is NNV. Lastly, for applicability, the best is ERAN and the worst is SHERLOCK.

We have compared multiple state-of-the-art tools for verifying ReLU neural networks. We have compared file formats for the definition of properties and the definition of neural networks. We have also given insights on how the property, network size, and type of input set affects the operation of these tools. This can, not only help operation of these tools, but help to make an informed decision on the type of tool appropriate to a network and problem.

Our works suggests: first, possible ways of combining tools so that their results complement each other, and increase the explainability of the network. Second, tool combination and integration is an interesting avenue for future work. Finally, that a unified format of representing linear inequalities is useful in the development of verification tools.

This work can help inform the decision process in selecting tools to verify neural networks, as well as how to use them in combination. Further discussion of the usability of these tools is needed and can inform further development of tools in this area. Further exploration into the utility of these tools and methods is of utmost importance to the emerging area of neural network verification.

Acknowledgements

This work is funded by the Royal Academy of Engineering grant CiET1718/45, and UK EPSRC grants EP/M025756/1 and EP/R025479/1. No new primary data was created as part of the study reported here.

References

1. Eth robustness analyzer for neural networks (eran), //github.com/eth-sri/eran, accessed 14/04/20
2. Marabou, //github.com/NeuralNetworkVerification/Marabou, accessed 15/04/20
3. Matlab toolbox for neural network verification (nnv), //github.com/verivital/nnv, accessed 13/04/20
4. Neuralverification.jl, //github.com/sisl/NeuralVerification.jl, accessed 14/04/20
5. Nnet repository, https://github.com/sisl/NNet, accessed 14/04/20
6. Nnvmt: A translation tool for feedforward neural network models, //github.com/verivital/nnvmt, accessed 20/04/20
7. Onnx, //github.com/onnx/onnx, accessed 14/04/20
8. Reluplex cav 2017, //github.com/guykatzz/ReluplexCav2017, accessed 14/04/20
9. Sherlock, //github.com/souradeep-111/sherlock, accessed 14/04/20
10. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Kumar, M.P.: A Unified View of Piecewise Linear Neural Network Verification. In: NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems. pp. 4795–4804 (December 2018)
11. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks (2017), arXiv:1709.09130
12. de Figueiredo, L.H., Stolf, J.: Affine arithmetic: Concepts and applications. Numerical Algorithms **37**, 147–158 (2003), doi.org/10.1023/B:NUMA.0000049462.70970.b6
13. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 3–18 (2018)
14. Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelovic, R., Mann, T., Kohli, P.: On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models (2018), arXiv:1810.12715
15. Horne, W., Jamshidi, M., Vadiee, N.: Neural networks in robotics: A survey. Journal of Intelligent and Robotic Systems **3**, 51–66 (03 1990). https://doi.org/10.1007/BF00368972
16. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. Lecture Notes in Computer Science p. 97–117 (2017)
17. Katz, G., Huang, D., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D., Kochenderfer, M., Barrett, C.: The Marabou Framework for Verification and Analysis of Deep Neural Networks. In: Computer Aided Verification, CAV 2019. Lecture Notes in Computer Science, vol. 11561. Springer, Cham (07 2019)
18. Krishnamurthy, Dvijotham, Stanforth, R., Gowal, S., Mann, T., Kohli, P.: A Dual Approach to Scalable Verification of Deep Networks (2018), arXiv:1803.06578
19. Leofante, F., Narodytska, N., Pulina, L., Tacchella, A.: Automated Verification of Neural Networks: Advances, Challenges and Perspectives (2018), arXiv:1805.09938
20. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for Verifying Deep Neural Networks (2019), arXiv:1903.06758

21. Maler, O.: Computing reachable sets: An introduction (2008), //semanticscholar.org/paper/Computing-Reachable-Sets-%3A-An-Introduction-Maler/299949aef669b547a36c091b768cade091d35532, accessed 30/04/20
22. Nwankpa, C., Ijomah, W., Gachagan, A., Marshall, S.: Activation Functions: Comparison of trends in Practice and Research for Deep Learning (2018), arXiv:1811.03378
23. Salay, R., Czarnecki, K.: Using Machine Learning Safely in Automotive Software: An Assessment and Adaption of Software Process Requirements in ISO 26262 (2018), arXiv:1808.01614
24. Salman, H., Yang, G., Zhang, H., Hsieh, C.J., Zhang, P.: A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks. In: NeurIPS 2019 (2019)
25. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and Effective Robustness Certification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 10802–10813. Curran Associates, Inc. (2018), http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification.pdf
26. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**, 1–30 (01 2019). https://doi.org/10.1145/3290354
27. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (ICLR) (2019)
28. Tran, H.D., Manzanas Lopez, D., Musau, P., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Star-Based Reachability Analysis of Deep Neural Networks. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods – The Next 30 Years. pp. 670–686. Springer International Publishing, Cham (2019)
29. Tran, H.D., Musau, P., Lopez, D.M., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Parallelizable Reachability Analysis Algorithms for Feed-Forward Neural Networks. In: Proceedings of the 7th International Workshop on Formal Methods in Software Engineering. p. 31–40. FormaliSE '19, IEEE Press (2019). https://doi.org/10.1109/FormaliSE.2019.00012, https://doi.org/10.1109/FormaliSE.2019.00012
30. Xiang, W., Musau, P., Wild, A.A., Lopez, D.M., Hamilton, N., Yang, X., Rosenfeld, J., Johnson, T.T.: Verification for Machine Learning, Autonomy, and Neural Networks Survey (2018), arXiv:1810.01989
31. Xiang, W., Tran, D., Johnson, T.: Output reachable set estimation and verification for multi-layer neural networks. IEEE Transactions on Neural Networks and Learning Systems **PP** (08 2017). https://doi.org/10.1109/TNNLS.2018.2808470

Efficient Verification of ReLU-based Neural Networks via Dependency Analysis

Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener

Department of Computing, Imperial College London, UK
{e.botoeva,p.kouvaros,j.kronqvist,a.lomuscio,r.misener}@imperial.ac.uk

Abstract. We introduce an efficient method for the verification of ReLU-based feed-forward neural networks. We derive an automated procedure that exploits dependency relations between the ReLU nodes, thereby pruning the search tree that needs to be considered by MILP-based formulations of the verification problem. We augment the resulting algorithm with methods for input domain splitting and symbolic interval propagation. We present Venus, the resulting verification toolkit, and evaluate it on the ACAS collision avoidance networks and models trained on the MNIST and CIFAR-10 datasets. The experimental results obtained indicate considerable gains over the present state-of-the-art tools.

Keywords: Feedforward ReLU networks · Mixed Integer Linear Programming · Dependency analysis.

1 Introduction

Artificial Intelligence (AI) methods are increasingly used in safety critical applications including, but not limited to, autonomous vehicles, avionics, and power generation. These domains typically require a certification aimed at establishing the safety of the application to be deployed.

Formal verification methods commonly used in software verification cannot address the validation of AI applications due to the inherently different components. In particular, AI applications increasingly utilise neural networks in key parts of their designs, most notably in perception and control modules. Due to this, the area of formal verification of neural networks has recently received considerable attention. Simply put, methods for assessing neural systems can provide the mathematical underpinning for safely deploying a wide number of AI applications.

The typical decision problem tackled by verification approaches is whether a neural network, or a closed-loop system in which neural networks are present, can output particular values, i.e., output reachability. Reachability is often studied in conjunction with local robustness properties, i.e., whether for a given input, e.g., an image, small alterations of this input can cause output variation, e.g., a different classification. The present state of the art [18] includes several ways of formulating this problem (see related work below); however, no method scales

to the analysis of the neural networks presently used in industrial strength applications, including autonomous vehicles. Therefore, it remains of considerable importance to develop more scalable approaches. This is the aim of the present contribution.

This paper introduces a novel, MILP-based approach to verifying feed-forward ReLU-based neural networks. Rectified Linear Units (ReLUs) are one of the most commonly-used activation functions in vision and are the typical object of study in the above cited literature. This manuscript develops the concept of *dependency*. Two nodes in a neural network are in a dependency relation if there is a strict connection between their active or inactive state during the overall network computation. As we show, dependency can be exploited to improve the performance of a MILP formulation. Crucially, and differently from the related state-of-the-art, our dependency analysis is not aimed at reducing the number of variables in the verification problem, but rather at reducing the search space during a branch-and-bound approach to generate satisfiable assignments. This paper (i) develops effective methods exploiting these dependencies and (ii) integrates the methods into a larger implementation incorporating scalability-improving methods such as domain splitting. The resulting implementation generates speed-ups of at least one order of magnitude against competing methods.

The rest of the paper is organised as follows. After discussing related work below, Section 2 reports key concepts on neural networks and related verification approaches. Section 3 first presents the theoretical contribution on dependency analysis and then gives a dependencies-based verification algorithm. Section 4 presents a toolkit exploiting dependency analysis. Section 5 reports the experimental results obtained on the MNIST, CIFAR-10, and ACAS datasets and compares these against state-of-the-art implementations.

Related Work. Verification methods for neural networks can be partitioned into complete and incomplete ones. Complete methods can in principle return a definite answer as to whether the property in question is satisfied. Differently, incomplete methods may erroneously conclude that the network is not robust when it actually is or a certain output is reachable when it is actually not.

Incomplete methods include approaches based on duality [6], abstract interpretation [10], symbolic interval analysis [27, 25] and semidefinite relaxations [20, 8]. While the techniques differ, they all overestimate the output of the network from a given input region in an attempt to draw a conclusion from this. While incomplete methods may be very efficient in some cases, they are not comparable to the ones here presented as they may not answer the verification problem due to false negatives.

Complete approaches can be divided into 3 main groups: (i) MILP-based [3, 19, 5, 9, 4, 23] techniques that formulate the verification problem at hand as a mixed integer linear program; (ii) SMT-based [7, 13, 14] techniques that encode the verification problem as the satisfiability modulo theory problem; (iii) techniques that use a combination of overestimation and refinement techniques to get a definite answer [26, 25].

Closely related to this paper is some of the recent work, which has focused on conquering scalability and increasing precision in incomplete approaches. These include: (i) computing tight bounds using symbolic interval analysis [25, 27]; (ii) input splitting [26, 14, 21]; (iii) optimised MILP formulations [4, 23, 2]. The work here presented uses MILP formulations to the verification problem combined with input splitting and symbolic interval analysis methods. However, differently from the work cited above, it uses novel heuristics based on dependency analysis to guide the search for feasible solutions.

2 Background

This section summarises and fixes the notation on some of the key notions used later in the paper.

Feed-forward neural networks. A feed-forward neural network (FFNN) is a directed acyclic graph whose nodes are structured in layers. The first layer is the *input layer*, also referred to as layer 0, the last layer is the *output layer*, also referred to as layer k, and every layer in-between is a *hidden layer*, also referred to as layer i, for $1 \leq i < k$. Every node other than an input node is connected to every node in the preceding layer. Each edge is associated with a weight, which is learned during the training phase. Given an input vector, the network computes a function by propagating the input through the network, where, at each step, a node's *output* results from applying an activation function to the *pre-activation* of the node, which is the weighted sum of the outputs of the nodes from the previous layer. Here, we only consider the *ReLU* activation function defined by $\mathsf{ReLU}(x) \triangleq \max(x, 0)$ for $x \in \mathbb{R}$.

We denote by s_i the number of nodes in layer i. We use $n_{i,q}$ to refer to the q-th node of layer i. Given an input $\mathbf{x}$ to the network, we write $\hat{\mathbf{x}}_{i,q}$ ($\mathbf{x}_{i,q}$, respectively) for the pre-activation (output, respectively) of the node $n_{i,q}$. For a set of inputs over a bounded domain, every node is associated with lower and upper pre-activation and activation bounds. These can be derived in a number of ways discussed below. We write $\hat{\mathbf{l}}_{i,q}$ and $\hat{\mathbf{u}}_{i,q}$ ($\mathbf{l}_{i,q}$ and $\mathbf{u}_{i,q}$, respectively) for the pre-activation's (output's, respectively) lower and upper bounds. Similarly, $\hat{\mathbf{x}}_i$ and $\mathbf{x}_i$ refer to the vector of pre-activations and outputs of layer i over domains $[\hat{\mathbf{l}}_i, \hat{\mathbf{u}}_i]$ and $[\mathbf{l}_i, \mathbf{u}_i]$, respectively, where $\mathbf{x}_0 = \mathbf{x}$, and $\mathbf{l}_0$ and $\mathbf{u}_0$ are the input lower and upper bounds. We write W_i and b_i to refer to the weight matrix and bias vector of layer i, $i \geq 1$, respectively.

Given an input $\mathbf{x}$ and for $i \geq 1$, the output $\mathbf{x}_i$ of layer i is computed from $\mathbf{x}_{i-1}$ by applying the function $f_i \colon \mathbb{R}^{s_{i-1}} \to \mathbb{R}^{s_i}$, which is defined as $f_i(\mathbf{x}_{i-1}) \triangleq \mathsf{ReLU}(W_i\mathbf{x}_{i-1} + b_i) = \mathsf{ReLU}(\hat{\mathbf{x}}_i)$, where the ReLU function is applied element-wise. Given the above, a neural network of $k+1$ layers, is defined as a function $f \colon \mathbb{R}^{s_0} \to \mathbb{R}^{s_k}$, corresponding to the composition of the functions f_i computed by each layer i, i.e., $f(\mathbf{x}) \triangleq f_k(\ldots f_1(\mathbf{x})\ldots)$.

A ReLU node $n_{i,q}$ can be in one of two states. It is in the *strictly active* state (or, is strictly active), denoted $\mathsf{st}(n_{i,q}) = \top$ if $\hat{\mathbf{l}}_{i,q} \geq 0$. It is in the *strictly inactive* state (or, is strictly inactive), denoted $\mathsf{st}(n_{i,q}) = \bot$, if $\hat{\mathbf{u}}_{i,q} \leq 0$. A *stable*

node is a node that is either strictly active or strictly inactive. Otherwise, the node is said to be *unstable*, denoted $\mathsf{st}(n_{i,q}) =?$.

Verification problem. Given a network $f\colon \mathbb{R}^{s_0} \to \mathbb{R}^{s_k}$ and a specification $(\mathcal{X}_0, \mathcal{X}_k) \subseteq \mathbb{R}^{s_0} \times \mathbb{R}^{s_k}$, the verification problem determines whether $\forall \mathbf{x}_0 \in \mathcal{X}_0\colon \mathbf{x}_k \in \mathcal{X}_k$.

To enable the MILP representation, we hereafter assume that $\mathcal{X}_0$ is an intersection of finite sets of polyhedra. The *local robustness* and *reachability* problems are instantiations of the verification problem. The local robustness problem establishes if the network's output is unaffected by small perturbations of a given input $\mathbf{x}'$. In the case of image classifiers, local robustness checks if all images within a norm-ball of $\mathbf{x}'$ are classified equivalently. The problem can be represented by setting $\mathcal{X}_0 = \{\mathbf{x} \in \mathbb{R}^{s_0} \mid \|\mathbf{x} - \mathbf{x}'\|_p \leq \epsilon\}$, for some $\epsilon \geq 0$ and norm p, and $\mathcal{X}_k = \{\mathbf{x}_k \in \mathbb{R}^{s_k} \mid \forall i \neq c\colon (\mathbf{x}_k)_i < (\mathbf{x}_k)_c\}$, where $(\mathbf{x}_k)_j$ is the j-th component of $\mathbf{x}_k$ and c is the class of $\mathbf{x}'$.

The reachability problem establishes if there exists an admissible input in a given set $\mathcal{I}$ for which the network computes a given output $\mathbf{y}$. The reachability problem is not directly expressible as the verification problem defined above as it includes an existential quantification over the inputs. The dual problem can, however, be represented by taking $\mathcal{X}_0 = \mathcal{I}$ and $\mathcal{X}_k = \mathbb{R}^{s_k} \setminus \{\mathbf{y}\}$. Therefore, the answer to the reachability problem is the complement of the answer to the verification problem encoded as the dual above.

MILP formulation. The verification problem admits a *precise* representation as a Mixed Integer Linear Program (MILP) by means of the "big-M" encoding [1]. Specifically, the corresponding MILP program is feasible iff the answer to the verification problem is *no*. Assuming the pre-activation bounds of the nodes have already been calculated (see below), the MILP encoding of a node $n_{i,q}$ depends on its state. If the node is strictly active, then it can be encoded by $\mathbf{x}_{i,q} = \hat{\mathbf{x}}_{i,q}$. If the node is strictly inactive, then it can be encoded by $\mathbf{x}_{i,q} = 0$. Otherwise, the encoding of the node is given by:

$$\begin{aligned} &\mathbf{x}_{i,q} \geq 0, && \mathbf{x}_{i,q} \geq \hat{\mathbf{x}}_{i,q}, \\ &\mathbf{x}_{i,q} \leq \hat{\mathbf{u}}_{i,q} \cdot \delta_{i,q}, && \mathbf{x}_{i,q} \leq \hat{\mathbf{x}}_{i,q} - \hat{\mathbf{l}}_{i,q} \cdot (1 - \delta_{i,q}), \end{aligned}$$

where $\delta_{i,q}$ is a binary variable such that $\delta_{i,q} = 0$ iff $\mathbf{x}_{i,q} = 0$ and $\delta_{i,q} = 1$ iff $\mathbf{x}_{i,j} = \hat{\mathbf{x}}_{i,q}$.

For an MILP program comprising a set Δ of binary variables, a (partial) *configuration* is a (partial) function $h : \Delta \to \{0, 1\}$ that assigns to each variable (some of the variables) a value from $\{0, 1\}$. The set of all possible partial configurations is said to be the program's *configuration space*.

A leading approach for solving MILP programs is the branch-and-bound method. In branch-and-bound, each integrality constraint $\delta_{i,j} \in \{0, 1\}$ is relaxed to a linear constraint $\delta_{i,j} \in [0, 1]$, thereby defining a linear program which can be solved in polynomial-time [12]. Integrality is iteratively enforced by dividing the search domain into sub-regions excluding fractional solutions. In the context of neural networks, the efficacy of branch-and-bound depends on (i) the number of binary variables, i.e., the number of unstable nodes, and (ii) the tightness of the linear relaxations, i.e., the tightness of the pre-activation bounds.

Calculating bounds. Interval arithmetic derives pre-activation bounds by propagating the interval of the input domain through the network. However, the resulting bounds are often over-approximated as the method neglects dependencies between the input nodes, and propagating the over-approximated bounds leads to larger over-approximations following each layer. To enable tighter approximations, rather than propagating concrete intervals, approaches based on symbolic interval analysis [25] define linear equations for the lower and upper bounds which are built from variables expressing the inputs of the network. To tackle the non-linearity of the ReLU function, propagating the equations involves their linear relaxation [25].

Lastly, even tighter bounds can be obtained by splitting the input domains into several sub-domains and solving the verification problem for each sub-domain [26, 25, 14].

3 Dependency Analysis

As discussed in the previous section, a major impediment to the scalable verification of ReLU-based FFNNs is the configuration space generated by the piecewise linearity of the ReLU nodes. Several approaches have been put forward for reducing the number of non-linearities that need to be considered for solving the verification problem. In particular, techniques that split the input domain have been shown effective in stabilising the ReLU nodes, thereby generating easier verification problems whose solutions can be combined to decide the original problem in a more efficient manner. Still, since the number of splits that need to be carried out grows exponentially in the number of input dimensions, networks with high input dimensionality remain hard to tackle. To overcome this, we introduce a technique that exploits what we define below as the *network's dependency relation* to reduce the configuration space that needs to be considered in solving a verification problem. Informally, the network's dependency relation can be used to stabilise a ReLU node on the basis of an assumed stable state of another node. Formally it is defined as follows.

Definition 1 (Dependency relation). *Given a neural network f that comprises a set of unstable nodes U, the* dependency relation for U, $\mathcal{D}_f \subseteq U \times U$ *is the set of all pairs* $(n_{i,q}, n_{j,r})$ *such that* $\mathsf{st}(n_{i,q}) \neq ? \implies \mathsf{st}(n_{j,r}) \neq ?$.

A node $n_{j,r}$ depends on a node $n_{i,q}$ if whenever $n_{i,q}$ is either strictly active or inactive, then $n_{j,r}$ has to be either strictly active or inactive. It follows that the configuration space generated by a branch-and-bound method can be reduced by stabilising $n_{j,r}$ whenever $n_{i,q}$ becomes stable. In particular, for a network with n unstable nodes, there are 2^{n-2} configurations that violate a given dependency; therefore, each dependency provides a means to reduce the configuration space by a factor of $1/4$.

Example 1. Consider the network shown in the left part of Figure 1. In the figure each interval next to a node denotes the pre-activation bounds of the node. Note

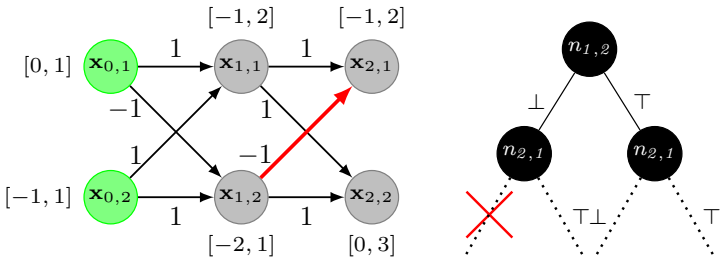


Fig. 1. Left: Feedforward neural network exhibiting the dependency "if $n_{1,2}$ is inactive, then $n_{2,1}$ is active". **Right:** Depiction of the configuration space reduction induced by the dependency.

that nodes $n_{1,2}$ and $n_{2,1}$ are unstable. Assume that a branch-and-bound method branches on node $n_{1,2}$, thereby splitting the optimisation problem into two sub-problems: one where $n_{1,2}$ is strictly active and one where $n_{1,2}$ is strictly inactive. Consider the latter sub-problem. We have that $\mathbf{l}_{1,2} = 0$ and $\mathbf{u}_{1,2} = 0$. Therefore, $\hat{\mathbf{l}}_{2,1} = 1 \cdot 0 + -1 \cdot 0 = 0$ and $\hat{\mathbf{u}}_{2,1} = 1 \cdot 2 - 1 \cdot 0 = 2$. Hence, $n_{2,1}$ is strictly active, and consequently, $(n_{1,2}, n_{2,1}) \in \mathcal{D}_f$. The right part of Figure 1 depicts the configuration space satisfying said dependency.

We now proceed to derive a procedure for computing a network's dependency relations. To ease the presentation, we express dependency relations as unions of four disjoint sets $\mathcal{D}_f = \bigcup_{z,z' \in \{\top,\bot\}} \mathcal{D}_f^{z,z'}$, where each $\mathcal{D}_f^{z,z'}$ comprises dependencies pertaining to the ReLU states z and z', i.e.,

$$\mathcal{D}_f^{z,z'} \triangleq \{(n_{i,q}, n_{j,r}) \mid \mathsf{st}(n_{i,q}) = z \implies \mathsf{st}(n_{j,r}) = z'\}\,.$$

Also, we distinguish between consecutive- and intra-layer dependencies, which require a different algorithmic treatment. We begin by studying dependencies in the same layer.

Intra-layer dependencies. A dependency $(n_{i,q}, n_{j,r})$ is said to be an *intra-layer dependency* if $i = j$. To compute the dependency relation, given a pair of nodes, we compute the lower and upper bounds of a node under the assumption that the pre-activation of the other is zero, and use the bounds to determine the dependencies.

Formally, for a pair of nodes $n_{i,q}$, $n_{i,r}$, we define $\hat{\mathbf{x}}_{i,q,r=0}$ as the set of pre-activations of $n_{i,q}$ when the pre-activation of $n_{i,r}$ is zero:

$$\hat{\mathbf{x}}_{i,q,r=0} \triangleq \{(W_i)_q \cdot \mathbf{x}_{i-1} + (b_i)_q \mid (W_i)_r \mathbf{x}_{i-1} + (b_i)_r = 0\}\,.$$

Geometrically, this can be viewed as the intersection of the plane generated by the pre-activations of $n_{i,q}$ and $n_{i,r}$ with $\hat{\mathbf{x}}_{i,r} = 0$. Note that said intersection always exists as both $n_{i,q}$ and $n_{i,r}$ are unstable; therefore, there is an input for which their pre-activations equal zero. On the basis of the lower and upper

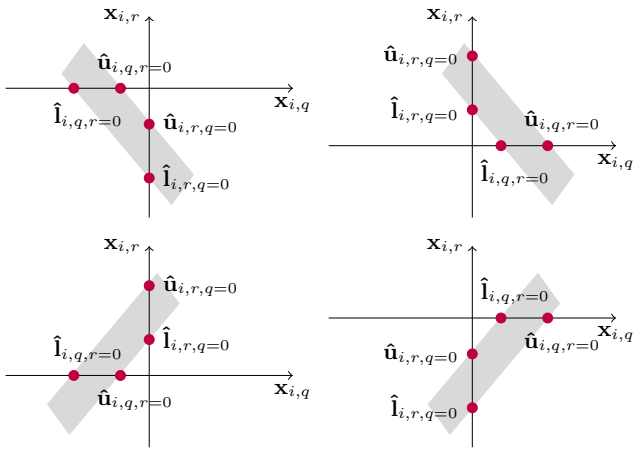


Fig. 2. The types of intra-layer dependencies. **Top-left:** if $n_{i,q}$ is active, then $n_{i,r}$ is inactive. **Top-right:** if $n_{i,q}$ is inactive, then $n_{i,r}$ is active. **Bottom-left:** if $n_{i,q}$ is active, then $n_{i,r}$ is active. **Bottom-right:** If $n_{i,q}$ is inactive, then $n_{i,r}$ is inactive.

bounds of $\hat{\mathbf{x}}_{i,q,r=0}$ and $\hat{\mathbf{x}}_{i,r,q=0}$, which can be computed as standard using interval arithmetic, the following lemma identifies the intra-layer dependencies (see Figure 2).

Lemma 1. *For a neural network f and a pair of unstable nodes $(n_{i,q}, n_{i,r})$, the following hold:*

1. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\top,\perp}$ *iff* $\hat{\mathbf{u}}_{i,q,r=0} < 0$ *and* $\hat{\mathbf{u}}_{i,r,q=0} < 0$.
2. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\perp,\top}$ *iff* $\hat{\mathbf{l}}_{i,q,r=0} > 0$ *and* $\hat{\mathbf{l}}_{i,r,q=0} > 0$.
3. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\top,\top}$ *iff* $\hat{\mathbf{u}}_{i,q,r=0} < 0$ *and* $\hat{\mathbf{l}}_{i,r,q=0} > 0$.
4. $(n_{i,q}, n_{i,r}) \in \mathcal{D}_f^{\perp,\perp}$ *iff* $\hat{\mathbf{l}}_{i,q,r=0} > 0$ *and* $\hat{\mathbf{u}}_{i,r,q=0} < 0$.

Lemma 1 gives a procedure for identifying intra-layer dependencies by computing the right hand side of each of the above clauses for every pair of unstable nodes in a layer. Dependencies between layers require a different treatment.

Consecutive-layer dependencies. A dependency $(n_{i,q}, n_{j,r})$ is said to be an *consecutive-layer dependency* if $j = i$. To obtain a procedure for calculating consecutive-layer dependencies we introduce the result below.

Lemma 2. *For a neural network f and a pair of unstable nodes $n_{i,q}, n_{j,r}$, for $j = i + 1$, the following hold:*

1. $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{\perp,\perp} \Leftrightarrow \hat{\mathbf{u}}_{j,r} - (W_j)_{r,q} \cdot \mathbf{u}_{i,q} \leq 0$.
2. $(n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{\perp,\top} \Leftrightarrow \hat{\mathbf{l}}_{j,r} - (W_j)_{r,q} \cdot \mathbf{u}_{i,q} \geq 0$.
3. $\mathcal{D}_f^{\top,\perp} = \emptyset$
4. $\mathcal{D}_f^{\top,\top} = \emptyset$

Lemma 2 gives a procedure for identifying consecutive-layer dependencies by checking the right hand side of clauses (1) and (2) for every pair of unstable nodes in consecutive layers.

Dependency analyser. Given the above, we now put forward a procedure using the identification of dependencies to reduce the configuration space. The procedure runs in conjunction with a MILP solver, where it builds a new constraint for each dependency which it adds to the program being analysed by the solver. This is performed at runtime during the branch-and-bound procedure, as the computation of the dependencies is consistent with the current, partial configuration of ReLU nodes being considered by the MILP solver. This allows for the identification of dependencies whilst several nodes have already been stabilised, as opposed to offline methods where most nodes would typically be unstable, thereby hindering the existence of dependencies, as it is rarer for a node to cause a state change on another.

Consider a partial configuration h being considered by the MILP solver. To determine its validity, the solver either extends it to a complete one that satisfies all constraints or to a partial one that violates at least one of the constraints. The dependency analysis procedure put forward here reduces the number of extensions of h that need to be evaluated. Algorithm 1 enumerates the steps of the procedure. First, it stabilises the ReLU nodes as per h and re-computes the bounds for the ones being unstable under h. On the basis of the bounds, it determines the dependencies as per Lemmas 2 and 1. These are then expressed as constraints, referred to as *dependency cuts*, which are added at runtime to the MILP program. The dependency cuts are defined as follows.

Definition 2 (Dependency cuts). *For a partial configuration* $h\colon \Delta \to [0,1]$*, the associated* dependency cut $cut_{d,h}$ *of a dependency* $d \triangleq (n_{i,q}, n_{j,r}) \in \mathcal{D}_f^{z,z'}$ *is a MILP constraint defined as follows:*

$$cut_{d,h} \triangleq \gamma_{j,r}(z') \leq \sum_{h(\delta)=0} \delta + \sum_{h(\delta)=1} 1 - \delta + \gamma_{i,q}(z),$$

where $\gamma_{i,q}(z)$ *equals* $\delta_{i,q}$ *if* $z = \bot$ *and* $1 - \delta_{i,q}$ *if* $z = \top$*.*

A dependency cut derived from a configuration h is satisfied by an extension of h iff the extended configuration satisfies the corresponding dependency; it follows that each dependency cut removes from the search space all configurations extending h that do not satisfy the dependency. Additionally, for any configuration that does not extend h, the cut is trivially satisfied, thereby not altering the search space for those configurations. The former is shown by clause (1) and the latter is proved by clause (2) of the following theorem.

Theorem 1. *Let* h *be a partial configuration and* $d \in \mathcal{D}_f^{z,z'}$ *a dependency. Then, the following hold:*

1. *For every* h' *with* $h \subseteq h'$*,* $h' \models cut_{d,h}$ *iff* $h' \models d$*.*
2. *For every* h' *with* $h \not\subseteq h'$*,* $h' \models cut_{d,h}$*.*

Algorithm 1 The dependency analysis procedure.

procedure DEPENDENCY ANALYSIS(milp, h)
  Input: MILP milp, partial configuration h.
  for each $h(\delta_{i,q}) = 0$ **do**
    $\mathbf{l}_{i,q} \leftarrow 0$, $\mathbf{u}_{i,q} \leftarrow 0$
  Compute remaining bounds (Section 2).
  Compute $\mathcal{D}_f$ (Lemmas 2 and 1).
  Add $\left\{ cut_{d,h} \mid d \in \mathcal{D}_f \right\}$ to milp (Definition 2)

Algorithm 2 The verification procedure.

procedure VERIFY(N, $(\mathcal{X}_0, \mathcal{X}_k)$)
  Input: network N, specification $(\mathcal{X}_0, \mathcal{X}_k)$
  Output: YES/NO
  sub-problems $\leftarrow$ split$(N, (\mathcal{X}_0, \mathcal{X}_k))$
  result $\leftarrow$ YES
  for P **in** sub-problems **do**
    milp $\leftarrow$ encode(P)
    sub-result $\leftarrow$ milp_solver(milp)
    if sub-result is feasible **then**
      result $\leftarrow$ NO
      break
  return result

The above concludes the description of the dependency analysis procedure. The procedure runs in time $\mathcal{O}(k \cdot s^2)$, where k is the number of layers and s is the layers' maximal size. Clearly running this procedure has a cost. In the next section we will experimentally evaluate how frequent these calls should be. Also, note that since our dependency framework is a function of the bounds of the ReLU nodes, the procedure can further be optimised by using domain splitting and symbolic interval propagation methods, since these lead to tighter intervals for the ReLU nodes.

In the next section we show that all these factors combined improve the scalability of formal verification of neural networks over the state-of-the-art.

4 The Venus Verification Tool

In this section we introduce Venus [24], a verification toolkit that implements the dependency analysis procedure and augments it with symbolic interval arithmetic and domain splitting techniques. While methods on domain splitting divide the input domain into sub-domains, thereby tightening the nodes' bound intervals, methods on symbolic interval arithmetic enable the efficient and tight approximation of the latter; therefore, by Lemmas 1 and 2, both methods promote the existence of dependencies.

The verification procedure upon which Venus is based is outlined in Algorithm 2. The procedure follows a divide-and-conquer approach whereby it re-

Algorithm 3 The splitting procedure.

```
1: procedure SPLIT(P)
2:     Input: verification problem P = (N, (X_0, X_k))
3:     Output: a set of verification sub-problems.
4:     d ← 1                                              ▷ Splitting depth
5:     tosplit ← [(d, X_0)]
6:     sub-problems ← []
7:     while tosplit not empty do
8:         d, R ← pop top element of tosplit
9:         R_1, R_2 ← best_split(R)
10:        if worth_split(R, R_1, R_2, d) then
11:            add (d + 1, R_1), (d + 1, R_2) to tosplit
12:        else
13:            add (N, (R, X_k)) to sub-problems
14:    return sub-problems
```

cursively splits the input domain until certain heuristic criteria are met and solves the verification sub-problems associated with each sub-domain. Each sub-problem is encoded as a MILP program. These can be analysed in parallel. A MILP program is feasible iff the answer to its associated verification problem is "no". By the definition of the verification problem (Section 2), the answer to the original problem is "no" iff there is at least one sub-problem whose answer is "no". So, as soon as one of the sub-problems is found to be feasible, the procedure terminates without analysing the remaining MILP programs.

Venus uses the "big-M" encoding for the verification problems, and strengthens the linear relaxations by adding *dependency cuts* and *"ideal cuts"* [2] to the MILP programs. The cuts are added at runtime through solver callbacks. Although the cuts strengthen the relaxation, they add complexity to the sub-problems within the solver. Therefore, the addition of a large number of cuts can slow down the solver. Following this, cuts are only added in a fraction of all solver callbacks.

Splitting procedure. The splitting procedure is outlined by Algorithm 3. The procedure recursively splits the input domain by selecting at each step one of the input dimensions and dividing its range in half.

The dimension is heuristically selected on the basis of what we call the *stability-ratio*, the ratio of stable to total number of nodes for a given network and input domain (Line 9). In particular, for each input dimension, we bisect the input domain along the dimension, compute the stability-ratio for each of the two resulting sub-domains and record the average stability-ratio. Then, the dimension along which to split is selected as the one that maximises the recorded averages, or, equivalently, as the one that achieves (on average) the greatest reduction of the configuration space of the induced sub-problems.

Clearly, the number of splits that need to be performed in order to obtain (significantly) simpler sub-problems grows in the number of dimensions. As a result, since the number of sub-problems grows exponentially in the number of

input dimensions, the number of sub-problems that need to be considered grows exponentially in the number of dimensions. So, whereas verification problems for networks with low input dimensionality can effectively be divided into a number of small sub-problems that are easier to solve, problems for networks with high input dimensionality render such partitions intractable. As reported in the next section, a key advantage of Venus over related tools lies in its ability to solve both low and high input dimensionalities. While domain splitting is very effective for low input dimensionalities, MILP solvers in conjunction with dependency analysers are very powerful for high input dimensionalities. Venus combines the two approaches by considering a heuristic criterion that terminates splitting and signals the employment of an MILP solver (Line 10). The criterion expresses an estimation of the difficulty of the verification problem before splitting versus its difficulty after splitting.

The estimation of the difficulty of a problem p at splitting depth d that we consider is defined by

$$score(p, d) = \frac{stability\_ratio(p) - stability\_ratio(P)}{d^{\frac{1}{m}}},$$

where P is the original verification problem and m is the *splitting parameter*. The larger the score the less difficult the verification problem is estimated to be. The score rewards the improvement of the stability ratio with respect to the original problem and penalises large splitting depths. The splitting parameter controls the degree of "discount" to the splitting depth penalty, where higher values of m signify larger discount. So, following the above discussion, in the case of problems over networks with low input dimensionality, the splitting parameter should be kept high so as to favour splitting. Differently, for problems over networks with high input dimensionality, the splitting parameter should be kept low in order to discourage splitting.

Given a problem p at splitting depth d, and the sub-problems p_1 and p_2 resulting from splitting the chosen dimension of the input domain of p, the splitting is carried out only if the score of (p, d) is less than the average of the scores of $(p_1, d + 1)$ and $(p_2, d + 1)$. In cases where excessive splitting is still observed, a cut-off stability-ratio is used above which the splitting process terminates independently of the aforementioned scores.

Implementation. The architecture of Venus is shown in Figure 3. The toolkit comprises the following components: (i) the *Splitter* performing domain splitting and adding the derived sub-problems to the jobs queue; and (ii) the *Worker* reading sub-problems from the jobs queue, solving them by calling an MILP-solver and dependency analyser ensemble, and recording the verification results to the results queue. Venus aggregates the results from the workers and reports the combined verification result as per Algorithm 2. Both the *Splitter* and *Worker* follow a parallelisation scheme whereby several splitters and workers carry out the domain splitting and the MILP analysis in parallel. Venus is implemented in Python 3.7 and relies on Gurobi 8.1 for the MILP backend.

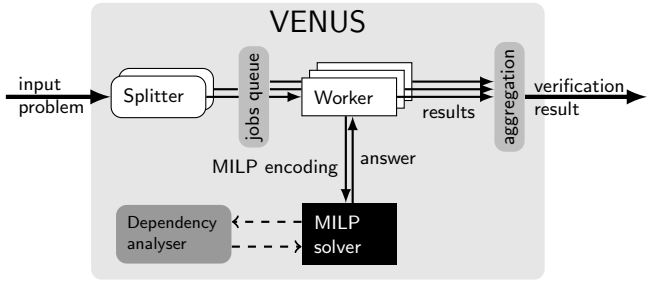


Fig. 3. The architecture of Venus.

	MNIST (100 queries)					**CIFAR-10** (100 queries)					**ACAS XU** (172 queries)				
	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t^{\text{Venus}}_{all}}$	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t^{\text{Venus}}_{all}}$	n_s	t_{all}	t_{solved}	av_s	$\frac{t_{all}}{t^{\text{Venus}}_{all}}$
Venus	100	5953	573	9	–	100	857	560	7	–	170	19643	5,528	36	–
Marabou	0	86400	–	–	14	0	86400	–	–	101	156	140916	75,747	498	7
Neurify	65	126007	7	0	21	76	87178	778	10	102	167	23628	2555	17	1
NSVerify	95	26907	2515	40	5	100	6899	3460	46	8	6	86400	–	–	31

Table 1. Experimental results obtained when running Venus, Marabou, Neurify and NSVerify.

5 Experimental Results and Evaluation

In this section we evaluate Venus on a number of widely used benchmarks and compare it against the state-of-the-art neural-network verification engines.

For the comparisons we restrict our attention to *complete* methods; while these are often less scalable than incomplete ones, they provide full guarantees on the correctness of their outputs, which is a key objective here. At present, the leading complete verification tools are Marabou [14] and Neurify [25]. To assess the improvement of Venus over plain MILP-based verification, we additionally compare Venus against NSVerify [1]. We used the most commonly used benchmarks in the context of FFNNs verification:

- **ACAS Xu** [11] comprises 45 ReLU-based FFNNs, which were developed as part of an airborne collision avoidance system to advise horizontal steering decisions for unmanned aircrafts. We considered the specifications reported in [13]. Each was tested on all 45 networks, thereby giving rise (for a total of 10 specifications) to 172 verification problems. For the experiments, Venus was run with 2 splitters, 4 workers, the stability-ratio cutoff set to 0.7, the depth discount set to 20 and with the dependency analyser turned off; Neurify was run with `MAX_THREAD` set to 2; Marabou was run with the parameters reported in [14].

- **MNIST** [17] is a dataset comprising images of hand-written digits 0-9, each formatted as a 28x28x1-pixel grayscale image. We used MNIST to train a

FFNN with 2 hidden layers, each layer comprising 512 neurons. The prediction accuracy of the network is 97%. We verified the network against local robustness for a perturbation radius of 0.05 on 100 randomly selected images. For the experiments, we ran Venus with 2 splitters, 2 workers, the stability-ratio set to 0.4, the depth discount set to 4, and the dependency analyser turned on; Neurify was run with `MAX_THREAD` set to 1; Marabou was run with the parameters reported in [14].

– **CIFAR-10** [16] is a dataset comprising images of objects from 10 different classes (airplanes, cars, birds, cats, etc.). Each image is formatted as a 32x32x3-pixel colour image. We used CIFAR-10 to train a FFNN with 3 hidden layers, the first layer comprising 1024 neurons, and the second and third layers comprising 512 neurons. The prediction accuracy of the network is 45%. We verified the network against local robustness for a perturbation radius of 0.01 on 100 randomly selected images. We ran all tools with the same parameters as for MNIST.

All experiments were carried out on an Intel Core i7-7700K (4 cores) equipped with 16GB RAM, running Ubuntu 18.04. Each verification query had a *local* timeout of 1 hour. The sum of verification queries associated with each benchmark had a *global* timeout of 24 hours. Table 1 reports the experimental results. For each of the tools and benchmarks, the table gives the number n_s of verification queries that were solved, the overall time t_{all} taken for all queries, the overall time t_{solved} and the average time av_s taken for the queries that three best performing tools were able to solve, and the ratio between overall time taken t_{all} by a tool over that of Venus, t_{all}^{Venus}.

The results obtained on MNIST show that Venus was the most performing of the toolkits, both in terms of the overall verification time and the number of verification queries solved. Neurify was not able to analyse 35 of the queries because of excessive memory consumption. For these cases, we considered Neurify as having timed out. Marabou did not solve any of the queries under local and global timeouts. NSVerify performed better than both Neurify and Marabou. Venus was found 4.52 times faster than NSVerify and 21 times faster than Neurify. For CIFAR-10 the difference between Venus and the other tools was greater, suggesting that the higher the dimensionality and complexity of the model, the bigger the difference.

Venus's performance was also found superior on ACAS XU, both in terms of the overall verification time and the number of queries solved. Neurify was the fastest tool w.r.t. the number of queries that all the tools could solve. Marabou solved a comparable number of queries to Venus and Neurify but was slower than both of them. NSVerify solved only 6 queries within the local and global timeouts.

Figure 4 gives a graphical representation of the total number of verification queries that each tool could verify as a function of time. In summary, Venus solved most verification instances after approx. 15 secs. Also, to the best of our knowledge, Venus is the only tool that can seemingly analyse both low-

Ablation test	n_s	av_s	t_{all}	t_{solved}
Big-M formulation	98	38.15	13,555.49	3,700.62
Splitting	98	42.58	11,409.96	4,129.83
Ideal formulation	100	36.02	6,777.95	3,460.17
Splitting+Ideal	100	36.75	8,277.85	3,565.02
Consecutive Deps	99	30.06	8,561.21	2,916.23
Intra Deps	98	33.03	10,426.82	3,203.54
Consecutive+Intra Deps	98	25.81	9,729.07	2,503.41
All methods enabled	100	26.52	5,953.46	2,572.90

Table 2. Ablation experiments for MNIST. The average is calculated for the images that are verified in all cases. Similarly, t_{solved} is calculated for the images verified in all cases.

dimensional and high-dimensional networks, outperforming the state-of-the-art tools for each class, often by more than one order of magnitude. The only aspect we found Venus to be less performing was counterexample generation where Neurify was the fastest tool.

The experiments also suggest that the verification of networks with low input-dimensionality is particularly amenable to domain splitting, as domain splitting techniques act as effective configuration-space minimisers. As a result, branch-and-bound methods that combine domain splitting are advantageous over ones that do not, as indicated by the outperformance of Venus over NSVerify on ACAS.

Differently, for high-dimensional input domains, the experiments suggest that domain splitting methods do not significantly reduce the configuration space, as indicated by the degraded performance of Neurify and Marabou on MNIST and CIFAR-10. In contrast, techniques that directly target the reduction of the configuration space exhibit high efficacy over high-dimensional inputs, as exemplified by Venus and NSVerify. Venus is more effective than NSVerify by considering FFNN-specific configuration-space reductions. This suggests that MILP solvers are not necessarily best used as black boxes, but application-specific considerations can help to improve their effectiveness.

Indeed, the performance gains exhibited by Venus over NSVerify on MNIST and CIFAR-10 are a consequence of combining dependency analysis and ideal formulations. This is evidenced by separately evaluating Venus on MNIST for different combinations of the techniques that the tool implements. Table 2 reports ablation experiments to analyse this in detail. The results confirm that domain splitting is not effective for high dimensional inputs. They also show that ideal formulations and dependency analysis improve on pure big-M formulations not only when they are jointly utilised but also when they are independently employed. In the latter case, ideal formulations enabled the verification of two images that could not be verified by dependency analysis, whereas dependency analysis led to better average and total time for verifying all images that could be verified in all cases. In the latter case the results suggest that the combina-

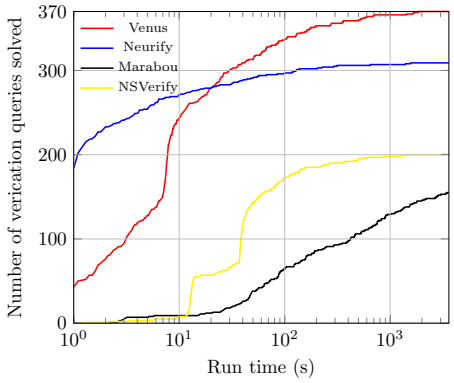


Fig. 4. Number of verification queries that Venus, Neurify, Marabou and NSVerify could solve as a function of time.

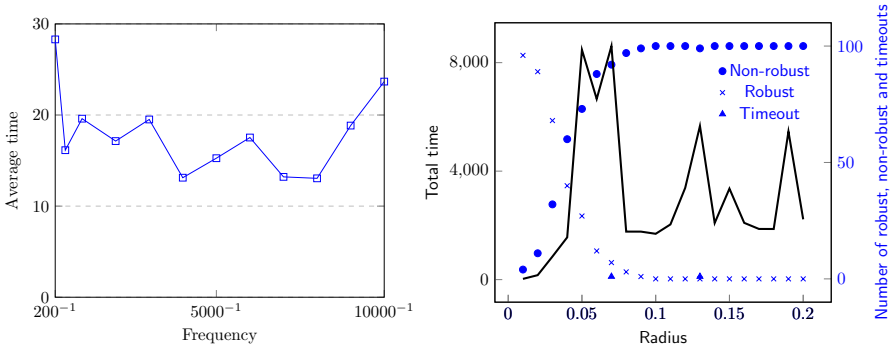


Fig. 5. Left: Average runtime of Venus on MNIST as a function of callback frequency. **Right:**Total runtime of Venus on MNIST as a function of perturbation radius.

tion of dependency analysis and ideal formulations is preferable in terms of all performance metrics than either technique considered in isolation.

As discussed in Section 3, running the dependency analysis procedure has a cost. As a result, the above performance gains can only be obtained after determining how often these calls should be. The left part of Figure 5 gives the average runtime of Venus on 100 MNIST images as a function of the frequency with which the dependency analysis procedure is called from a Gurobi callback. High and low frequencies degrade the performance of Venus, whereas frequencies in the range $[0.00014 - 0.00025]$ balance out the cost of computing dependencies and the reduction of the configuration space enabled by their computation.

We conclude this section by studying the performance of Venus as a function of the perturbation radius for which the robustness of MNIST is established. Intuitively, small perturbation radiuses pertain to easy verification problems on

the one hand, as the bounds for the nodes are tighter, and to hard problems on the other hand, as the network is more likely to be robust w.r.t. the corresponding perturbed images. Similarly, large perturbation radiuses pertain to hard verification problems on the one hand, as the bounds for the nodes are looser, and to easy problems on the other hand, as the network is less likely to be robust w.r.t. the corresponding perturbed images. The right part of Figure 5 reports Venus's total running time for verifying 100 MNIST images and for perturbation radiuses that range from 0.01 to 0.2. The figure also shows the number of images for which Venus has timed out and for which the network was found non-robust and robust. The figure shows that Venus is consistently efficient for all perturbation radiuses (with an average verification time per image of less than 90 seconds). The figure also indicates that the performance of Venus is mostly degraded for perturbation radiuses within the range $[0.05, 0.07]$. Notably, these radiuses result in verification problems that do not permit for sufficiently tight bounds for the nodes whilst not exhibiting sufficiently adversarial regions.

6 Conclusions

As we argued in the introduction, the deployment of learning methods based on neural networks in safety critical AI applications urgently requires verification and validation methods. A growing area of research is concerned with the development of formal verification methods for neural networks with particular emphasis to ReLU-based deep networks used in vision and control. While progress in this area has been rapid, the present state-of-the-art still falls short of the capabilities required to verify industry-strength models. It is unlikely that this scalability issue will be solved in the immediate future; but there is a need for novel methods to gradually conquer larger and larger networks.

In this paper we introduced the concept of dependency analysis which we developed in the context of a MILP-based verification method. The method further benefits from input splitting and symbolic interval propagation. We derived algorithms based on the resulting theory and reported the results obtained with Venus, a novel tool for the verification of neural networks. As we demonstrated experimentally on three different, widely used benchmarks, Venus could solve more verification queries than the present state-of-the art tool based on complete methods. Venus is also the fastest tool to verify the correctness of a network; in some cases Neurify proved to be faster in finding counterexamples.

In future work we intend to apply Venus to the verification of more complex specifications for neural networks including transformational robustness [15]. Also we intend to analyse alternative methods for the calculation of the bounds, such as those employed by ERAN [22] and CROWN [27], in the context of dependency analysis and study the extend to which the methods can be used to improve the performance of Venus.

Acknowledgements. This research was partly funded by DARPA under the Assured Autonomy program, the Royal Academy of Engineering (via a Chair

in Emerging Technologies), the Royal Society (NIF\R1\182194), and ESPRC (grant EP/P016871/1).

References

1. M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano. Reachability analysis for neural agent-environment systems. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR18)*, pages 184–193. AAAI Press, 2018.
2. R. Anderson, J. Huchette, C. Tjandraatmadja, and J.P. Vielma. Strong mixed-integer programming formulations for trained neural networks. In *Integer Programming and Combinatorial Optimization (IPCO19)*, Lecture Notes in Computer Science, pages 27–42. Springer, 2019.
3. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS16)*, pages 2613–2621, 2016.
4. R. .R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. K. Mudigonda. A unified view of piecewise linear neural network verification. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS18)*, pages 4790–4799. Curran Associates, Inc., 2018.
5. C.-H. Cheng, G. Nührenberg, and H. Ruess. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis*, pages 251–268. Springer International Publishing, 2017.
6. K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. In *Proceedings of the 34th Annual Conference on Uncertainty in Artificial Intelligence (UAI18)*, pages 162–171. AUAI Press, 2018.
7. R. Ehlers. 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*, pages 269–286. Springer, 2017.
8. M. Fazlyab, M. Morari, and G. J. Pappas. Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. *arXiv preprint arXiv:1903.01287*, 2019.
9. M. Fischetti and J. Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, pages 1–14, 2018.
10. T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. AI^2: Safety and robustness certification of neural networks with abstract interpretation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P18)*, pages 948–963, 2018.
11. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th Digital Avionics Systems Conference (DASC16)*, pages 1–10, 2016.
12. N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC84)*, pages 302–311. ACM, 1984.
13. G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. 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*, pages 97–117. Springer, 2017.
14. G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV19)*, pages 443–452, 2019.
15. P. Kouvaros and A. Lomuscio. Formal verification of cnn-based perception systems. *arXiv preprint arXiv:1811.11373*, 2018.
16. A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 dataset. http://www.cs.toronto.edu/kriz/cifar.html, 2014.
17. Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits, 1998.
18. C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for verifying deep neural networks. *CoRR*, abs/1903.06758, 2019.
19. A. Lomuscio and L. Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.
20. A. Raghunathan, J. Steinhardt, and P. S. Liang. Semidefinite relaxations for certifying robustness to adversarial examples. In *Proceedings of 31st Annual Conference on Neural Information Processing Systems (NeurIPS18)*, pages 10900–10910, 2018.
21. V. Rubies-Royo, R. Calandra, D. M. Stipanovic, and C. Tomlin. Fast neural network verification via shadow prices. In *Proceedings of the 36th International Conference on Machine Learning (ICML19)*, 2019.
22. G. Singh, T. Gehr, M. Püschel, and P. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41, 2019.
23. V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *Proceedings of the 7th International Conference on Learning Representations (ICLR19)*, 2019.
24. Venus. https://vas.doc.ic.ac.uk/software/neural, 2019.
25. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems 2018 (NeurIPS18)*, pages 6369–6379, 2018.
26. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Security Symposium, (USENIX18)*, pages 1599–1614, 2018.
27. H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems 2018 (NeurIPS2018)*, pages 4944–4953. Curran Associates, Inc., 2018.

Robustness Verification for Ensemble Stumps and Trees

Hongge Chen[2]*, Yihan Wang[3]*, Huan Zhang[1]*, Si Si[4], Yang Li[4], Duane Boning[2], and Cho-Jui Hsieh[1]

[1] University of California, Los Angeles, USA
[2] Massachusetts Institute of Technology, Massachusetts, USA
[3] Tsinghua University, China
[4] Google Research, USA
* Equaly contributed, ranked by alphabetical order

Abstract. We study the robustness verification problem for ensemble decision stumps and trees, including random forest, gradient boosting trees, and Adaboost. Although these models are widely used in practice, there is very limited understanding on how to formally verify the robustness of those models. In this study, we aim to give a comprehensive complexity analysis as well as provide efficient verification algorithms. For ensemble decision stumps, we show that exact robustness verification with L_p norm ball is NP-complete for $p \in (0, \infty)$, while polynomial time algorithms exist for $p = 0$ and $p = \infty$. Approximation algorithms based on dynamic programming are then developed for verifying ensemble stumps for $p \in (0, \infty)$. For ensemble decision trees, it has been proved that exact robustness verification is NP-complete, and the existing verification approach is based on MILP, which does not scale to large-scale problems. We show that ensemble tree verification can be cast as a max-clique problem on a multi-partite graph with bounded boxicity, and by exploiting the boxicity of the graph, we develop an efficient multi-level verification algorithm that can give tight lower bounds on robustness of ensemble decision trees, while allowing iterative improvement and any-time termination.

1 Introduction

Machine learning verification aims to develop methods to bound the behavior of a model within a given input set, and they have become fundamental tools for verifying robustness and safety properties of given models. In this paper, we study the robustness verification problem of ensemble decision stumps and trees, which covers several important machine learning models such as AdaBoost, Random Forests (RFs) and Gradient Boosted Decision Trees GBDTs). These models have been widely used in practice [7, 12, 22] and recent studies have demonstrated that they are vulnerable to adversarial perturbations [10, 8, 6], but there is limited understanding on how to efficiently verify them.

We focus on the robustness verification problem, which can be defined as finding the minimum adversarial perturbation within a given input region (usually an ℓ_p norm ball). [10] showed that computing minimum adversarial perturbation for tree ensemble is NP-complete in general, and they proposed a Mixed-Integer Linear Programming (MILP) based approach to compute the minimum adversarial perturbation. Although exact verification is NP-hard for general tree ensemble, in order to have an efficient verification algorithm for real applications we seek to answer the following questions:

- Do we have polynomial time algorithms for exact verification under some special circumstances?
- For general tree ensemble models with a large number of trees, can we efficiently compute meaningful lower bounds on robustness while scaling to large tree ensembles?

In this paper, we provide the answers to the above-mentioned questions. Our contributions can be summarized below:

- **Robustness Verification for Ensemble Decision Stumps**: For an ensemble of decision stumps (trees with depth 1), we show that there is a fundamental different between the complexity of verifying ℓ_p norm ball with different p. When $p \in (0, \infty)$, we prove that ℓ_p norm verification problem is NP-complete while polynomial time algorithms exist for $p = 0, \infty$. However, we are able to propose an efficient dynamic programming algorithm that can compute a reasonably tight verification bound efficiently for general p.
- **Robustness Verification for Ensemble Decision Trees**: we show that for a single decision tree, robustness verification can be done exactly in linear time. Then we show that for an ensemble of K trees, the verification problem is equivalent to finding the maximum cliques in a K-partite graph, and the graph is in a special form with boxicity equal to the input feature dimension. Therefore, for low-dimensional problems, verification can be done in polynomial time with maximum clique searching algorithms. Finally, for large-scale tree ensembles, we propose a multiscale verification algorithm by exploiting the boxicity of the graph, which can give tight lower bounds on robustness.

2 Background and Related Work

Assume $F : \mathbb{R}^d \to \{1, \ldots, C\}$ is a C-way classification model, given a correctly classified example $\boldsymbol{x}_0$ with $F(\boldsymbol{x}_0) = y_0$, an adversarial perturbation is defined as $\boldsymbol{\delta} \in \mathbb{R}^d$ such that $F(\boldsymbol{x}_0 + \boldsymbol{\delta}) \neq y_0$.

Definition 1 (Robustness Verification Problem). *Given $F, \boldsymbol{x}_0$ and a perturbation radius ϵ, the robustness verification problem aims to determine whether there exists an adversarial example within ϵ ball around $\boldsymbol{x}_0$. In the other word, determine whether the following statement is true:*

$$\exists \boldsymbol{\delta} \text{ s.t. } \|\boldsymbol{\delta}\|_p < \epsilon \text{ and } F(\boldsymbol{x} + \boldsymbol{\delta}) \neq y_0. \tag{1}$$

Exactly solving (1) is usually hard, especially for deep neural networks [11, 20]. **Adversarial attack** algorithms are developed to find an adverarial perturbation $\boldsymbol{\delta}$ that satisfies (1). For example, several widely used attacks have been developed for attacking neural networks [4, 13, 9]. However, adversarial attacks can only find adversarial examples which do not provide a **sound** safety guarantee — even if an attack fails to find an adversarial example, it does not imply no adversarial example exists. Therefore, recent researches have been studied the sound solution to (1) and using them to evaluate safety of a model, leading to the recent developments of robustness verification.

Robustness verification aims to provide a **sound** answer to (1), which means a valid verification algorithm should answer no to (1) only when the existence of adversarial example can be disapproved. A tighter verification algorithm will be able to disaprove (1) for a larger ϵ than looser algorithms. For neural network, it has been shown that solving (1) exactly is NP-complete (for ReLU networks), and thus many recent works have been focusing on developing an efficient and reasonable tight robustness verification algorithm for neural networks [21, 23, 20, 17, 19, 18]. Most of them are following the linear relaxation based approach, where they find linear upper and lower bounds of output neurons with respect to input neurons and then try to answer (1) based on these. However, all of these algorithms are specifically designed for neural networks and cannot be extended to ensemble trees.

Robustness verification for tree ensembles Since ensemble trees are discrete step functions, none of the neural network verification algorithms can be applied. Specialized algorithms is required for verifying tree ensembles. Robustness evaluation and verification is first studied in [10], where they showed that ensemble tree verification is NP-complete when there are multiple trees with depth ≥ 2. An integer programming method was proposed to compute (1) in exponential time. Later on in [2], a single deicion tree is verified for evaluating robustness of an RL policy. A recent work [1] provides a certified defense algorithm for training tree ensembles against ℓ_∞ perturbation, and their algorithm implicitedly use the fact the ℓ_∞ robustness verification for ensemble stumps can be computed efficiently.

3 Robustness Verification for Ensemble Decision Trees and Stumps

3.1 Verification for a single decision tree

We first consider the simplified case with a single decision tree. Assume the decision tree has n nodes and for a given example $\boldsymbol{x}$ with d features, starting from the root, $\boldsymbol{x}$ traverses the decision tree model until reaching a leaf node. Each internal node i determines whether $\boldsymbol{x}$ will be passed to left or right child by checking $\mathbf{I}(x_{t_i} > \eta_i)$, and each leaf node has a value v_i indicating the prediction value of the tree.

If we define B^i as the set of $\boldsymbol{x} \in \mathcal{X}$ that can reach node i, due to the decision tree structure, B^i can be represented as a d-dimensional box:

$$B^i = (l_1^i, r_1^i] \times \cdots \times (l_d^i, r_d^i]. \tag{2}$$

The box can be computed efficiently in linear time by traversing the tree. The detailed algorithm can be found in Appendix A.

We aim to certify whether there exists any misclassified points under perturbation $\|\delta\|_p \leq \epsilon$. We can enumerate boxes for all the leaf nodes and check the minimum distance from $\boldsymbol{x}_0$ to each box. The following proposition shows that the ℓ_p norm distance between a point and a box can be computed in $O(d)$ time, and thus the exact robustness verification problem for a single tree can be solved in $O(dn)$ time.

Proposition 1. *Given a box* $B = (l_1, r_1] \times \cdots \times (l_d, r_d]$ *and a point* $\boldsymbol{x} \in \mathbb{R}^d$*. The closest* ℓ_p *distance from x to B is* $\|z - x\|_p$ *where:*

$$z_i = \begin{cases} x_i, & l_i \le x_i \le u_i \\ l_i, & x_i < l_i \\ u_i, & x_i > u_i. \end{cases} \tag{3}$$

3.2 Ensemble Decision Stumps

We assume there are T decision stumps and the i-th decision stump gives the prediction

$$f^i(x) = \begin{cases} w_l^i & \text{if } x_{t_i} < \eta^i \\ w_r^i & \text{if } x_{t_i} \ge \eta^i. \end{cases}$$

The prediction of decision stump ensemble $F(x) = \sum_i f^i(x)$ can be decomposed into each feature in the following way. For each feature j, assume $j_1, \ldots, j_{T_j}$ are the decision stumps using feature j, we can collect all the thresholds $[\eta^{j_1}, \ldots, \eta^{j_{T_j}}]$. Without loss of generality, assume $\eta^{j_1} \le \cdots \le \eta^{j_{T_j}}$ then the prediction values assigned in each interval can be denoted as

$$g^j(x_j) = v^{j_t} \quad \text{if } \eta^{j_t} < x_j \le \eta^{j_{t+1}} \tag{4}$$

where

$$v^{j_t} = w_l^{j_1} + \cdots + w_l^{j_t} + w_r^{j_{t+1}} + \cdots + w_r^{j_{T_j}}.$$

The overall prediction can be written as summation over the predicted values of each feature:

$$F(x) = \sum_{j=1}^{d} g^j(x_j), \tag{5}$$

and the final prediction is given by $y = sgn(F(x))$.

ℓ_∞ ensemble stump verification Due to the separability of (5), the ℓ_∞ norm perturbation can be done easily in linear time. For each feature j, we just need to check the worst-case perturbation within the range $(x_j - \epsilon, x_j + \epsilon)$ and this can be done by a linear scan through the thresholds $\eta^{j_1}, \ldots, \eta^{j_{T_j}}$. Therefore the verification can be done in polynomial time. This algorithm is implicitly mentioned in [1] for conducting ℓ_∞ certiied defense for tree ensembles.

ℓ_0 ensemble stump verification Assume $F(x)$ is positive and we want to make it the most negative by perturbing δ features (in this case, δ should be an integer). For each feature j, we want to know *the maximum decrease of prediction value by changing this feature,* which can be computed as

$$c^j = \min_t v^{j_t} - g^j(x_j), \tag{6}$$

and we should choose δ features with smallest c^j values to perturb. Let S_δ denotes the set with δ smallest c^j values, we have

$$\min_{\|x-x'\|_0 \le K} F(x') = F(x) + \sum_{i \in S_K} c^j. \tag{7}$$

Therefore verification can be done exactly in $O(T + d)$ time.

ℓ_p ensemble stump verification The difficulty of ℓ_p norm robustness verification is that the perturbations on each feature are correlated, so we can't separate all the features. In the following, we prove that the exact ℓ_p norm verification is NP-complete by showing a reduction from Knapsack to ℓ_p norm ensemble stump verification. This shows that ℓ_p norm verification can belong to a different complexity class compared with the ℓ_∞ norm case. The proof can be found in Appendix B, where we make a connection between ensemble stump verification and Knapsack problem.

Theorem 1. *Exact ℓ_p norm robustness verification (solving eq* (1)*) for an ensemble decision stump is NP-complete when $p \in (0, \infty)$.*

Although it is impossible to solve ℓ_p verification for decision stumps in polynomial time, we show an upper bound of this can be solved in polynomial time by dynamic programming, inspired by the pseudo-polynomial time algorithm for Knapsack.

Let $\eta^{j_1}, \ldots, \eta^{j_{T_j}}$ be the thresholds for feature j and $v^{j_1}, \ldots, v^{j_{T_j}}$ be the corresponding values, our dynamic programming maintains the following value for each ϵ: "given maximal ϵ perturbation to the first j features, what's the minimal prediction of the perturbed x". We denote this value as $D(\epsilon, j)$, then the following recursion holds:

$$D(\epsilon, j+1) = \min_{\delta \in [0, \epsilon]} D(\epsilon - \delta, j) + C(\delta, j+1),$$

where $C(\delta, j+1) := \min_{|x'_j - x_j| < \delta} g^j(x'_j)$ which can be precomputed. Note that δ, ϵ can be real numbers so exactly running this DP requires exponential time. Our approximate algorithm allows ϵ, δ only up to certain precision. If we choose precision ν, then we only consider values $\nu, 2\nu, \ldots, P\nu$ (the smallest P with $P\nu > \epsilon$). To ensure the verification algorithm is sound, the recursion will become

$$D(a\nu, j+1) = \min_{b \in \{1, \ldots, a\}} D((a-b+1)\nu, j) + C(b\nu, j+1), \tag{8}$$

and the final solution should be $D(\lceil \epsilon \rceil, d)$ where $\lceil \epsilon \rceil := T\nu$ means rounding ϵ up to the closest grid. Note that the $+1$ term in the recursion is to ensure that the resulting value is a lower bound of the original solution. The verification algorithm can verify N samples in $O(N(Pd + T))$ time, in which d is dimension and P is the number of discretizations.

3.3 Ensemble Decision Trees: Connection to max clique finding

Now we discuss robustness verification for tree ensembles. Assuming the tree ensemble has K decision trees, we use $S^{(k)}$ to denote the set of leaf nodes of tree k and $m^{(k)}(x)$ to denote the function that maps the input example x to the leaf node of tree k according to

its traversal rule. Given an input example x, the tree ensemble will pass x to each of these K trees independently and x reaches K leaf nodes $i^{(k)} = m^{(k)}(x)$ for all $k = 1, \ldots, K$. Each leaf node will assign a prediction value $v_{i^{(k)}}$. For simplicity we consider the binary classification problem where the original sample is classified as negative and the goal is to find whether there exists an input in the ϵ-ball that will be classified as positive . We will first consider the ℓ_∞ ball verification problem (input region is an ϵ-radius ℓ_∞ ball around x).

We start by defining some notation: let $\mathbb{C} = \{(i^{(1)}, \ldots, i^{(K)}) \mid i^{(k)} \in S^{(k)}, \forall k = 1, \ldots, L\}$ to be all the possible tuples of leaf nodes and let $C(x) = [m^{(1)}(x), \ldots, m^{(K)}(x)]$ be the function that maps x to the corresponding leaf nodes. Therefore, a tuple $C \in \mathbb{C}$ directly determines the model prediction $\sum v_C := \sum_k v_{i^{(k)}}$. Now we define a valid tuple for robustness verification:

Definition 2. *A tuple $C = (i^{(1)}, \ldots, i^{(K)})$ is valid if and only if there exists an $x' \in Ball(x, \epsilon)$ such that $C = C(x')$.*

The robustness verification (1) can then be written as:

$$\textit{Does there exist a valid tuple } C \textit{ such that } \sum v_C > 0?$$

Next, we show how to model the set of valid tuples. We have two observations. First, if a tuple contains any node i with $\inf_{x' \in B^i}\{\|x - x'\|_\infty\} > \epsilon$, then it will be invalid. Second, there exists an x such that $C = C(x)$ if and only if $B^{i^{(1)}} \cap \cdots \cap B^{i^{(K)}} \neq \emptyset$, or equivalently:

$$(l_t^{i^{(1)}}, r_t^{i^{(1)}}] \cap \cdots \cap (l_t^{i^{(K)}}, r_t^{i^{(K)}}] \neq \emptyset, \quad \forall t = 1, \ldots, d.$$

We show that the set of valid tuples can be represented as cliques in a graph $G = (V, E)$, where $V := \{i | B^i \cap \text{Ball}(x, \epsilon) \neq \emptyset\}$ and $E := \{(i, j) | B^i \cap B^j \neq \emptyset\}$. In this graph, nodes are the leaves of all trees and we remove every leaf that has empty intersection with $\text{Ball}(x, \epsilon)$. There is an edge (i, j) between node i and j if and only if their boxes intersect. The graph will then be a K-partite graph since there cannot be any edge between nodes from the same tree, and thus maximum cliques in this graph will have K nodes. We define each part of the K-partite graph as V_k. Here a "part" means a disjoint and independent set in the K-partite graph. The following lemma shows that intersections of boxes have very nice properties:

Lemma 1. *For boxes $B^1, \ldots, B^K$, if $B^i \cap B^j \neq \emptyset$ for all $i, j \in [K]$, let $\bar{B} = B^1 \cap B^2 \cap \cdots \cap B^K$ be their intersection. Then $\bar{B}$ will also be a box and $\bar{B} \neq \emptyset$.*

The proof can be found in the Appendix C. Based on the above lemma, each K-clique (fully connected subgraph with K nodes) in G can be viewed as a set of leaf nodes that has nonempty intersection with each other and also has nonempty intersection with $\text{Ball}(x, \epsilon)$, so the intersection of those K boxes and $\text{Ball}(x, \epsilon)$ will be a nonempty box, which implies each K-clique corresponds to a valid tuple of leaf nodes:

Lemma 2. *A tuple $C = (i^{(1)}, \ldots, i^{(K)})$ is valid if and only if nodes $i^{(1)}, \ldots, i^{(K)}$ form a K-clique (maximum clique) in graph G constructed above.*

Therefore the robustness verification problem can be formulated as

$$\textit{Is there a maximum clique } C \textit{ in } G \textit{ such that } \sum v_C > 0? \tag{9}$$

This reformulation indicates that the tree ensemble verification problem can be solved by an efficient maximum clique enumeration algorithm. Some standard maximum clique searching algorithms can thus be applied here to perform verification:

- **Finding K-cliques in K-partite graphs:** Any algorithm for finding all the maximum cliques in G can be used. The classic B-K backtracking algorithm [3] takes $O(3^{\frac{m}{3}})$ time to find all the maximum cliques where m is the number of nodes in G. Furthermore, since our graph is a K-partite graph, we can apply some specialized algorithms designed for finding all the K-cliques in K-partite graphs [14, 15, 16].
- **Polynomial time algorithms exist for low-dimensional problems:** Another important property for graph G is that each node in G is a d-dimensional box and each edge indicates the intersection of two boxes. This implies our graph G is with "boxicity d" (see [5] for detail). [5] proved that the number of maximum cliques will only be $O((2m)^d)$ and it is able to find the maximum weight clique in $O((2m)^d)$ time. Therefore, for problems with a very small d, the time complexity for verification is actually polynomial.

3.4 An Efficient and Sound Verification Algorithm for Tree Ensemble

Practical tree ensembles usually have tens or hundreds of trees with large feature dimensions, so exact clique findings will take exponential time and will be too slow. We thus develop an efficient multi-level algorithm for computing verification bounds by further exploiting the boxicity of the graph.

Figure 1 illustrates the graph and how our multilevel algorithm runs. There are four trees and each tree has four leaf nodes. A node is colored if it has nonempty intersection with $\text{Ball}(x, \epsilon)$; uncolored nodes are discarded. To answer question (9), we need to compute the maximum $\sum v_C$ among all K-cliques, denoted by v^*. As mentioned before, for robustness verification we only need to compute an upper bound of v^* in order to get a lower bound of minimal adversarial perturbation. In the following, we will first discuss algorithms for computing an upper bound at the top level, and then show how our multi-scale algorithm iteratively refines this bound until reaching the exact solution v^*.

Bounds for a single level. To compute an upper bound of v^*, a naive approach is to assume that the graph is fully connected between independent sets (fully connected K-partite graph) and in this case the maximum sum of node values is the sum of the maximum value of each independent set:

$$\sum\nolimits_{k=1}^{|\tilde{V}|} \max_{i \in V_k} v_i \geq v^*. \tag{10}$$

Here we abuse the notation v_i by assuming that each node i in V_k has been assigned a "pseudo prediction value", which will be used in the multi-level setting. In the simplest case, each independent set represents a single tree, $V_k = S^{(k)}$ and v_i is the prediction of a leaf. One can easily show this is an upper bound of v^* since any K-clique in the graph

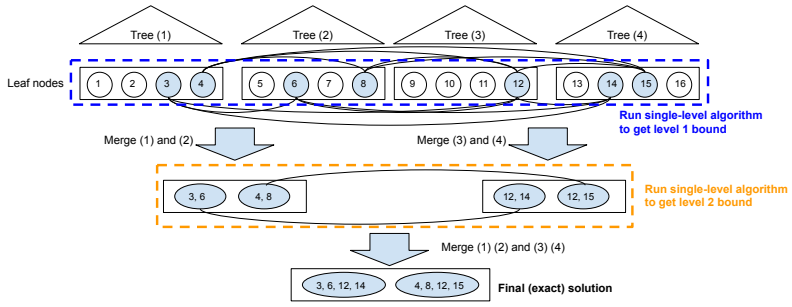


Fig. 1: The proposed multi-level verification algorithm. Lines between leaf node i on tree t_1 and leaf node j on t_2 indicate that their ℓ_∞ feature boxes intersect (i.e., there exists an input such that tree 1 predicts v_i and tree 2 predicts v_j).

is still considered when we add more edges to the graph, and eventually it becomes a fully connected K-partite graph.

Another slightly better approach is to exploit the edge information but only between tree t and $t+1$. If we search over all the length-K paths $[i^{(1)}, \ldots, i^{(K)}]$ from the first to the last part and define the value of a path to be $\sum_k v_{i^{(k)}}$, then the maximum valued path will be a upper bound of v^*. This can be computed in linear time using dynamic programming. We scan nodes from tree 1 to tree K, and for each node we store a value d_i which is the maximum value of paths from tree 1 to this node. At tree k and node i, the d_i value can be computed by

$$d_i = v_i + \max_{j: j \in V_{k-1} \text{ and } (j,i) \in E} d_j. \tag{11}$$

Then we take the max d value in the last tree. It produces an upper bound of v^*, since the maximum valued path found by dynamic programming is not necessarily a K-clique. Again $V_{k-1} = S^{(k-1)}$ in the first level but it will be generalized below.

Merging T independent sets To refine the relatively loose single-level bound, we partition the graph into K/T subgraphs, each with T independent sets. Within each subgraph, we find all the T-cliques and use a new "pseudo node" to represent each T-clique. T-cliques in a subgraph can be enumerated efficiently if we choose T to be a relatively small number (e.g., 2 or 3 in the experiments).

Now we exploit the boxicity property to form a new graph among these T-cliques (illustrated as the second level nodes in Figure 1). By Lemma 1, we know that the intersection of T boxes will still be a box, so each T-clique is still a box and can be represented as a pseudo node in the level-2 graph. Also because each pseudo node is still a box, we can easily form edges between pseudo nodes to indicate the nonempty overlapping between them and this will be a (K/T)-partite boxicity graph since no edge can be formed for the cliques within the same subgraph. Thus we get the level-2 graph. With the level-2 graph, we can again run the single level algorithm to compute a upper bound on v^* to get a lower bound of r^* in (1), but different from the level-1 graph, now we already considered all the within-subgraph edges so the bounds we get will be tighter.

The overall multi-level framework We can run the algorithm level by level until merging all the subgraphs into one, and in the final level the pseudo nodes will correspond to the K-cliques in the original graph, and the maximum value will be exactly v^*. Therefore, our algorithm can be viewed as an anytime algorithm that refines the upper bound level-by-level until reaching the maximum value. Although getting to the final level still requires exponential time, in practice we can stop at any level (denoted as L) and get a reasonable bound. In experiments, we will show that by merging few trees we already get a bound very close to the final solution. Algorithm 1 gives the complete procedure.

Algorithm 1: Multi-level verification framework

input : The set of leaf nodes of each tree, $S^{(1)}, S^{(2)}, \ldots, S^{(K)}$; maximum number of independent sets in a subgraph (denoted as T); maximum number of levels (denoted as L), $L \leq \lceil \log_T(K) \rceil$;

for $k \leftarrow 1, 2, \ldots, K$ **do**
  $U_k^{(0)} \leftarrow \{(A_i, B^{i^{(k)}}) | i^{(k)} \in S^{(k)}, A_i = \{i^{(k)}\}\}$;
  /* U is defined the same as in Algorithm **??**. At level 0, each V_k forms a 1-clique by itself. */
end
for $l \leftarrow 1, 2, \ldots, L$ **do**
  /* Enumerate all cliques in each subgraph at this level. Total $\lceil K/T^l \rceil$ subgraphs. */
  for $k \leftarrow 1, 2, \ldots, \lceil K/T^l \rceil$ **do**
    $U_k^{(l)} \leftarrow U_{(k-1)T+1}^{(l-1)}, U_{(k-1)T+2}^{(l-1)}, \ldots, U_{kT}^{(l-1)}$;
  end
end
for $k \leftarrow 1, 2, \ldots, \lceil K/T^L \rceil$ **do**
  /* Define an independent set V'_k for each $U_k^{(L)}$. In each V'_k, we create ''pseudo nodes'' which combines multiple nodes from lower levels, and assign ''pseudo prediction values'' to them. */
  $V'_k \leftarrow \{A \big| (A, B) \in U_k^{(L)}\}$; /* V'_k is a set of sets; each element in V'_k represents a clique. */
  /* Construct the ''pseudo prediction value'' for each element in V'_k by summing up all prediction values in the corresponding clique. */
  For all $A \in V'_k$, $v_A \leftarrow \sum_{i \in A} v_i$
end
$\bar{v} \leftarrow$ an upper bound of v^* using (10) or (11), given $\tilde{V} = \{V'_1, \cdots, V'_{\lceil K/T^L \rceil}\}$;
/* If $\lceil K/T^L \rceil = 1$, only 1 independent set left and each pseudo node represents a K-clique; (10) or (11) will have a trivial solution where v^* is the maximum v_A in $U_1^{(L)}$. */

Handling multi-class tree ensembles. For a multiclass classification problem, say a C-class classification problem, C groups of tree ensembles (each with K trees) are built for the classification task; for the k-th tree in group c, prediction outcome is denoted as $i^{(k,c)} = m^{(k,c)}(x)$ where $m^{(k,c)}(x)$ is the function that maps the input example x to a leaf node of tree k in group c. The final prediction is given by $\arg\max_c \sum_k v_{i^{(k,c)}}$. Given an input example x with ground-truth class c and an attack target class c', we extract $2K$ trees for class c and class c', and flip the sign of all prediction values for trees in group c', such that initially $\sum_t v_{i^{(t,c)}} + \sum_t v_{i^{(t,c')}} < 0$ for a correctly classified example. Then, we are back to the binary classification case with $2K$ trees, and we can still apply our multi-level framework to obtain a lower bound $\underline{r}_{(c,c')}$ of $r^*_{(c,c')}$ for this target attack pair (c, c'). Robustness of an untargeted attack can be evaluated by taking $\underline{r} = \min_{c' \neq c} \underline{r}_{(c,c')}$.

Dataset		ℓ_1 MILP		Ours ℓ_1 DP approx.			Ours vs. MILP		Ours ℓ_0 verification		
name	ϵ	robust err.	avg. time	precision	robust err.	avg. time	MILP/ours	speedup	avg. robust r^*	robust acc.	avg. time
breast-cancer	0.3	10.94%	.030s	0.01	10.94%	.00025s	1.00	120X	.04	95.62%	.0006
diabetes	0.05	35.06%	.017s	0.0002	35.06%	.0004s	1.00	40X	.0	100%	.0005s
Fashion-MNIST shoes	0.1	10.45%	.105s	0.005	10.55%	.0013s	.99	80.8X	2.09	16.35%	.010s
MNIST 1 vs. 5	0.3	3.30%	0.11s	0.005	3.35%	0.0013s	1.00	71X	3.33	3.50%	.010s
MNIST 2 vs. 6	0.3	9.64%	0.099s	0.005	9.69%	.0012s	.98	82X	1.22	26.43%	.012s

Table 1: **General ℓ_p-norm ensemble stump verification.** This table reports robust test error (robust err.) and average per sample time consumption (avg. time) of each method. For our proposed DP based verification, precision is also reported. For ℓ_0 verification, we also report average robust radius r^*, which means averagely how many features can be perturbed at most when the prediction stays the same.

Handling ℓ_p norm verification for $p < \infty$. In the ℓ_p norm case when $p < \infty$, the elimination step will lead to incorrect answer. Let $Ball_p(x, \epsilon)$ be the ℓ_p-norm ball with radius ϵ around x. For boxes $(B^1, \ldots, B^T)$, even if $B^i \cap B^j \neq \emptyset$ for all i, j and $B^i \cap Ball_p(x, \epsilon) \neq \emptyset$ for all i, it is not guaranteed that $B^1 \cap B^1 \ldots \cap B^T \cap Ball_p(x, \epsilon) \neq \emptyset$. Here we can generalize the framework to ℓ_p cases. In each layer from 1 to L, we split the T trees into groups of K. We find the K-size cliques in each group, which are non-intersected boxes, and form a group of new virtual nodes. After that, we keep the cliques which have nonempty intersection with $Ball_p(x, \epsilon)$ in the group. This group can then be treated as a virtual tree at the next level. This gives us an efficient algorithm for ℓ_p robustness verification.

4 Experimental Results

The results on real datasets demonstrate the proposed verification algorithms can compute a reasonably tight bound while being able to scale to large datasets. The statistics of the data sets are shown in Appendix D.

Robustness Verification for Ensemble Stumps. As discussed in the paper, we show ℓ_p norm verification has polynomial time algorithm only when $p = 0, \infty$. We thus pick $p = 0$ to demonstrate the algorithm works exactly and $p = 1$ to demonstrate our approximate verification algorithm can output reasonably tight bounds. Ensembles that are verified are trained with ℓ_∞ training proposed in [1].

For the ℓ_1 norm robustness verification problem, we have shown it's NP-complete to conduct exact verification. To demonstrate the tightness and efficiency of the proposed Dynamic Programming (DP) based verification, we also run the Mixed Integer Linear Programming [10] to get the exact robust bound which takes exponential time. In Table 1, we can find that the proposed DP algorithm gives almost exactly the same bound with MILP, while being $50 - 100$ times faster. This speedup guarantees its further applications in certified robust training. For the ℓ_0 norm robustness verification problem, we propose a linear time algorithm for conducting exact robustness verification. The results are also reported in Table 1. We can observe that the proposed method can conduct exact verification in less than 0.1 second.

Robustness Verification for Ensemble Trees. We evaluate our approximate ℓ_p verification method for tree ensembles on five real datasets. Ensembles that are verified are trained with ℓ_∞ training proposed in [1], each of which contains 20 trees. Again, we compare the proposed algorithm with MILP-based verification [10] which takes

Dataset		ℓ_1 MILP		Ours ℓ_1 approx.				Ours vs. MILP	
name	ϵ	robust err.	avg. time	K	L	robust err.	avg. time	robust err.	speedup
breast-cancer	0.3	8.03%	.036s	3	2	8.03%	.012s	1.00	3X
diabetes	0.05	33.12%	.027s	3	2	33.12%	.012s	1.00	2.25X
Fashion-MNIST shoes	0.1	10%	.091s	3	2	10%	.011s	1.00	8.23X
MNIST 1 vs. 5	0.3	4.20%	0.088s	3	2	4.20%	.011s	1.00	8X
MNIST 2 vs. 6	0.3	8.60%	.098s	3	2	8.80%	.012s	.98	8.17X

Table 2: **General ℓ_p-norm tree ensemble verification.** This table reports robust test error (robust err.) and average per sample time consumption (avg. time) of each method. For our generalized verification framework, K: size of cliques, and L: layer of the framework are also reported.

Dataset	MILP [10]		LP relaxation		Ours				Ours vs. MILP	
	avg. r^*	avg. time	avg. r_{LP}	avg. time	T	L	avg. r_{our}	avg. time	r_{our}/r^*	speedup
breast-cancer	.210	.012s	.064	.009s	2	1	.208	.001s	.99	12X
diabetes	.049	.061s	.015	.026s	3	2	.042	.018s	.86	3.4X
Fashion-MNIST	.014*	1150*s	.003*	898*s	2	1	.012	11.8s	.86	97X
MNIST	.011*	367*s	.003*	332*s	2	2	.011	5.14s	1.00	71X
MNIST 2 vs. 6	.057	23.0s	.016	11.6s	4	1	.046	.585s	.81	39X

Table 3: Average ℓ_∞ distortion over 500 examples and average verification time per example for three verification methods. Here we evaluate the bounds for **standard (natural) GBDT models**. Results marked with a start ("$\star$") are the averages of 50 examples due to long running time. T is the number of independent sets and L is the number of levels in searching cliques used in our algorithm. A ratio $\underline{r}_{our}/r^*$ close to 1 indicates better lower bound quality.

exponential time to get the exact bound. The results are presented in Table 2, and parameters of the proposed method (K and L) are also reported. We observe that the proposed verification method gets very tight robust error while being much faster than the MILP solver.

Note that as described in the previous section, the ℓ_∞ norm tree verification using our algorithm can be more efficient than a general ℓ_p norm. Here we then show the results on the ℓ_∞ norm verification in Table 3. Note that for this experiment we are trying to verify naturally trained tree ensembles by XGBoost. And instead of computing the robust error for a particular ϵ, we further conduct binary search for each sample to find the minimum ℓ_∞ norm adversarial perturbation, denoted as $\bar{r}_{\text{ours}}$, and compared with the optimal value by MILP, denoted by r^*. Furthermore, to show that directly relaxing MILP to Linear Programming (LP) won't give a tight lower bound, we also report its value $\bar{r}_{\text{LP}}$ in Table 3. The results show that our method can get a reasonably tight lower bound of MILP solution efficiently and much better than the LP relaxation.

5 Conclusion

In this paper, we study the robustness verification problem for ensemble stumps and trees. For both cases, we conduct a careful analysis of the computational complexity and propose efficient approximation algorithms to compute a sound robustness verification bound when the problem is NP-complete. Experimental results on real datasets demonstrate the efficiency and tightness of the proposed methods.

Bibliography

[1] M. Andriushchenko and M. Hein. Provably robust boosted decision stumps and trees against adversarial attacks. In *NeurIPS*, 2019.

[2] O. Bastani, Y. Pu, and A. Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pages 2494–2504, 2018.

[3] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[4] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[5] L. S. Chandran, M. C. Francis, and N. Sivadasan. Geometric representation of graphs in low dimension using axis parallel boxes. *Algorithmica*, 56(2):129, 2010.

[6] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh. Robust decision trees against adversarial examples. In *ICML*, 2019.

[7] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[8] M. Cheng, T. Le, P.-Y. Chen, J. Yi, H. Zhang, and C.-J. Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. In *ICLR*, 2019.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[10] A. Kantchelian, J. Tygar, and A. Joseph. Evasion and hardening of tree ensemble classifiers. In *ICML*, 2016.

[11] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[12] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[13] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[14] M. Mirghorbani and P. Krokhmal. On finding k-cliques in k-partite graphs. *Optimization Letters*, 7(6):1155–1165, 2013.

[15] C. A. Phillips, K. Wang, E. J. Baker, J. A. Bubier, E. J. Chesler, and M. A. Langston. On finding and enumerating maximal and maximum k-partite cliques in k-partite graphs. *Algorithms*, 12(1):23, 2019.

[16] M. Schneider and B. Wulfhorst. Cliques in k-partite graphs and their application in textile engineering. 2002.

[17] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. In *NIPS*, 2018.

[18] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL): 41, 2019.

[19] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *NIPS*, 2018.
[20] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel. Towards fast computation of certified robustness for relu networks. In *ICML*, 2018.
[21] E. Wong and J. Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *ICML*, 2018.
[22] H. Zhang, S. Si, and C.-J. Hsieh. GPU-acceleration for large-scale tree boosting. *SysML Conference*, 2018.
[23] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions. In *NIPS*, 2018.

A Algorithm for computing the box for each leaf

Conceptually, the main idea of our single tree verification algorithm is to compute a d-dimensional box for each leaf node such that any example in this box will fall into this leaf. Mathematically, the node i's box is defined as the Cartesian product $B^i = (l_1^i, r_1^i] \times \cdots \times (l_d^i, r_d^i]$ of d intervals on the real line. By definition, the root node has box $[-\infty, \infty] \times \cdots \times [-\infty, \infty]$ and given the box of an internal node i, its children's boxes can be obtained by changing only one interval of the box based on the split condition (t_i, η_i). More specifically, if p, q are node i's left and right child node respectively, then we set their boxes $B^p = (l_1^p, r_1^p] \times \cdots \times (l_d^p, r_d^p]$ and $B^q = (l_1^q, r_1^q] \times \cdots \times (l_d^q, r_d^q]$ by setting

$$(l_t^p, r_t^p] = \begin{cases} (l_t^i, r_t^i] & \text{if } t \neq t_i \\ (l_t^i, \min\{r_t^i, \eta_i\}] & \text{if } t = t_i \end{cases}, \quad (l_t^q, r_t^q] = \begin{cases} (l_t^i, r_t^i] & \text{if } t \neq t_i \\ (\max\{l_t^i, \eta_i\}, r_t^i] & \text{if } t = t_i. \end{cases} \tag{12}$$

After computing the boxes for internal nodes, we can also obtain the boxes for leaf nodes using (12). Therefore computing the boxes for all the leaf nodes of a decision tree can be done by a depth-first search traversal of the tree with time complexity $O(nd)$.

B Proof of Theorem 1

Proof. We show that a 0-1 Knapsack problem can be reduced to an ensemble stump verification problem. A 0-1 Knapsack problem can be defined as follows. Assume there are T items each with weight w_i and value v_i, the (decision version of) 0-1 Knapsack problem aims to determine whether there exists a subset of items S such that $\sum_{i \in S} w_i \leq C$ and with value $\sum_{i \in S} v_i \geq D$.

We construct a decision stump verification problem with T features and T stumps, where each decision stump corresponds to one feature. Assume x is the original example, we define each decision stump to be

$$g^i(s) = -v_i I(s > \eta_i) + \frac{D}{d}, \quad \text{where } \eta_i = x_i + w_i^{(1/p)}, \tag{13}$$

where $I()$ is the indicator function. The goal is to verify ℓ_p robustness with $\epsilon = C^{(1/p)}$. We need to show that this robustness verification problem outputs YES ($\min_{\|x-x'\|_p \leq \epsilon} \sum_i g^i(x'_i) < 0$) if and only if the Knapsack solution is also YES. If the verification found $v^* = \min_{\|x-x'\|_p \leq \epsilon} \sum_i g^i(x'_i) < 0$, let x' be the corresponding solution of verification, then we can choose the following S for 0-1 Knapsack:

$$S = \{i \mid x'_i > \eta_i\} \tag{14}$$

It is guaranteed that

$$\sum_{i \in S} w_i = \sum_{i \in S} |\eta_i - x_i|^p \leq \sum_i |x'_i - x_i|^p \leq \epsilon^p = C \tag{15}$$

and by the definition of g^i we have $\sum_i g^i(x'_i) = D - \sum_{i \in S} v_i \leq 0$, so this subset S will also be feasible for the Knapsack problem. On the other hand, if the 0-1 Knapsack problem has a solution S, for robustness verification problem we can choose x' such that

$$x'_i = \begin{cases} \eta_i & \text{if } i \in S \\ x_i & \text{otherwise} \end{cases}$$

By definition we have $\sum_i g^i(x'_i) = D - \sum_{i \in S} v_i < 0$. Therefore the Knapsack problem, which is NP-complete, can be reduced to ℓ_p norm decision stump verification problem with any $p \in (0, \infty)$ in polynomial time.

C Proof of Lemma 1

Proof. If we have K one dimensional intervals $I_1 = (l_1, r_1], I_2 = (l_2, r_2], \ldots, I_T = (l_K, r_K]$, we want to prove if every pair of them have nonempty overlap $I_1 \cap \cdots \cap I_K \neq \emptyset$. This can be proved by the following. Without loss of generality we assume $l_1 \leq l_2 \leq \cdots \leq l_K$. For each $k < K$, $I_k \cap I_K \neq \emptyset$ implies $l_K < r_k$. Therefore, $(l_T, \min(r_1, r_2, \ldots, r_K)]$ will be a nonempty set that is contained in $I_1, I_2, \ldots, I_K$. Therefore $I_1 \cap I_2 \cap \cdots \cap I_K \neq \emptyset$ and it is another interval.

This can be generalized to d-dimensional boxes. Assume we have boxes $B_1, \ldots, B_K$ such that $B_i \cap B_j \neq \emptyset$ for any i and j. Then for each dimension we can apply the above proof, which implies that $B_1 \cap B_2 \cap \cdots \cap B_K \neq \emptyset$ and the intersection will be another box.

D Data Statistics and Model Parameters

Table 4 presents data statistics and parameters for the models in the main text. The standard test accuracy is the model accuracy on natural, unmodified test sets.

Dataset	training set size	test set size	# of features	# of classes	# of trees	robust ϵ	depth robust	depth natural	standard test acc. robust	standard test acc. natural
breast-cancer	546	137	10	2	4	0.3	8	6	.978	.964
diabetes	614	154	8	2	20	0.2	5	5	.786	.773
Fashion-MNIST	60,000	10,000	784	10	200	0.1	8	8	.903	.903
MNIST	60,000	10,000	784	10	200	0.3	8	8	.980	.980
MNIST 2 vs. 6	11,876	1,990	784	2	1000	0.3	6	4	.997	.998

Table 4: The data statistics and parameters for the models presented in this paper.

On Symbolically Encoding the Behavior of Random Forests

Arthur Choi[1], Andy Shih[2], Anchal Goyanka[1], and Adnan Darwiche[1]

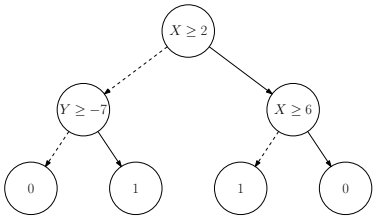
[1] Computer Science Department, UCLA
{aychoi, anchal, darwiche}@cs.ucla.edu
[2] Computer Science Department, Stanford University
andyshih@cs.stanford.edu

Abstract. Recent work has shown that the input-output behavior of some machine learning systems can be captured symbolically using Boolean expressions or tractable Boolean circuits, which facilitates reasoning about the behavior of these systems. While most of the focus has been on systems with Boolean inputs and outputs, we address systems with discrete inputs and outputs, including ones with discretized continuous variables as in systems based on decision trees. We also focus on the suitability of encodings for computing prime implicants, which have recently played a central role in explaining the decisions of machine learning systems. We show some key distinctions with encodings for satisfiability, and propose an encoding that is sound and complete for the given task.

Keywords: Explainable AI · Random Forests · Prime Implicants.

1 Introduction

Recent work has shown that the input-output behavior of some machine learning systems can be captured symbolically using Boolean expressions or tractable Boolean circuits [10, 12, 16, 25, 7, 8, 26, 23]. These encodings facilitate the reasoning about the behavior of these systems, including the explanation of their decisions, the quantification of their robustness and the verification of their properties. Most of the focus has been on systems with Boolean inputs and outputs, with some extensions to discrete inputs and outputs, including discretizations of continuous variables as in systems based on decision trees; see, e.g., [2, 15, 9]. This paper is concerned with the latter case of discrete/continuous systems but those that are encoded using Boolean variables, with the aim of utilizing the vast machinery available for reasoning with Boolean logic. Most prior studies of Boolean encodings have focused on the tasks of satisfiability and model counting [11, 27, 2]. In this paper, we focus instead on prime implicants which have recently played a central role in explaining the decisions of machine learning systems [25, 20, 7–9, 5]; cf. [21]. We first highlight how the prime implicants of a multi-valued expression are not immediately obtainable as prime implicants of a corresponding Boolean encoding. We reveal how to compute these prime implicants, by computing them instead on a Boolean expression derived from



value	interval
x_1	$(-\infty, 2)$
x_2	$[2, 6)$
x_3	$[6, +\infty)$
y_1	$(-\infty, -7)$
y_2	$[-7, +\infty)$

X	Y	$f(X,Y)$
x_1	y_1	0
x_1	y_2	1
x_2	y_1	1
x_2	y_2	1
x_3	y_1	0
x_3	y_2	0

Fig. 1. (Left) A decision tree of continuous variables X and Y, where a solid branch means the test is true, and a dashed branch means false. (Center) A discretization of X and Y into intervals. (Right) The discrete function represented by the decision tree.

the encoding. Our study is conducted in the context of encoding the behavior of random forests using majority voting, but our results apply more broadly.

This paper is structured as follows. We introduce the task in Section 2 as well as review related work. We discuss in Section 3 the problem of explaining the decisions of machine learning systems whose continuous features can be discretized into intervals. We follow in Section 4 by a discussion on encoding the input-output behavior of such systems, where we analyze three encodings from the viewpoint of computing explanations for decisions. We show that one of these encodings is suitable for this purpose, if employed carefully, while proving its soundness and completeness for the given task. We finally close in Section 5.

2 Boolean, Discrete and Continuous Behaviors

The simplest behaviors to encode are for systems with Boolean inputs and outputs. Consider a neural network whose inputs are Boolean and that has only step activation functions. Each neuron in this network computes a Boolean function and therefore each output of the network also computes a Boolean function. The input-output behavior of such networks can be immediately represented using Boolean expressions, or Boolean circuits as proposed in [3, 23].

Suppose now that the inputs to a machine learning system are discrete variables, say, variable A with values $1, 2, 3$, variable B with values r, b, g and variable C with values l, m, h. One can define a multi-valued propositional logic to capture the behavior of such a system. The atomic expressions in this case will be of the form $V{=}v$, indicating that discrete variable V has the value v. We can then construct more complex expressions using Boolean connectives. An example expression in this logic would be $(B{=}r \vee B{=}b) \implies (A{=}2 \wedge \neg C{=}h)$.

Some systems may have continuous variables as inputs, which get discretized during the learning process as is the case with systems based on decision trees. Consider for example the decision tree in Figure 1 (left) over continuous variables X and Y. The algorithm that learned this tree discretized its variables as follows: X to intervals $(-\infty, 2), [2, 6), [6, +\infty)$ and Y to intervals $(-\infty, -7), [-7, +\infty)$.

We can now think of variable X as a discrete variable with three values x_1, x_2, x_3, each corresponding to one of the intervals as shown in Figure 1

(center). Variable Y is binary in this case, with each value corresponding to one of the two intervals. According to this decision tree, the infinite number of input values for variables X and Y can be grouped into six equivalence classes as shown in Figure 1 (right). Hence, the input-output behavior of this decision tree can be captured using the multi-valued propositional expression $f(X, Y) = (X{=}x_1 \wedge Y{=}y_2) \vee X{=}x_2$, even though we have continuous variables.

Our goal is therefore to encode multi-valued expressions using Boolean expressions as we aim to exploit the vast machinery currently available for reasoning with propositional logic. This includes SAT-based and knowledge compilation tools, which have been used extensively recently to reason about the behavior of machine learning systems [10, 12, 16, 25, 7, 8, 26, 23].

Encoding multi-valued expressions using Boolean expressions has been of interest for a very long time and several methods have been proposed for this purpose; see, e.g., [11, 27, 2]. In some cases, different encodings have been compared in terms of the efficacy of applied SAT-based tools; see, e.g., [27]. In this paper, we consider another dimension for evaluating encodings, which is based on their suitability for computing prime implicants. This is motivated by the fundamental role that implicants have been playing recently in explaining the decisions of machine learning systems [25, 20, 7–9, 5]

The previous works use the notion of a *PI-explanation* when explaining the decision of a classifier on an instance. A PI-explanation, introduced in [25], is a minimal set of instance characteristics that are sufficient to trigger the decision. That is, if these characteristics are fixed, other instance characteristics can be changed freely without changing the decision. In an image, for example, a PI-explanation corresponds to a minimal set of pixels that guarantees the stability of a decision against any perturbation of the remaining pixels.[3]

PI-explanations are based on *prime implicants* of Boolean functions, which have been studied extensively in the literature [4, 17, 13, 18]. Consider the following Boolean function over variables A, B and C: $f = (A + \overline{C})(B + C)(A + B)$. A prime implicant of the function is a minimal setting of its variables that causes the function to trigger. This function has three prime implicants: AB, AC and $B\overline{C}$. Consider now the instance $AB\overline{C}$ leading to a positive decision $f(AB\overline{C}) = 1$. The PI-explanations for this decision are the prime implicants of function f that are compatible with the instance: AB and $B\overline{C}$. Explaining negative decisions requires working with the function's complement $\overline{f}$. Consider instance $\overline{A}BC$, which sets the function f to 0. The complement $\overline{f}$ has three prime implicants $\overline{A}C$, $\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}$. Only one of these is compatible with the instance, $\overline{A}C$, so it is the only PI-explanation for the decision on this instance.[4]

[3] A PI-explanation can be viewed as a (minimally) *sufficient reason* for the decision [5].

[4] The popular Anchor system [22] can be viewed as computing approximations of PI-explanations. The quality of these approximations has been evaluated on some datasets and corresponding classifiers in [9], where an approximation is called *optimistic* if it is a strict subset of a PI-explanation and *pessimistic* if it is a strict superset of a PI-explanation. Anchor computes approximate explanations without having to abstract the machine learning system behavior into a symbolic representation.

When considering the encoding of multi-valued expressions using Boolean ones, we will be focusing on whether the prime implicants of multi-valued expressions can be soundly and completely obtained from the prime implicants of the corresponding Boolean expressions. This is motivated by the desire to exploit existing algorithms and tools for computing prime implicants of Boolean expressions (one may also consider developing a new set of algorithms and tools for operating directly on multi-valued expressions).

Before we propose and evaluate some encodings, we need to first define the notion of a prime implicant for multi-valued expressions and then examine explanations from that perspective. This is needed to settle the semantics of explanations in a multi-valued setting, which will then form the basis for deciding whether a particular encoding is satisfactory from the viewpoint of computing explanations. As the following discussion will reveal, the multi-valued setting leads to some new considerations that are preempted in a Boolean setting.

3 Explaining Decisions in a Multi-Valued Setting

Consider again the decision tree in Figure 1 whose behavior is captured by the multi-valued expression $(X{=}x_1 \wedge Y{=}y_2) \vee X{=}x_2$ as discussed earlier. Consider also the positive instance $X{=}3 \wedge Y{=}12$, which can be represented using the multi-valued expression $\alpha : X{=}x_2 \wedge Y{=}y_2$ as shown in Figure 1.

Instance α has two characteristics $X{=}x_2$ and $Y{=}y_2$, yet one of them $X{=}x_2$ is sufficient to trigger the positive decision. Hence, one explanation for the decision is that variable X takes a value in the interval $[2, 6)$, which justifies $X{=}x_2$ as a PI-explanation of this positive decision. In fact, if we stick to the literal definition of a PI-explanation from the Boolean setting, then this would be the only PI-explanation since $Y{=}y_2$ is the only characteristic that can be dropped from the instance while guaranteeing that the decision will stick.

Looking closer, this decision would also stick if the value of X were contained in the larger interval $(-\infty, 6)$ as long as characteristic $Y{=}y_2$ is maintained. The interval $(-\infty, 6)$ corresponds to $(X{=}x_1 \vee X{=}x_2)$, leading to the expression $(X{=}x_1 \vee X{=}x_2) \wedge Y{=}y_2$. This expression is the result of *weakening* literal $X{=}x_2$ in instance $X{=}x_2 \wedge Y{=}y_2$. It can be viewed as a candidate explanation of the decision on this instance, just like $X{=}x_2$, in the sense that it also represents an abstraction of the instance that preserves the corresponding decision.

For another example, consider the negative decision on instance $X{=}10 \wedge Y{=}{-}20$, and its corresponding multi-valued expression $\alpha : X{=}x_3 \wedge Y{=}y_1$. Recall that x_3 represents the interval $[6, +\infty)$ and y_1 represents the interval $(-\infty, -7)$. We can drop the characteristic $Y{=}y_1$ from this instance while guaranteeing that the negative decision will stick (i.e., regardless of what value variable Y takes). Hence, $X{=}x_3$ is a PI-explanation in this case. But again, if we maintain the characteristic $Y{=}y_1$, then this negative decision will stick as long as the value of X is in the larger, disconnected interval $(-\infty, 2] \cup [6, +\infty)$. This interval is represented by the expression $X{=}x_1 \vee X{=}x_3$ which is a weakening of characteristic $X{=}x_3$. This makes $(X{=}x_1 \vee X{=}x_3) \wedge Y{=}y_1$ a candidate explanation as well.

3.1 Multi-Valued Literals, Terms and Implicants

We will now formalize some notions on multi-valued variables and then use them to formally define PI-explanations in a multi-valued setting [19, 14]. We use three multi-valued variables for our running examples: Variable A with values $1, 2, 3$, variable B with values r, b, g and variable C with values l, m, h.

A *literal* is a non-trivial propositional expression that mentions a single variable. The following are literals: $B{=}r \vee B{=}b$, $A{=}2$ and $C{\neq}h$. The following are not literals as they are trivial: $B{=}r \vee B{=}b \vee B{=}g$ and $C{=}h \wedge C{\neq}h$. Intuitively, for a variable with n values, a literal specifies a set of values S where the cardinality of set S is in $\{1, \ldots, n-1\}$. A literal is *simple* if it specifies a single value (cardinality of set S is 1). When multi-valued variables correspond to the discretization of continuous variables, our treatment allows a literal to specify non-contiguous intervals of a continuous variable.

Consider two literals ℓ_i and ℓ_j for the same variable. We say ℓ_i is *stronger* than ℓ_j iff $\ell_i \models \ell_j$ and $\ell_i \not\equiv \ell_j$. In this case, ℓ_j is *weaker* than ℓ_i. For example, $B{=}r$ is stronger than $B{=}r \vee B{=}b$. It is possible to have two literals where neither is stronger or weaker than the other (e.g., $B{=}r \vee B{=}b$ and $B{=}g$).

A *term* is a conjunction of literals over distinct variables. The following is a term: $A{=}2 \wedge (B{=}r \vee B{=}b) \wedge C{\neq}h$. A term is *simple* if all of its literals are simple. The following term is simple: $A{=}2 \wedge B{=}r \wedge C{=}h$. The following terms are not simple: $A{\neq}2 \wedge B{=}r \wedge C{=}h$ and $A{=}2 \wedge (B{=}r \vee B{=}b) \wedge C{=}h$. A simple term that mentions every variable is called an *instance.*

Term τ_i *subsumes* term τ_j iff $\tau_j \models \tau_i$. If we also have $\tau_i \not\equiv \tau_j$, then τ_i *strictly subsumes* τ_j. For example, the term $A{=}2 \wedge (B{=}r \vee B{=}b) \wedge C{\neq}h$ is strictly subsumed by the terms $A{\neq}1 \wedge (B{=}r \vee B{=}b) \wedge C{\neq}h$ and $A{=}2 \wedge C{\neq}h$.

We stress two points now. First, if term τ_i strictly subsumes term τ_j that does not necessarily mean that τ_i mentions a fewer number of variables than τ_j. In fact, it is possible that the literals of τ_i and τ_j are over the same set of variables. Second, a term does not necessarily fix the values of its variables (unless it is a simple term), which is a departure from how terms are defined in Boolean logic.

In Boolean logic, the only way to get a term that strictly subsumes term τ is by dropping some literals from τ. In multi-valued logic, we can also do this by weakening some literals in term τ (i.e., without dropping any of its variables). This notion of *weakening* a literal generalizes the notion of *dropping* a literal in the Boolean setting. In particular, dropping a Boolean literal ℓ from a Boolean term can be viewed as weakening it into $\ell \vee \neg\ell$.

Term τ is an *implicant* of expression Δ iff $\tau \models \Delta$. Term τ is a *prime implicant* of Δ iff it is an implicant of Δ that is not strictly subsumed by another implicant of Δ. It is possible to have two terms over the same set of variables such that (a) the terms are compatible in that they admit some common instance, (b) both are implicants of some expression Δ, yet (c) only one of them is a prime implicant of Δ. We stress this possibility as it does not arise in a Boolean setting. We define the notions of *simple implicant* and *simple prime implicant* in the expected way.

3.2 Multi-Valued Explanations

Consider now a *classifier* specified using a multi-valued expression Δ. The variables of Δ will be called *features* so an *instance* α is a simple term that mentions all features. That is, an instance fixes a value for each feature of the classifier. A decision on instance α is positive iff the expression Δ evaluates to 1 on instance α, written $\Delta(\alpha) = 1$. Otherwise, the decision is negative (when $\Delta(\alpha) = 0$).

The notation Δ_α is crucial for defining explanations: Δ_α is defined as Δ if decision $\Delta(\alpha)$ is positive and Δ_α is defined as $\neg\Delta$ if decision $\Delta(\alpha)$ is negative. A *PI-explanation* for decision $\Delta(\alpha)$ is a prime implicant of Δ_α that is consistent with instance α. This basically generalizes the notion of PI-explanation introduced in [25] to a multi-valued setting.

The term *explanation* is somewhat too encompassing so any definition of this general notion is likely to draw criticism as being too narrow. The *PI-explanation* is indeed narrow as it is based on a syntactic restriction: it must be a conjunction of literals (i.e., a term) [25]. In the Boolean setting, a PI-explanation is a minimal subset of instance characteristics that is sufficient to trigger the same decision made on the instance. In the multi-valued setting, it can be more generally described as an *abstraction* of the instance that triggers the same decision made on the instance (still in the syntactic form of a term).

As an example, consider the following truth table representing the decision function of a classifier over two ternary variables X and Y:

X, Y	x_1y_1	x_1y_2	x_1y_3	x_2y_1	x_2y_2	x_2y_3	x_3y_1	x_3y_2	x_3y_3
$f(X, Y)$	1	0	0	1	0	0	1	1	1

Consider instance $X{=}x_3 \wedge Y{=}y_1$ leading to a positive decision. The sub-term $X{=}x_3$ is a PI-explanation for this decision: setting input X to x_3 suffices to trigger a positive decision. Similarly, the sub-term $Y{=}y_1$ is a second PI-explanation for this decision. Consider now instance $X{=}x_1 \wedge Y{=}y_2$ leading to a negative decision. This decision has a single PI-explanation: $X{\neq}x_3 \wedge Y{\neq}y_1$. Any instance consistent with this explanation will be decided negatively.

4 Encoding Multi-Valued Behavior

We next discuss three encodings that we tried for the purpose of symbolically representing the behavior of decision trees (and random forests). The first two encodings turned out unsuitable for computing prime implicants. Here, *suitability* refers to the ability of computing multi-valued prime implicants by processing Boolean prime implicants *locally* and *independently*. The third encoding, based on a classical encoding [11], was suitable for this purpose but required a usage that deviates from tradition. Using this encoding in a classical way makes it unsuitable as well. The summary of the findings below is that while an encoding may be appropriate for testing satisfiability or counting models, it may not be suitable for computing prime implicants (and, hence, explanations). While much attention was given to encodings in the context of satisfiability and model counting, we are not aware of similar treatments for computing prime implicants.

4.1 Prefix Encoding

Consider a multi-valued variable X with values $x_1, \ldots, x_n$. This encoding uses Boolean variables $x_2, \ldots, x_n$ to encode the values of variable X. Literal $X{=}x_i$ is encoded by setting the first $i-1$ Boolean variables to 1 and the rest to 0. For example, if $n = 3$, the values of X are encoded as $\bar{x}_2\bar{x}_3$, $x_2\bar{x}_3$ and x_2x_3. Some instantiations of these Boolean variables will not correspond to any value of variable X and are ruled out by enforcing the following constraint: all Boolean variables set to 1 must occur before all Boolean variables set to 0. We denote this constraint by Ψ_X: $\bigwedge_{i\in\{3,\ldots,n\}}(x_i \Rightarrow x_{i-1})$.

The fundamental problem with this encoding is that a multi-valued literal that represents non-contiguous values cannot be represented by a Boolean term. Hence, this encoding cannot generate prime implicants that include such literals. Consider the multi-valued expression $\Delta = (X{=}x_1 \vee X{=}x_3)$, where X has values $x_1, \ldots, x_4$, and its Boolean encoding $\Delta_b = \bar{x}_2\bar{x}_3\bar{x}_4 + x_2x_3\bar{x}_4$. There is only one prime implicant of Δ, which is $X{=}x_1 \vee X{=}x_3$, but this prime implicant cannot be represented by a Boolean term (that implies Δ_b) so it will never be generated.

4.2 Highest-Bit Encoding

Consider a multi-valued variable X with values $x_1, x_2, \ldots, x_n$. This encoding uses Boolean variables $x_2, x_3, \ldots, x_n$ to encode the values of variable X. Every instantiation of these Boolean variables will map to a value of variable X in the following way. If all Boolean variables are 0, then we map the instantiation to value x_1. Otherwise we map an instantiation to the maximum index whose variable is 1. The following table provides an example for $n = 4$.

$x_2x_3x_4$	000	001	010	011	100	101	110	111
highest 1-index	-	4	3	4	2	4	3	4
value	x_1	x_4	x_3	x_4	x_2	x_4	x_3	x_4

We can alternatively view this encoding as representing literal $X{=}x_1$ using the Boolean term $\bar{x}_2 \ldots \bar{x}_n$ and literal $X{=}x_i$, $i \geq 2$, using the term $x_i\bar{x}_{i+1} \ldots \bar{x}_n$. Literals over multiple values can also be represented with this encoding. For example, we can represent the literal $X{=}x_1 \vee X{=}x_2$ using the term $\bar{x}_3\bar{x}_4$.

This encoding also turned out to be unsuitable for computing prime implicants. Consider the multi-valued expression $\Delta = (X{=}x_1 \vee X{=}x_3)$, which has one prime implicant Δ. The Boolean encoding Δ_b is $\bar{x}_2\bar{x}_3\bar{x}_4 + x_3\bar{x}_4$ and has *two* prime implicants $\bar{x}_2\bar{x}_4$ and $x_3\bar{x}_4$. The term $x_3\bar{x}_4$ corresponds to the multi-valued implicant $X{=}x_3$, which is not prime. The term $\bar{x}_2\bar{x}_4$ does not even correspond to a multi-valued term. So in this encoding too, prime implicants of the original multi-valued expression Δ cannot be computed by locally and independently processing prime implicants of the encoded Boolean expression Δ_b.

4.3 One-Hot Encoding

The prefix and highest-bit encodings provide some insights into requirements that enable one to locally and independently map Boolean prime implicants into

multi-valued ones. The requirements are: (1) every multi-valued literal should be representable using a Boolean term, and (2) equivalence and subsumption relations over multi-valued literals should be preserved over their Boolean encodings. The next encoding satisfies these requirements. It is based on [11] but deviates from it in some significant ways that we explain later.

Suppose X is a multi-valued variable with values $x_1, \ldots, x_n$. This encoding uses a Boolean variable x_i for each value x_i of variable X. Suppose now that ℓ is a literal that specifies a subset S of these values. The literal will be encoded using the *negative* Boolean term $\bigwedge_{x_i \notin S} \bar{x}_i$. For example, if variable X has three values, then literal $X{=}x_2$ will be encoded using the negative Boolean term $\bar{x}_1\bar{x}_3$ and literal $X{=}x_1 \vee X{=}x_2$ will be encoded using the negative Boolean term $\bar{x}_3$. This encoding requires the employment of an exactly-one constraint for each variable X, which we denote by Ψ_X: $(\bigvee_i x_i) \wedge \bigwedge_{i \neq j} \neg(x_i \wedge x_j)$. We also use Ψ to denote the conjunction of all exactly-one constraints.

Using the encoding in [11], one typically represents literal $X{=}x_i$ by the Boolean term x_i which *asserts* value x_i. Our encoding, however, represents this literal by *eliminating* all other values of X. The following result reveals why we made this choice (proofs of results can be found in the appendix).

Proposition 1. *Multi-valued terms correspond one-to-one to negative Boolean terms that are consistent with Ψ. Equivalence and subsumption relations on multi-valued terms are preserved on their Boolean encodings.*

Exactly-one constraints are normally added to an encoding as done in [11]. We next show that this leads to unintended results when computing prime implicants, requiring another deviation from [11]. Consider two ternary variables X and Y, the expression $\Delta : X{=}x_1 \vee Y{=}y_1$ and its Boolean encoding $\Delta_b : \bar{x}_2\bar{x}_3 + \bar{y}_2\bar{y}_3$. If Ψ is the conjunction of all exactly-one constraints ($\Psi = \Psi_X \wedge \Psi_Y$), then Δ and $\Delta_b \wedge \Psi$ will each have five models:

Δ	$X{=}x_1, Y{=}y_1$	$X{=}x_1, Y{=}y_2$	$X{=}x_1, Y{=}y_3$	$X{=}x_2, Y{=}y_1$	$X{=}x_3, Y{=}y_1$
$\Delta_b \wedge \Psi$	$x_1\bar{x}_2\bar{x}_3y_1\bar{y}_2\bar{y}_3$	$x_1\bar{x}_2\bar{x}_3\bar{y}_1y_2\bar{y}_3$	$x_1\bar{x}_2\bar{x}_3\bar{y}_1\bar{y}_2y_3$	$\bar{x}_1x_2\bar{x}_3y_1\bar{y}_2\bar{y}_3$	$\bar{x}_1\bar{x}_2x_3y_1\bar{y}_2\bar{y}_3$

The term $X{=}x_1$ is an implicant of Δ. However, its corresponding Boolean encoding $\bar{x}_2\bar{x}_3$ is not an implicant of $\Delta_b \wedge \Psi$ (neither is $x_1\bar{x}_2\bar{x}_3$). For example, $x_1\bar{x}_2\bar{x}_3y_1y_2\bar{y}_3$ does not imply $\Delta_b \wedge \Psi$ since $y_1y_2\bar{y}_3$ does not satisfy the exactly-one constraint Ψ_Y. This motivates Definition 1 below and further results on handling exactly-one constraints, which we introduce after some notational conventions.

In what follows, we use Δ/τ to denote multi-valued expressions/terms, and Γ/ρ to denote Boolean expressions/terms. We also use Δ_b and τ_b to denote the Boolean encodings of Δ and τ. A *completion* of a term is a complete variable instantiation that is consistent with the term. We use α to denote completions. Finally, we use Ψ to denote the conjunction of all exactly-one constraints.

Definition 1. *We define $\rho \models_\Psi \Gamma$ iff $\alpha \models \Gamma$ for all completions α of Boolean term ρ that are consistent with constraint Ψ.*

Note that $\rho \models \Gamma$ implies $\rho \models_\Psi \Gamma$ but the converse is not true.

Proposition 2. *$\rho \models_\Psi \Gamma$ iff $\rho \models (\Psi \Rightarrow \Gamma)$.*

We now show how one-hot encodings can be used for computing prime implicants, particularly, how exactly-one constraints should be integrated.

Proposition 3. *If τ is a term, then $\tau \models \Delta$ iff $\tau_b \models (\Psi \Rightarrow \Delta_b)$.*

The proof is based on two lemmas that hold by construction and that use the notion of *full encoding* of an instance. Consider ternary variables X and Y. For instance $\tau : X{=}x_1 \wedge Y{=}y_1$ the full encoding is $\rho : x_1\bar{x}_2\bar{x}_3y_1\bar{y}_2\bar{y}_3$ (x_1 and y_1 are included). Note that $\rho \wedge \Psi = \rho$ since ρ is guaranteed to satisfy constraints Ψ.

Lemma 1. *If τ is an instance and ρ is its full encoding, then $\tau \models \Delta$ iff $\rho \models \Delta_b$.*

Lemma 2. *For term τ, there is a one-to-one correspondence between the completions of τ and the completions of τ_b that are consistent with Ψ.*

Term $\tau : X{=}x_1 \vee X{=}x_2$ has six completions: $X{=}x_1 \wedge Y{=}y_1$, $X{=}x_2 \wedge Y{=}y_1$, …, $X{=}x_2 \wedge Y{=}y_3$. Its Boolean encoding $\tau_b : \bar{x}_3$ also has six completions that are consistent with Ψ: $x_1\bar{x}_2\bar{x}_3y_1\bar{y}_2\bar{y}_3$, $\bar{x}_1x_2\bar{x}_3y_1\bar{y}_2\bar{y}_3$, …, $\bar{x}_1x_2\bar{x}_3\bar{y}_1\bar{y}_2y_3$. Each of these completions α is guaranteed to satisfy constraints Ψ leading to $\alpha \wedge \Psi = \alpha$. Next, we relate the prime implicants of multi-valued expressions and their encodings.

Proposition 4. *Consider a multi-valued expression Δ and its Boolean encoding Δ_b. If τ is a prime implicant of Δ, then τ_b is a negative term, consistent with Ψ and a prime implicant of $\Psi \Rightarrow \Delta_b$. If ρ is a prime implicant of $\Psi \Rightarrow \Delta_b$, negative and consistent with Ψ, then ρ encodes a prime implicant of Δ.*

This proposition suggests the following procedure for computing multi-valued prime implicants from Boolean prime implicants. Given a multi-valued expression Δ, we encode each literal in Δ using its negative Boolean term, leading to the Boolean expression Δ_b. We then construct the exactly-one constraints Ψ and compute prime implicants of $\Psi \Rightarrow \Delta_b$, keeping those that are negative and consistent with constraints Ψ.[5] Those Boolean prime implicants correspond precisely to the multi-valued prime implicants of Δ.[6]

The only system we are aware of that computes prime implicants of decision tree encodings (and forests) is Xplainer [9]. This system bypasses the encoding complications we alluded to earlier as it computes prime implicants in a specific manner [6, 7]. In particular, it encodes a multi-valued expression into a Boolean expression using the classical one-hot encoding. But rather than computing

[5] It is straightforward to augment the algorithm of [25] so that it only enumerates such prime implicants, by blocking the appropriate branches.

[6] Note that when computing PI-explanations, we are interested only in prime implicants that are consistent with a given instance. Any negative prime implicant which is consistent with an instance must also be consistent with constraints Ψ. The only way a negative Boolean term ρ can violate constraints Ψ is by setting all Boolean variables of some multi-valued variable to false. However, every instance α will set one of these Boolean variables to true so ρ cannot be consistent with α.

prime implicants of the Boolean encoding directly (which would lead to incorrect results), it reduces the problem of computing prime implicants of a multi-valued expression into one that requires only consistency testing of the Boolean encoding, which can be done using repeated calls to a SAT solver. The classical one-hot encoding is sound and complete for this purpose. Our treatment, however, is meant to be independent of the specific algorithm used to compute prime implicants. It would be needed, for example, when compiling the encoding into a tractable circuit and then computing prime implicants as done in [25, 5].

4.4 Encoding Decision Trees and Random Forests

Consider a decision tree, such as the one depicted in Figure 1. Each internal node in the tree represents a decision, which is either true or false. Each leaf is annotated with the predicted label. We can thus view a decision tree as a function whose inputs are all of the unique decisions that can be made in the tree, and whose output is the resulting label. Each leaf of the decision tree represents a simple term over the decisions made on the path to reach it, found by conjoining the appropriate literals. The Boolean function representing a particular class can then be found by simply disjoining the paths for all leaves of that class. That is, this Boolean function outputs true for all inputs that result in the corresponding class label, and false otherwise. We can also obtain this function for an ensemble of decision trees, such as a random forest. We first obtain the Boolean functions of each individual decision tree, and then aggregate them appropriately. For a random forest, we can use a simple majority gate whose inputs are the outputs of each decision tree; see also [1]. Finally, once we have the Boolean function of a classifier, we could apply a SAT or SMT solver to analyze it as proposed by [10, 16, 7]. We could also compile it into a tractable representation, such as an Ordered Binary Decision Diagram (OBDD), and then analyze it as proposed by [25, 24, 26, 23]. In the latter case, a representation such as an OBDD allows us to perform certain queries and transformation on a Boolean function efficiently, which facilitates the explanation and formal verification of the underlying machine learning classifier, as also shown more generally in [1].

5 Conclusion

We considered the encoding of input-output behavior of decision trees and random forests using Boolean expressions. Our focus has been on the suitability of encodings for computing prime implicants, which have recently played a central role in explaining the decisions of machine learning classifiers. Our findings have identified a particular encoding that is suitable for this purpose. Our encoding is based on a classical encoding that has been employed for the task of satisfiability but that can lead to incorrect results when computing prime implicants, which further emphasizes the merit of the investigation we conducted in this paper.

Ack. This work has been partially supported by grants from NSF IIS-1910317, ONR N00014-18-1-2561, DARPA N66001-17-2-4032 and a gift from JP Morgan.

References

1. Audemard, G., Koriche, F., Marquis, P.: On tractable XAI queries based on compiled representations. In: Proc. of KR'20 (2020), to appear
2. Bessiere, C., Hebrard, E., O'Sullivan, B.: Minimising decision tree size as combinatorial optimisation. In: CP. Lecture Notes in Computer Science, vol. 5732, pp. 173–187. Springer (2009)
3. Choi, A., Shi, W., Shih, A., Darwiche, A.: Compiling neural networks into tractable Boolean circuits. In: AAAI Spring Symposium on Verification of Neural Networks (VNN) (2019)
4. Crama, Y., Hammer, P.L.: Boolean Functions - Theory, Algorithms, and Applications, Encyclopedia of mathematics and its applications, vol. 142. Cambridge University Press (2011)
5. Darwiche, A., Hirth, A.: On the reasons behind decisions. In: Proceedings of the 24th European Conference on Artificial Intelligence (ECAI) (2020)
6. Ignatiev, A., Morgado, A., Marques-Silva, J.: Propositional abduction with implicit hitting sets. In: Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI). pp. 1327–1335 (2016)
7. Ignatiev, A., Narodytska, N., Marques-Silva, J.: Abduction-based explanations for machine learning models. In: Proceedings of the Thirty-Third Conference on Artificial Intelligence (AAAI). pp. 1511–1519 (2019)
8. Ignatiev, A., Narodytska, N., Marques-Silva, J.: On relating explanations and adversarial examples. In: Advances in Neural Information Processing Systems 32 (NeurIPS). pp. 15857–15867 (2019)
9. Ignatiev, A., Narodytska, N., Marques-Silva, J.: On validating, repairing and refining heuristic ML explanations. CoRR **abs/1907.02509** (2019)
10. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Computer Aided Verification CAV. pp. 97–117 (2017)
11. de Kleer, J.: A comparison of ATMS and CSP techniques. In: IJCAI. pp. 290–296. Morgan Kaufmann (1989)
12. Leofante, F., Narodytska, N., Pulina, L., Tacchella, A.: Automated verification of neural networks: Advances, challenges and perspectives. CoRR **abs/1805.09938** (2018)
13. McCluskey, E.J.: Minimization of boolean functions. The Bell System Technical Journal **35**(6), 1417–1444 (Nov 1956)
14. Miller, D.M., Thornton, M.A.: Multiple Valued Logic: Concepts and Representations, Synthesis lectures on digital circuits and systems, vol. 12. Morgan & Claypool Publishers (2008)
15. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J.: Learning optimal decision trees with SAT. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI). pp. 1362–1368 (2018)
16. Narodytska, N., Kasiviswanathan, S.P., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI) (2018)
17. Quine, W.V.: The problem of simplifying truth functions. The American Mathematical Monthly **59**(8), 521–531 (1952)
18. Quine, W.V.: On cores and prime implicants of truth functions. The American Mathematical Monthly **66**(9), 755–760 (1959)

19. Ramesh, A., Murray, N.V.: Computing prime implicants/implicates for regular logics. In: Proceedings of the 24th IEEE International Symposium on Multiple-Valued Logic (ISMVL). pp. 115–123 (1994)
20. Renooij, S.: Same-decision probability: Threshold robustness and application to explanation. In: Studeny, M., Kratochvil, V. (eds.) Proceedings of the International Conference on Probabilistic Graphical Models (PGM). Proceedings of Machine Learning Research, vol. 72, pp. 368–379. PMLR (2018)
21. Ribeiro, M.T., Singh, S., Guestrin, C.: Anchors: High-precision model-agnostic explanations. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI) (2018)
22. Ribeiro, M.T., Singh, S., Guestrin, C.: Anchors: High-precision model-agnostic explanations. In: AAAI. pp. 1527–1535. AAAI Press (2018)
23. Shi, W., Shih, A., Darwiche, A., Choi, A.: On tractable representations of binary neural networks. In: Proc. of KR'20 (2020), to appear
24. Shih, A., Choi, A., Darwiche, A.: Formal verification of bayesian network classifiers. In: PGM. Proceedings of Machine Learning Research, vol. 72, pp. 427–438. PMLR (2018)
25. Shih, A., Choi, A., Darwiche, A.: A symbolic approach to explaining bayesian network classifiers. In: IJCAI. pp. 5103–5111. ijcai.org (2018)
26. Shih, A., Darwiche, A., Choi, A.: Verifying binarized neural networks by angluin-style learning. In: SAT (2019)
27. Walsh, T.: SAT v CSP. In: CP. Lecture Notes in Computer Science, vol. 1894, pp. 441–456. Springer (2000)

A Proofs

Proof (of Proposition 1). For multi-valued term τ, the Boolean encoding τ_b is a negative term and consistent with Ψ by construction. Suppose now that ρ is a negative Boolean term that is consistent with Ψ. If ρ mentions a Boolean variable of multi-valued variable X, then ρ cannot mention all Boolean variables of X, otherwise ρ will be ruling out all possible values of X and hence inconsistent with Ψ. Hence, ρ encodes a literal over variable X when ρ mentions a Boolean variable for X. More generally, ρ encodes a term over multi-valued variables whose Boolean variables are mentioned in ρ. To prove the second part of the theorem, consider literals ℓ_1 and ℓ_2, which specify values S_1 and S_2 for variable X. The two literals are equivalent iff $S_1 = S_2$ iff $\bigwedge_{x_i \notin S_1} \bar{x}_i$ and $\bigwedge_{x_i \notin S_2} \bar{x}_i$ are equivalent. Moreover, $\ell_1 \models \ell_2$ iff $S_1 \subseteq S_2$ iff $\bigwedge_{x_i \notin S_1} \bar{x}_i \models \bigwedge_{x_i \notin S_2} \bar{x}_i$. Equivalence and subsumption relations are then preserved on literals, and on terms as well.

Proof (of Proposition 2). ($\Rightarrow$) Suppose $\rho \models_\Psi \Gamma$ and let α be a completion of ρ. If α is consistent with Ψ, then $\alpha \models \Gamma$ by Definition 1. If α is not consistent with Ψ, then $\alpha \models \neg\Psi$. Hence, $\rho \models \neg\Psi \vee \Gamma$. ($\Leftarrow$) Suppose $\rho \models \neg\Psi \vee \Gamma$ and let α be a completion of ρ that is consistent with Ψ. Then $\alpha \models \neg\Psi \vee \Gamma$ and, hence, $\alpha \wedge \Psi \models \Gamma$ and $\alpha \models \Gamma$. We then have $\rho \models_\Psi \Gamma$ by Definition 1.

Proof (of Proposition 3). ($\Rightarrow$) Suppose $\tau \models \Delta$. Then $\alpha \models \Delta$ for all completions α of τ. By Lemmas 1 and 2, $\alpha_b \models \Delta_b$ for all completions α_b of τ_b that are consistent with Ψ. Hence $\tau_b \models \neg\Psi \vee \Delta_b$. ($\Leftarrow$) Suppose $\tau_b \models \neg\Psi \vee \Delta_b$ and let

α_b be a completion of τ_b ($\alpha_b \models \neg\Psi \vee \Delta_b$). For each α_b consistent with Ψ, we have $\alpha_b \models \Psi$ and hence $\alpha_b \models \Delta_b$. By Lemmas 1 and 2, the completions α of τ correspond to these α_b (consistent with Ψ), leading to $\alpha \models \Delta$ and hence $\tau \models \Delta$.

Proof (of Proposition 4). ($\Rightarrow$) Suppose τ is a prime implicant of Δ. Then $\tau \models \Delta$. Moreover, $\tau_b \models (\Psi \Rightarrow \Delta_b)$ by Proposition 3 so τ_b is an implicant of $\Psi \Rightarrow \Delta_b$ (τ_b is negative and consistent with Ψ by construction). Suppose τ_b is not a prime implicant of $\Psi \Rightarrow \Delta_b$. Then $\rho \models (\Psi \Rightarrow \Delta_b)$ for a strict subset ρ of τ_b, which must be consistent with Ψ since $\tau_b \supset \rho$ is consistent with Ψ. Hence, ρ encodes a term $\tau^\star$ that is strictly weaker than term τ by Proposition 1. Moreover, $\tau^\star \models \Delta$ by Proposition 3 so τ is not a prime implicant of Δ, which is a contradiction. Therefore, τ_b is a prime implicant of $\Psi \Rightarrow \Delta_b$. ($\Leftarrow$) Suppose ρ is a prime implicant of $\Psi \Rightarrow \Delta_b$, negative and consistent with Ψ. Then ρ encodes a term τ by Proposition 1. Moreover, $\rho = \tau_b \models \Psi \Rightarrow \Delta_b$ so $\tau \models \Delta$ by Proposition 3. Hence, τ is an implicant of Δ. Suppose now that $\tau^\star \models \Delta$ for some term $\tau^\star$ that is strictly weaker than term τ. Then $\tau_b^\star \models \Psi \Rightarrow \Delta_b$ by Proposition 3. This means ρ is not a prime implicant of $\Psi \Rightarrow \Delta_b$ since $\tau_b^\star \subset \tau_b = \rho$ by Proposition 1, which is a contradiction. Hence, the term τ encoded by ρ is a prime implicant of Δ.

An Abstraction-Based Framework for Neural Network Verification

Abstract. Deep neural networks are increasingly being used as controllers for safety-critical systems. Because neural networks are opaque, certifying their correctness is a significant challenge. To address this issue, several approaches have recently been proposed to formally verify them. However, network size is often a bottleneck for such approaches and it can be difficult to apply them to large networks. In this paper, we propose a framework that can enhance neural network verification techniques by using over-approximation to reduce the size of the network — thus making it more amenable to verification. We perform the approximation such that if the property holds for the smaller (abstract) network, it holds for the original as well. The over-approximation may be too coarse, in which case the underlying verification tool might return a spurious counterexample. Under such conditions, we perform counterexample-guided refinement to adjust the approximation, and then repeat the process. Our approach is orthogonal to, and can be integrated with, many existing verification techniques. For evaluation purposes, we integrate it with the recently proposed Marabou framework, and observe a significant improvement in Marabou's performance. Our experiments demonstrate the great potential of our approach for verifying larger neural networks.

1 Introduction

Machine programming (MP), the automatic generation of software, is showing early signs of fundamentally transforming the way software is developed [11]. A key ingredient employed by MP is the *deep neural network* (DNN), which has emerged as an effective means to semi-autonomously implement many complex software systems. DNNs are artifacts produced by *machine learning*: a user provides examples of how a system should behave, and a machine learning algorithm generalizes these examples into a DNN capable of correctly handling inputs that it had not seen before. Systems with DNN components have obtained unprecedented results in fields such as image recognition [19], game playing [27], natural language processing [12], computer networks [22], and many others, often surpassing the results obtained by similar systems that have been carefully handcrafted. It seems evident that this trend will increase and intensify, and that DNN components will be deployed in various safety-critical systems [2,14].

DNNs are appealing in that (in some cases) they are easier to create than handcrafted software, while still achieving excellent results. However, their usage also raises a challenge when it comes to certification. Undesired behavior has been observed in many state-of-the-art DNNs. For example, in many cases slight perturbations to correctly handled inputs can cause severe errors [28,20]. Because

many practices for improving the reliability of hand-crafted code have yet to be successfully applied to DNNs (e.g., code reviews, coding guidelines, etc.), it remains unclear how to overcome the opacity of DNNs, which may limit our ability to certify them before they are deployed.

To mitigate this, the formal methods community has begun developing techniques for the formal verification of DNNs (e.g., [8,13,15,30]). These techniques can automatically prove that a DNN always satisfies a prescribed property. Unfortunately, the DNN verification problem is computationally difficult (e.g., NP-complete, even for simple specifications and networks [15]), and becomes exponentially more difficult as network sizes increase. Thus, despite recent advances in DNN verification techniques, network sizes remain a severely limiting factor.

In this work, we propose a technique by which the scalability of many existing verification techniques can be significantly increased. The idea is to apply the well-established notion of *abstraction and refinement* [5]: replace a network N that is to be verified with a much smaller, *abstract* network, $\bar{N}$, and then verify this $\bar{N}$. Because $\bar{N}$ is smaller it can be verified more efficiently; and it is constructed in such a way that if it satisfies the specification, the original network N also satisfies it. In the case that $\bar{N}$ does not specify the specification, the verification procedure provides a counterexample x. This x may be a true counterexample demonstrating that the original network N violates the specification, or it may be *spurious*. If x is spurious, the network $\bar{N}$ is *refined* to make it more accurate (and slightly larger), and then the process is repeated. A particularly useful variant of this approach is to use the spurious x to guide the refinement process, so that the refinement step rules out x as a counterexample. This variant, known as *counterexample guided abstraction refinement* (*CEGAR*) [5], has been successfully applied in many verification contexts.

As part of our technique we propose a method for abstracting and refining neural networks. Our basic abstraction step *merges* two neurons into one, thus reducing the overall number of neurons by one. This basic step can be repeated numerous times, significantly reducing the network size. Conversely, refinement is performed by splitting a previously merged neuron in two, increasing the network size but making it more closely resemble the original. A key point is that not all pairs of neurons can be merged, as this could result in a network that is smaller but is not an over-approximation of the original. We resolve this by first transforming the original network into an equivalent network where each node belongs to one of four classes, determined by its edge weights and its effect on the network's output; merging neurons from the same class can then be done safely. The actual choice of which neurons to merge or split is performed heuristically. We propose and discuss several possible heuristics.

For evaluation purposes, we implemented our approach as a Python framework that wraps the Marabou verification tool [17]. We then used our framework to verify properties of the Airborne Collision Avoidance System (ACAS Xu) set of benchmarks [15]. Our results strongly demonstrate the potential usefulness of abstraction in enhancing existing verification schemes: specifically, in most cases the abstraction-enhanced Marabou significantly outperformed the original. Fur-

ther, in most cases the properties in question could indeed be shown to hold or not hold for the original DNN by verifying a small, abstract version thereof.

To summarize, our contributions are: (i) we propose a general framework for over-approximating and refining DNNs; (ii) we propose several heuristics for abstraction and refinement, to be used within our general framework; and (iii) we provide an implementation of our technique that integrates with the Marabou verification tool and use it for evaluation.[1]

The rest of this paper is organized as follows. In Section 2, we provide a brief background on neural networks and their verification. In Section 3, we describe our general framework for abstracting an refining DNNs. In Section 4, we discuss how to apply these abstraction and refinement steps as part of a CEGAR procedure, followed by an evaluation in Section 5. In Section 6, we discuss related work, and we conclude in Section 7.

2 Background

2.1 Neural Networks

A neural network consists of an *input layer*, an *output layer*, and one or more intermediate layers called *hidden layers*. Each layer is a collection of nodes, called *neurons*. Each neuron is connected to other neurons by one or more directed edges. In a feedforward neural network, the neurons in the first layer receive input data that sets their initial values. The remaining neurons calculate their values using the weighted values of the neurons that they are connected to through edges from the preceding layer (see Fig. 1). The output layer provides the resulting value of the DNN for a given input.

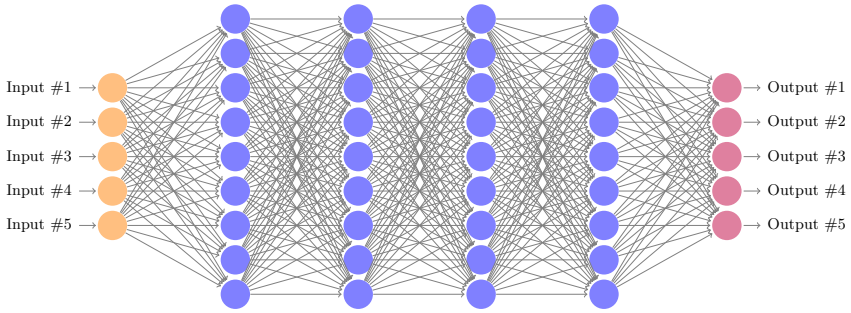


Fig. 1. A fully connected, feedforward DNN with 5 input nodes (in orange), 5 output nodes (in purple), and 4 hidden layers containing a total of 36 hidden nodes (in blue). Each edge is associated with a weight value (not depicted).

[1] We intend to make our code publicly available with the final version of this paper.

There are many types of DNNs, which may differ in the way their neuron values are computed. Typically, a neuron is evaluated by first computing a weighted sum of the preceding layer's neuron values according to the edge weights, and then applying an activation function to this weighted sum [9]. We focus here on the Rectified Linear Unit (ReLU) activation function [23], given as $\mathrm{ReLU}(x) = \max(0, x)$. Thus, if the weighted sum computation yields a positive value, it is kept; and otherwise, it is replaced by zero.

More formally, given a DNN N, we use n to denote the number of layers of N. We denote the number of nodes of layer i by s_i. Layers 1 and n are the input and output layers, respectively. Layers $2, \ldots, n-1$ are the hidden layers. We denote the value of the j-th node of layer i by $v_{i,j}$, and denote the column vector $[v_{i,1}, \ldots, v_{i,s_i}]^T$ as V_i.

Evaluating N is performed by calculating V_n for a given input assignment V_1. This is done by sequentially computing V_i for $i = 2, 3, \ldots, n$, each time using the values of V_{i-1} to compute weighted sums, and then applying the ReLU activation functions. Specifically, layer i (for $i > 1$) is associated with a weight matrix W_i of size $s_i \times s_{i-1}$ and a bias vector B_i of size s_i. If i is a hidden layer, its values are given by $V_i = \mathrm{ReLU}(W_i V_{i-1} + B_i)$, where the ReLUs are applied element-wise; and the output layer is given by $V_n = W_n V_{n-1} + B_n$ (ReLUs are not applied). Without loss of generality, in the rest of the paper we assume that all bias values are 0, and can be ignored. This rule is applied repeatedly once for each layer, until V_n is eventually computed.

We will sometimes use the notation $w(v_{i,j}, v_{i+1,k})$ to refer to the entry of W_{i+1} that represents the weight of the edge between neuron j of layer i and neuron k of layer $i+1$. We will also refer to this edge as an *outgoing edge* for $v_{i,j}$, and as an *incoming edge* for $v_{i+1,k}$.

As part of our abstraction framework, we will sometimes need to consider a *suffix* of a DNN, in which the first layers of the DNN are omitted. For $1 < i < n$, we use $N^{[i]}$ to denote the DNN comprised of layers $i, i+1, \ldots, n$ of the original network. The sizes and weights of the remaining layers are unchanged, and layer i of N is treated as the input layer of $N^{[i]}$.

Fig. 2 depicts a small neural network. The network has $n = 3$ layers, of sizes $s_1 = 1, s_2 = 2$ and $s_3 = 1$. Its weights are $w(v_{1,1}, v_{2,1}) = 1$, $w(v_{1,1}, v_{2,2}) = -1$, $w(v_{2,1}, v_{3,1}) = 1$ and $w(v_{2,2}, v_{3,1}) = 2$. For input $v_{1,1} = 3$, node $v_{2,1}$ evaluates to 3 and node $v_{2,2}$ evaluates to 0, due to the ReLU activation function. The output node $v_{3,1}$ then evaluates to 3.

2.2 Neural Network Verification

DNN verification amounts to answering the following question: given a DNN N, which maps input vector x to output vector y, and predicates P and Q, does there exist an input x_0 such that $P(x_0)$ and $Q(y_0)$ both hold, where $y_0 = N(x_0)$? In other words, the verification process determines whether there exists a particular input that meets the input criterion P, and that is mapped to an output that meets the output criterion Q. We refer to $\langle N, P, Q \rangle$ as the *verification query*. As is usual in verification, Q represents the *negation* of the desired property. Thus,

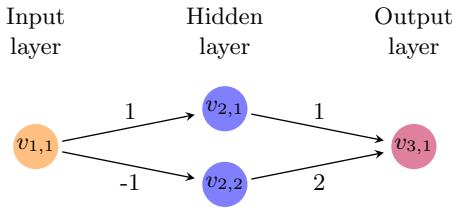


Fig. 2. A simple feedforward neural network.

if the query is *unsatisfiable* (UNSAT), the property holds; and if it is *satisfiable* (SAT), then x_0 constitutes a counterexample to the property in question.

Different verification approaches may differ in *(i)* the kinds of neural networks they allow (specifically, the kinds of activation functions in use); *(ii)* the kinds of input properties; and *(iii)* the kinds of output properties. For simplicity, we focus on networks that employ the ReLU activation function. In addition, our input properties will be conjunctions of linear constraints on the input values. Finally, we will assume that our networks have a single output node y, and that the output property is $y > c$ for a given constant c. We stress that these restrictions are for the sake of simplicity. Many properties of interest, including those with arbitrary Boolean structure and involving multiple neurons, can be reduced into the above single-output setting by adding a few neurons that encode the Boolean structure [15,26]; see Fig 3 for an example. The number of neurons to be added is typically negligible when compared to the size of the DNN. In particular, this is true for the ACAS Xu family of benchmarks [15], and also for adversarial robustness queries that use the L_∞ or the L_1 norm as a distance metric [4,10,16]. Additionally, other piecewise-linear activation functions, such as max-pooling layers, can also be encoded using ReLUs [4].

Several techniques have been proposed for solving the aforementioned verification problem in recent years (Section 6 includes a brief overview). Our abstraction technique is designed to be compatible with most of these techniques, by simplifying the network being verified, as we describe next.

3 Network Abstraction and Refinement

Because the complexity of verifying a neural network is strongly connected to its size [15], our goal is to transform a verification query $\varphi_1 = \langle N, P, Q \rangle$ into query $\varphi_2 = \langle \bar{N}, P, Q \rangle$, such that the abstract network $\bar{N}$ is significantly smaller than N (notice that properties P and Q remain unchanged). We will construct $\bar{N}$ so that it is an over-approximation of N, meaning that if φ_2 is UNSAT then φ_1 is also UNSAT. More specifically, since our DNNs have a single output, we can regard $N(x)$ and $\bar{N}(x)$ as real values for every input x. To guarantee that φ_2 over-approximates φ_1, we will make sure that for every x, $N(x) \leq \bar{N}(x)$; and thus, $\bar{N}(x) \leq c \implies N(x) \leq c$. Because our output properties always have the form $N(x) > c$, it is indeed the case that if φ_2 is UNSAT, i.e. $\bar{N}(x) \leq c$ for all x,

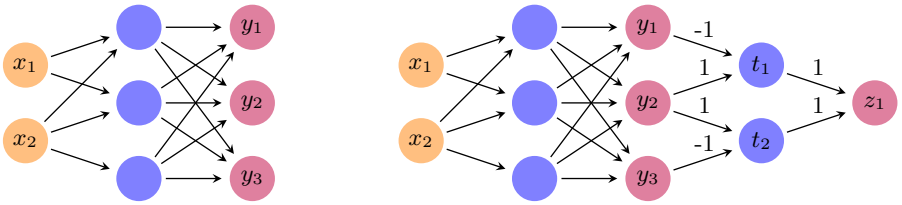


Fig. 3. Reducing a complex property to the $y > 0$ form. For the network on the left hand side, suppose we wish to examine the property $y_2 > y_1 \vee y_2 > y_3$, which is a property that involves multiple outputs and includes a disjunction. We do this (right hand side network) by adding two neurons, t_1 and t_2, such that $t_1 = \text{ReLU}(y_2 - y_1)$ and $t_2 = \text{ReLU}(y_2 - y_3)$. Thus, $t_1 > 0$ if and only if the first disjunct, $y_2 > y_1$, holds; and $t_2 > 0$ if and only if the second disjunct, $y_2 > y_3$, holds. Finally, we add a neuron z_1 such that $z_1 = t_1 + t_2$. It holds that $z_1 > 0$ if and only if $t_1 > 0 \vee t_2 > 0$. Thus, we have reduced the complex property into an equivalent property in the desired form.

then $N(x) \leq c$ for all x and so φ_1 is also `UNSAT`. We now propose a framework for generating various $\bar{N}$s with this property.

3.1 Abstraction

We seek to define an abstraction operator that removes a single neuron from the network, by merging it with another neuron. To do this, we will first transform N into an equivalent network, whose neurons have properties that will facilitate their merging. Equivalent here means that for every input vector, both networks produce the exact same output. First, each hidden neuron $v_{i,j}$ of our transformed network will be classified as either a `pos` neuron or a `neg` neuron. A neuron is `pos` if all the weights on its outgoing edges are positive, and is `neg` if all those weights are negative. Second, orthogonally to the `pos`/`neg` classification, each hidden neuron will also be classified as either an `inc` neuron or a `dec` neuron. $v_{i,j}$ is an `inc` neuron of N if, when we look at $N^{[i]}$ (where $v_{i,j}$ is an input neuron), increasing the value of $v_{i,j}$ increases the value of the network's output. Formally, $v_{i,j}$ is `inc` if for every two input vectors x_1 and x_2 where $x_1[k] = x_2[k]$ for $k \neq j$ and $x_1[j] > x_2[j]$, it holds that $N^{[i]}(x_1) > N^{[i]}(x_2)$. A `dec` neuron is defined symmetrically, so that *decreasing* the value of $x[j]$ *increases* the output. We first describe this transformation (an illustration of which appears in Fig. 4), and later we explain how it fits into our abstraction framework.

Our first step is to transform N into a new network, N', in which every hidden neuron is classified as `pos` or `neg`. This transformation is done by replacing each hidden neuron v_{i_j} with two neurons, $v^+_{i,j}$ and $v^-_{i,j}$, which are respectively `pos` and `dec`. Both $v^+_{i,j}$ an $v^-_{i,j}$ retain a copy of all incoming edges of the original $v_{i,j}$; however, $v^+_{i,j}$ retains just the outgoing edges with positive weights, and $v^-_{i,j}$ retains just those with negative weights. Outgoing edges with negative weights are removed from $v^+_{i,j}$ by setting their weights to 0, and the same is done for outgoing edges with positive weights for $v^-_{i,j}$. Formally, for every neuron $v_{i-1,p}$,

$$w'(v_{i-1,p}, v^+_{i,j}) = w(v_{i-1,p}, v_{i,j}), \qquad w'(v_{i-1,p}, v^-_{i,j}) = w(v_{i-1,p}, v_{i,j})$$

where w' represents the weights in the new network N'. Also, for every neuron $v_{i+1,q}$

$$w'(v^+_{i,j}, v_{i+1,q}) = \begin{cases} w(v_{i,j}, v_{i+1,q}) & w(v_{i,j}, v_{i+1,q}) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$w'(v^-_{i,j}, v_{i+1,q}) = \begin{cases} w(v_{i,j}, v_{i+1,q}) & w(v_{i,j}, v_{i+1,q}) \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

(see Fig. 4). This operation is performed once for every hidden neuron of N, resulting in a network N' that is roughly double the size of N. Observe that N' is indeed equivalent to N, i.e. their outputs are always identical.

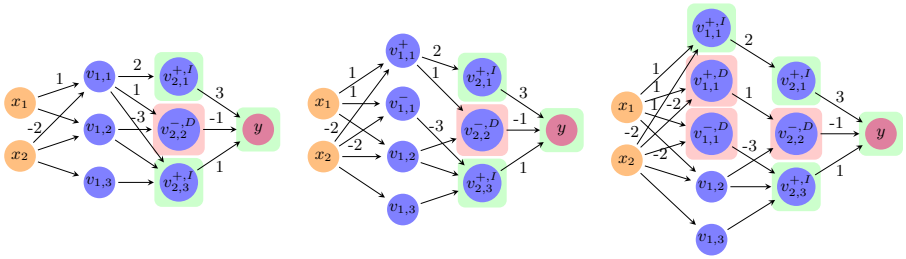


Fig. 4. Classifying neurons as `pos`/`dec` and `inc`/`dec`. In the initial network (left), the neurons of the second hidden layer are already classified: $^+$ and $^-$ superscripts indicate `pos` and `neg` neurons, respectively; the ^{I} superscript and green background indicate `inc`, and the ^{D} superscript and red background indicate `dec`. Classifying node $v_{1,1}$ is done by first splitting it into two nodes $v^+_{1,1}$ and $v^-_{1,1}$ (middle). Both nodes have identical incoming edges, but the outgoing edges of $v_{1,1}$ are partitioned between them, according to the sign of each edge's weight. In the last network (right), $v^+_{1,1}$ is split once more, into an `inc` node with outgoing edges only to other `inc` nodes, and a `dec` node with outgoing edges only to other `dec` nodes. Node $v_{1,1}$ is thus transformed into three nodes, each of which can finally be classified as `inc` or `dec`. Notice that in the worst case, each node is split into four nodes, although for $v_{1,1}$ three nodes were enough.

Our second step is to alter N' further, into a new network N'', where every hidden neuron is either `inc` or `dec` (in addition to already being `pos` or `neg`). Generating N'' from N' is performed by traversing the layers of N' backwards, each time handling a single layer and possibly doubling its number of neurons:

- Initial step: the output layer has a single neuron, y. This neuron is an `inc` node, because increasing its value will increase the network's output value.
- Iterative step: observe layer i, and suppose the nodes of layer $i+1$ have already been partitioned into `inc` and `dec` nodes. Observe a neuron $v^+_{i,j}$ in layer i which is marked `pos` (the case for `neg` is symmetrical). We replace $v^+_{i,j}$ with two neurons $v^{+,I}_{i,j}$ and $v^{+,D}_{i,j}$, which are `inc` and `dec`, respectively.

Both new neurons retain a copy of all incoming edges of $v^+_{i,j}$; however, $v^{+,I}_{i,j}$ retains only outgoing edges that lead to `inc` nodes, and $v^{+,D}_{i,j}$ retains only outgoing edges that lead to `dec` nodes. Thus, for every $v_{i-1,p}$ and $v_{i+1,q}$,

$$w''(v_{i-1,p}, v^{+,I}_{i,j}) = w'(v_{i-1,p}, v^+_{i,j}), \qquad w''(v_{i-1,p}, v^{+,D}_{i,j}) = w'(v_{i-1,p}, v^+_{i,j})$$

$$w''(v^{+,I}_{i,j}, v_{i+1,q}) = \begin{cases} w'(v^+_{i,j}, v_{i+1,q}) & \text{if } v_{i+1,q} \text{ is } \texttt{inc} \\ 0 & \text{otherwise} \end{cases}$$

$$w''(v^{+,D}_{i,j}, v_{i+1,q}) = \begin{cases} w'(v^+_{i,j}, v_{i+1,q}) & \text{if } v_{i+1,q} \text{ is } \texttt{dec} \\ 0 & \text{otherwise} \end{cases}$$

where w'' represents the weights in the new network N''. We perform this step for each neuron in layer i, resulting in neurons that are each classified as either `inc` or `dec`.

To understand the intuition behind this classification, recall that by our assumption all hidden nodes, including $v^+_{i,j}$, use the ReLU activation function and so take on only non-negative values. Because $v^+_{i,j}$ is `pos`, all its outgoing edges have positive weights, and so if its assignment was to increase (decrease), the assignments of all nodes to which it is connected in the following layer would also increase (decrease). Thus, we split $v^+_{i,j}$ in two, and make sure one copy, $v^{+,I}_{i,j}$, is only connected to nodes that need to increase (`inc` nodes), and that the other copy, $v^{+,D}_{i,j}$, is only connected to nodes that need to decrease (`dec` nodes). This ensures that $v^{+,I}_{i,j}$ is itself `inc`, and that $v^{+,D}_{i,j}$ is `dec`. Also, both $v^{+,I}_{i,j}$ and $v^{+,D}_{i,j}$ remain `pos` nodes, because their outgoing edges all have positive weights.

When this procedure terminates, N'' is equivalent to N', and so also to N; and N'' is double the size of N', and four times the size of N. Both transformation steps are only performed for hidden neurons, whereas the input and output neurons remain unchanged. This is summarized by the following lemma:

Lemma 1. *Any DNN N can be transformed into an equivalent network N'' where each hidden neuron is* `pos` *or* `dec`*, and also* `inc` *or* `dec`*, by increasing its number of neurons by a factor of at most* 4*.*

Using Lemma 1, we can assume without loss of generality that the DNN nodes in our input query φ_1 are each marked as `pos`/`neg` and as `inc`/`dec`. We are now ready to construct the over-approximation network $\bar{N}$. We do this by specifying an `abstract` operator that merges a pair of neurons in the network (thus reducing network size by one), and can be applied multiple times. The only restrictions are that the two neurons being merged need to be from the same hidden layer, and must share the same `pos`/`neg` and `inc`/`dec` attributes. Consequently, applying `abstract` to saturation will result in a network with at most 4 neurons in each hidden layer, which over-approximates the original network. This, of course, would be an immense reduction in the number of neurons for most reasonable input networks.

The `abstract` operator's behavior depends on the attributes of the neurons being merged. For simplicity, we will focus on the ⟨`pos`,`inc`⟩ case. Let $v_{i,j}$, $v_{i,k}$ be two hidden neurons of layer i, both classified as ⟨`pos`,`inc`⟩. Because layer i is hidden, we know that layers $i+1$ and $i-1$ are defined. Let $v_{i-1,p}$ and $v_{i+1,q}$ denote arbitrary neurons in the preceding and succeeding layer, respectively. We construct a network $\bar{N}$ that is identical to N, except that: (i) nodes $v_{i,j}$ and $v_{i,k}$ are removed and replaced with a new single node, $v_{i,t}$; and (ii) all edges that touched nodes $v_{i,j}$ or $v_{i,k}$ are removed, and other edges are untouched. Finally, we add new incoming and outgoing edges for the new node $v_{i,t}$ as follows:

- Incoming edges: $\bar{w}(v_{i-1,p}, v_{i,t}) = \max\{w(v_{i-1,p}, v_{i,j}), w(v_{i-1,p}, v_{i,k})\}$
- Outgoing edges: $\bar{w}(v_{i,t}, v_{i+1,q}) = w(v_{i,j}, v_{i+1,q}) + w(v_{i,k}, v_{i+1,q})$

where $\bar{w}$ represents the weights in the new network $\bar{N}$. An illustrative example appears in Fig. 5. Intuitively, this definition of `abstract` seeks to ensure that the new node $v_{i,t}$ always contributes more to the network's output than the two original nodes $v_{i,j}$ and $v_{i,k}$ — so that the new network produces a larger output than the original for every input. By the way we defined the incoming edges of the new neuron $v_{i,t}$, we are guaranteed that for every input x passed into both N and $\bar{N}$, the value assigned to $v_{i,t}$ in $\bar{N}$ is greater than the values assigned to both $v_{i,j}$ and $v_{i,k}$ in the original network. This works to our advantage, because $v_{i,j}$ and $v_{i,k}$ were both `inc` — so increasing their values increases the output value. By our definition of the outgoing edges, the values of any `inc` nodes in layer $i+1$ increase in $\bar{N}$ compared to N, and those of any `dec` nodes decrease. By definition, this means that the network's overall output increases.

The abstraction operation for the ⟨`neg`,`inc`⟩ case is identical to the one described above. For the remaining two cases, i.e. ⟨`pos`,`dec`⟩ and ⟨`neg`,`dec`⟩, the max operator in the definition is replaced with a min.

The next lemma (proof omitted due to lack of space) justifies the use of our abstraction step, and can be applied once per each application of `abstract`:

Lemma 2. *Let $\bar{N}$ be derived from N by a single application of `abstract`. For every x, it holds that $\bar{N}(x) \geq N(x)$.*

3.2 Refinement

The aforementioned `abstract` operator reduces network size by merging neurons, but at the cost of accuracy: whereas for some input x_0 the original network returns $N(x_0) = 3$, the over-approximation network $\bar{N}$ created by `abstract` might return $\bar{N}(x_0) = 5$. If our goal is prove that it is never the case that $N(x) > 10$, this over-approximation may be adequate: we can prove that always $\bar{N}(x) \leq 10$, and this will be enough. However, if our goal is to prove that it is never the case that $N(x) > 4$, the over-approximation is inadequate: it is possible that the property holds for N, but because $\bar{N}(x_0) = 5 > 4$, our verification procedure will return x_0 as a *spurious counterexample* (a counterexample for $\bar{N}$ that is not a counterexample for N). In order to handle this situation, we

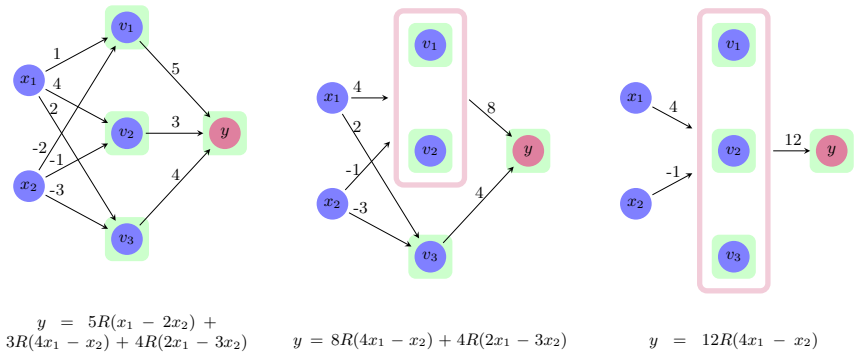


Fig. 5. Using `abstract` to merge ⟨`pos`,`inc`⟩ nodes. Initially (left), the three nodes v_1, v_2 and v_3 are separate. Next (middle), `abstract` merges v_1 and v_2 into a single node. For the edge between x_1 and the new abstract node we pick the weight 4, which is the maximal weight among edges from x_1 to v_1 and v_2. Likewise, the edge between x_2 and the abstract node has weight -1. The outgoing edge from the abstract node to y has weight 8, which is the sum of the weights of edges from v_1 and v_2 to y. Next, `abstract` is applied again to merge v_3 with the abstract node, and the weights are adjusted accordingly (right). With every abstraction, the value of y (given as a formula at the bottom of each DNN, where R represents the ReLU operator) increases. For example, to see that $12R(4x_1 - x_2) \geq 8R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$, it is enough to see that $4R(4x_1 - x_2) \geq 4R(2x_1 - 3x_2)$, which holds because ReLU is a monotonically increasing function and x_1 and x_2 are non-negative (being, themselves, the output of ReLU nodes).

define a *refinement operator*, `refine`, that is the inverse of `abstract`: it transforms $\bar{N}$ into yet another over-approximation, $\bar{N}'$, with the property that for every x, $N(x) \leq \bar{N}'(x) \leq \bar{N}(x)$. If $\bar{N}'(x_0) = 3.5$, it might be a suitable over-approximation for showing that never $N(x) > 4$. In this section we define the `refine` operator, and in Section 4 we explain how to use `abstract` and `refine` as part of a CEGAR-based verification scheme.

Recall that `abstract` merges together a couple of neurons that share the same attributes. After a series of applications of `abstract`, each hidden layer i of the resulting network can be regarded as a partitioning of hidden layer i of the original network, where each partition contains original, *concrete* neurons that share the same attributes. In the abstract network, each partition is represented by a single, *abstract* neuron. The weights on the incoming and outgoing edges of this abstract neuron are determined according to the definition of the `abstract` operator. For example, in the case of an abstract neuron $\bar{v}$ that represents a set of concrete neurons $\{v_1, \ldots, v_n\}$ all with attributes ⟨`pos`,`inc`⟩, the weight of each incoming edge to $\bar{v}$ is given by

$$\bar{w}(u, v) = \max(w(u, v_1), \ldots, w(u, v_n))$$

where u represents a neuron that has not been abstracted yet, and w is the weight function of the original network. The key point here is that the order of `abstract` operations that merged $v_1, \ldots, v_n$ does not matter — but rather, only

the fact that they are now grouped together determines the abstract network's weights. The following corollary, which is a direct result of Lemma 2, establishes this connection between sequences of `abstract` applications and partitions:

Corollary 1. *Let N be a DNN where each hidden neuron is labeled as* `pos/neg` *and* `inc/dec`*, and let $\mathcal{P}$ be a partitioning of the hidden neurons of N, that only groups together hidden neurons from the same layer that share the same labels. Then N and $\mathcal{P}$ give rise to an abstract neural network $\bar{N}$, which is obtained by performing a series of* `abstract` *operations that group together neurons according to the partitions of $\mathcal{P}$. This $\bar{N}$ is an over-approximation of N.*

We now define a `refine` operation that is, in a sense, the inverse of `abstract`. `refine` takes as input a DNN $\bar{N}$ that was generated from N via a sequence of `abstract` operations, and splits a neuron from $\bar{N}$ in two. Formally, the operator receives the original network N, the partitioning $\mathcal{P}$, and a finer partition $\mathcal{P}'$ that is obtained from $\mathcal{P}$ by splitting a single class in two. The operator then returns a new abstract network, $\bar{N}'$, that is the abstraction of N according to $\mathcal{P}'$.

Due to Corollary 1, and because $\bar{N}$ returned by `refine` corresponds to a partition $\mathcal{P}'$ of the hidden neurons of N, it is straightforward to show that $\bar{N}$ is indeed an over-approximation of N. The other useful property that we require is the following:

Lemma 3. *Let $\bar{N}$ be an abstraction of N, and let $\bar{N}'$ be a network obtained from $\bar{N}$ by applying a single* `refine` *step. Then for every input x it holds that $\bar{N}(x) \geq \bar{N}'(x) \geq N(x)$.*

The second part of the inequality, $\bar{N}'(x) \geq N(x)$ holds because $\bar{N}'$ is an over-approximation of N (Corollary 1). The first part of the inequality, $\bar{N}(x) \geq \bar{N}'(x)$, follows from the fact that $\bar{N}(x)$ can be obtained from $\bar{N}'(x)$ by a single application of `abstract`.

Of course, in practice, starting from the original N and applying a sequence of `abstract` operations is a wasteful way of implementing `refine`. Instead, it is possible to split an abstract neuron in two in a single step and examine just the concrete neurons that are mapped to the two new abstract neurons to determine the new edge weights.

4 A CEGAR-Based Approach

In Section 3 we defined the `abstract` operator that reduces network size at the cost of reducing network accuracy, and its inverse `refine` operator that increases network size and restores accuracy. Together with a black-box verification procedure *Verify* that can dispatch queries of the form $\varphi = \langle N, P, Q \rangle$, these components now allow us to design an abstraction-refinement algorithm for DNN verification, given as Alg. 1 (we assume that all hidden neurons in the input network have already been marked `pos`/`neg` and `inc`/`dec`).

Because our over-approximation network $\bar{N}$ is obtained via applications of `abstract` and `refine`, and because we assume the underlying *Verify* procedure

Algorithm 1 Abstraction-based DNN Verification(N, P, Q)

1: Use `abstract` to generate an initial over-approximation $\bar{N}$ of N
2: **if** *Verify*$(\bar{N}, P, Q)$ is `UNSAT` **then**
3: return `UNSAT`
4: **else**
5: Extract counterexample c
6: **if** c is a counterexample for N **then**
7: return `SAT`
8: **else**
9: Use `refine` to refine $\bar{N}$ into $\bar{N}'$
10: $\bar{N} \leftarrow \bar{N}'$
11: Goto step 2
12: **end if**
13: **end if**

is sound, Lemmas 2 and 3 guarantee the soundness of Alg. 1. Further, the algorithm always terminates: this is the case because all the `abstract` steps are performed first, followed by a sequence of `refine` steps. Because no additional `abstract` operations are performed beyond Step 1, after finitely many `refine` steps $\bar{N}$ will become identical to N, at which point no spurious counterexample will be found, and the algorithm will terminate with either `SAT` or `UNSAT`. Of course, termination is only guaranteed when the underlying *Verify* procedure is guaranteed to terminate.

There are two steps in the algorithm that we intentionally left ambiguous: Step 1, where the initial over-approximation is computed, and Step 9, where the current abstraction is refined due to the discovery of a spurious counterexample. The motivation was to make Alg. 1 general, and allow it to be customized by plugging in different heuristics for performing Steps 1 and 9, that may depend on the problem at hand. Below we propose a few such heuristics.

4.1 Generating an Initial Abstraction

The most naïve way to generate the initial abstraction is to apply the `abstract` operator to saturation. As previously discussed, `abstract` can merge together any pair of hidden neurons from a given layer that share the same attributes. Since there are four possible attribute combinations, this will result in each hidden layer of the network having four neurons or fewer. However, for a reasonably large DNN, we expect this abstraction to be very coarse, and so it might lead to multiple rounds of refinement before a `SAT` or `UNSAT` answer can be reached.

A different heuristic for producing abstractions that may require fewer refinement steps is as follows. First, we select a finite set of input points, $X = \{x_1, \ldots, x_n\}$, all of which satisfy the input property P. These points can be generated randomly, or according to some coverage criterion of the input space. The points of X are then used as indicators in estimating when the abstraction has become too coarse: after every abstraction step, we check whether the property

still holds for $x_1, \ldots, x_n$, and stop abstracting if this is not the case. The exact technique appears in Alg. 2, which is used to perform Step 1 of Alg. 1.

Algorithm 2 Create Initial Abstraction(N, P, Q)

1: $\bar{N} \leftarrow N$
2: **while** $\forall x \in X.\ \bar{N}(x)$ satisfies Q and there are still neurons that can be merged **do**
3: $\quad \Delta \leftarrow \infty$, bestPair $\leftarrow \bot$
4: $\quad$ **for** every pair of hidden neurons $v_{i,j}, v_{i,k}$ with identical attributes **do**
5: $\quad\quad$ m $\leftarrow 0$
6: $\quad\quad$ **for** every node $v_{i-1,p}$ **do**
7: $\quad\quad\quad$ a $\leftarrow \bar{w}(v_{i-1,p}, v_{i,j})$, b $\leftarrow \bar{w}(v_{i-1,p}, v_{i,k})$
8: $\quad\quad\quad$ **if** $|a - b| >$ m **then**
9: $\quad\quad\quad\quad$ m $\leftarrow |a - b|$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad\quad$ **if** m $< \Delta$ **then**
13: $\quad\quad\quad \Delta \leftarrow$ m, bestPair $\leftarrow \langle v_{i,j}, v_{i,k} \rangle$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: $\quad$ Use `abstract` to merge the nodes of bestPair, store the result in $\bar{N}$
17: **end while**
18: **return** $\bar{N}$

Another point that is address by Alg. 2, besides how many rounds of abstraction should be performed, is which pair of neurons should be merged in every application of `abstract`. This, too, is determined heuristically. Since any pair of neurons that we pick will result in the same reduction in network size, our strategy is to prefer neurons that will result in a a more accurate approximation. Inaccuracies are caused by the max and min operators within the `abstract` operator: e.g., in the case of max, every pair of incoming edges with weights a, b are replaced by a single edge with weight $\max(a, b)$. Our strategy here is to merge the pair of neurons for which the *maximal* value of $|a - b|$ (over all incoming edges with weights a and b) is *minimal*. Intuitively, this leads to $\max(a, b)$ being close to both a and b — which, in turn, leads to an over-approximation network that is smaller than the original, but is close to it weight-wise. We point out that although repeatedly exploring all pairs (line 4) may appear costly, in our experiments the time cost of this step was negligible compared to that of the verification queries that followed. Still, if this step happens to become a bottleneck, it is possible to adjust the algorithm to heuristically sample just some of the pairs, and pick the best pair among those considered — without harming the algorithm's soundness.

As a small example, consider the network depicted on the left hand side of Fig. 5. This network has three pairs of neurons that can be merged using `abstract` (any subset of $\{v_1, v_2, v_3\}$). Consider the pair v_1, v_2: the maximal value of $|a - b|$ for these neurons is $\max(|1 - 4)|, |(-2) - (-1)|) = 3$. For pair v_1, v_3,

the maximal value is 1; and for pair v_2, v_3 the maximal value is 2. According to the strategy described in Alg. 2, we would first choose to apply `abstract` on the pair with the minimal maximal value, i.e. on the pair v_1, v_3.

4.2 Performing the Refinement Step

A refinement step is performed when a spurious counterexample x has been found, indicating that the abstract network is too coarse. In other words, our abstraction steps, and specifically the max and min operators that were used to select edge weights for the abstract neurons, have resulted in the abstract network's output being too great for input x, and we now need to reduce it. Thus, our refinement strategies are aimed at applying `refine` in a way that will result in a significant reduction to the abstract network's output. We note that there may be multiple options for applying `refine`, on different nodes, such that any of them would remove the spurious counterexample x from the abstract network. In addition, it is not guaranteed that it is possible to remove x with a single application of `refine`, and multiple consecutive applications may be required.

One heuristic approach, which we refer to as *weight-based refinement*, is to look for a concrete neuron v, currently mapped into an abstract neuron $\bar{v}$, such that the incoming weights of v and $\bar{v}$ differ significantly. This indicates that by mapping v into $\bar{v}$ we have performed a coarse approximation, and that splitting v away from $\bar{v}$ may restore accuracy. This heuristic is formally defined in Alg. 3. The algorithm simply traverses the original neurons, looks for the edge weight that has changed the most as a result of the current abstraction, and then performs refinement on the neuron at the end of that edge. As was the case with Alg 2, if considering all possible nodes turns out to be too costly, it is possible to adjust the algorithm to explore only some of the nodes, and pick the best one among those considered — without jeopardizing the algorithm's soundness.

Algorithm 3 Weight-Based Refinement$(N, \bar{N})$

```
bestNeuron ← ⊥, m ← 0
for each concrete neuron v_{i,j} of N mapped into abstract neuron v̄_{i,j'} of N̄ do
    for each concrete neuron v_{i−1,k} of N mapped into abstract neuron v̄_{i−1,k'} of N̄ do
        if |w(v_{i,j}, v_{i−1,k}) − w̄(v̄_{i,j'}, v̄_{i−1,k'})| > m then
            m ← |w(v_{i,j}, v_{i−1,k}) − w̄(v̄_{i,j'}, v̄_{i−1,k'})|
            bestNeuron ← v_{i,j}
        end if
    end for
end for
Use refine to split bestNeuron from its abstract neuron
```

As an example, let us use Alg. 3 to choose a refinement step for the right hand side network of Fig. 5. Suppose v_1 is considered first. In the abstract

network, $\bar{w}(x_1, \bar{v_1}) = 4$ and $\bar{w}(x_2, \bar{v_1}) = -1$; whereas in the original network, $w(x_1, v_1) = 1$ and $w(x_2, v_1) = -2$. Thus, the largest value m computed for v_1 is $|w(x_1, v_1) - \bar{w}(x_1, \bar{v_1})| = 3$. This value of m is larger than the one computed for v_2 (0) and for v_3 (2), and so v_1 is selected for the refinement step. After this step is performed, v_2 and v_3 are still mapped to a single abstract neuron, whereas v_1 is mapped to a separate neuron in the abstract network.

Weight-based refinement attempts to reduce the output value y by as much as possible, but does not take into account the counterexample x that was discovered by the verification procedure. Consider a case where abstract neuron $\bar{v}$ is selected for refinement based on its incoming weights, but where $\bar{v}$ is actually assigned value 0 when the network is evaluated on counterexample x. By performing this refinement step, we might not change the network's output at all for x. Intuitively, by focusing our refinements efforts on neurons that take on small assignments in our input region of interest, we may be ignoring other choices that could decrease the network's output more significantly.

To address this situation, we propose to change Alg. 3 in a way that would make it counterexample-guided. Specifically, we suggest to adjust lines 4 and 5 of Alg. 3, by multiplying the term $|w(v_{i,j}, v_{i-1,k}) - \bar{w}(\bar{v}_{i,j'}, \bar{v}_{i-1,k'})|$ that appears therein by the value of neuron $\bar{v}$ when the abstract network is evaluated for counterexample x. This heuristic, which we refer to as *counterexample-guided refinement*, takes into account both edge weights and the discovered counterexample, and may ignore neurons that weight-based refinement might select if these neurons are assigned small values.

5 Implementation and Evaluation

Our implementation of the abstraction-refinement framework includes modules that read a DNN in the NNet format [14] and a property to be verified, create an initial abstract DNN as described in Section 4, invoke a black-box verification engine, and perform refinement as described in Section 4. The process terminates when the underlying engine returns either `UNSAT`, or an assignment that is a true counterexample for the original network. For experimentation purposes, we integrated our framework with the Marabou DNN verification engine [17]. We will make our implementation publicly available with the final version of this paper.

Our experiments included verifying several properties of the 45 ACAS Xu DNNs for airborne collision avoidance [14,15]. ACAS Xu is a system designed to produce horizontal turning advisories for an unmanned aircraft (the *ownship*), with the purpose of preventing a collision with another nearby aircraft (the *intruder*). The ACAS Xu system receive as input sensor readings, indicating the location of the intruder relative to the ownship, the speeds of the two aircraft, and their directions (see Fig. 6). Based on these readings, it selects one of 45 DNNs, to which the readings are then passed as input. The selected DNN then assigns scores to five output neurons, each representing a possible turning advisory: strong left, weak left, strong right, weak right, or clear-of-conflict (the

latter indicating that it is safe to continue along the current trajectory). The neuron with the *lowest* score represents the selected advisory. We verified several properties of these DNNs based on the list of properties that appeared in [15] — specifically focusing on properties that ensure that the DNNs always advise clear-of-conflict for distant intruders, and that they are robust to (i.e., do not change their advisories in the presence of) small input perturbations.

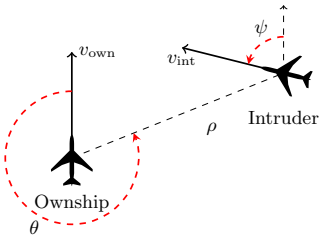


Fig. 6. (From [15]) An illustration of the sensor readings passed as input to the ACAS Xu DNNs.

Each of the ACAS Xu DNNs has 300 hidden nodes spread across 6 hidden layers, leading to 1200 neurons when the transformation from Section 3.1 is applied. In our experiments we set out to check whether the abstraction-based approach could indeed prove properties of the ACAS Xu networks on abstract networks that had significantly fewer neurons than the original ones. In addition, we wished to compare the proposed approaches for generating initial abstractions (the naïve approach versus Alg. 2) and the approaches for performing refinement steps (weight-based versus counterexample-guided), in order to identify an optimal configuration for our tool. Finally, once the optimal configuration has been identified, we used it to compare our tool's performance to that of vanilla Marabou. The results are described next.

Fig. 7 depicts a comparison of the two approaches for generating initial abstractions: the naïve abstraction scheme in which each abstract hidden layer initially has 4 neurons (x axis), and the method described in Alg. 2 in which the hidden layers are typically larger (y axis). Each experiment included running our tool twice on the same benchmark (network and property), with an identical configuration except for the initial abstraction being used. The left plot depicts the number of refinement steps required before the procedure terminated. It shows that properties can indeed be proved on abstract networks that are significantly smaller than the original — i.e., despite the initial 4x increase in network size due to the preprocessing phase, the final abstract network on which Alg. 1 could solve the query was usually substantially smaller than the original network. Among the experiments that terminated, the final network on which the property was shown to be `SAT` or `UNSAT` had an average size of 224 nodes, compared to the original 300 — a 25% reduction. The left plot also shows that, in many cases, using Alg. 2 leads to fewer refinement steps than using the

naïve scheme. The right plot indicates the total time (log-scale, in seconds, with a 20-hour timeout) spent by Marabou solving verification queries as part of the abstraction-refinement procedure. It shows that using the naïve approach usually leads to faster query solving times than using Alg. 2.

It is interesting to note that the superiority of the naïve approach, which is very clearly demonstrated by the runtime plot, is not so clearly reflected in the number-of-queries plot. By analyzing the logs, we discovered that although the number of refinement iterations required when using Alg. 2 is often smaller, the Marabou queries that it generates are typically harder: a query generated using Alg. 2 had an average solving time of 8430 seconds, versus an average solving time of 3574 when using the naïve approach. This means that although Marabou is invoked fewer times, it takes longer to run. We speculate that the reason for these queries being harder is that the naïve approach generates queries that are so coarse that (spurious) counterexamples are easier to find. We thus conclude that the naïve approach is superior to that of Alg. 2.

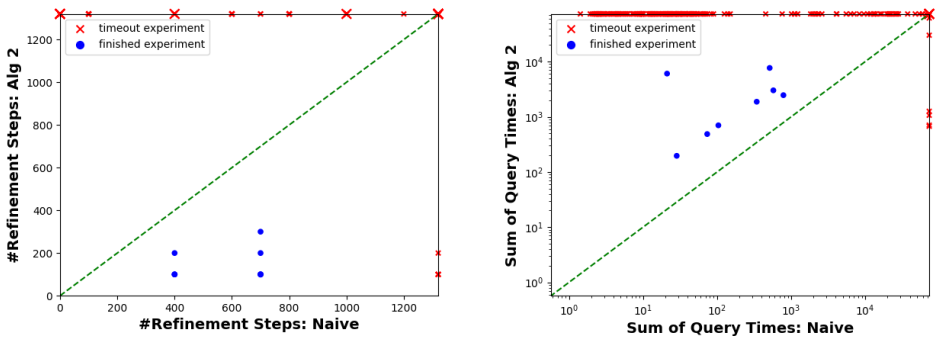


Fig. 7. Generating initial abstractions naïvely and using Alg. 2.

Next, we compared the weight-based and counterexample-guided approaches for performing refinement steps discussed in Section 4.2. To do so we again ran each benchmark through our tool twice, with identical configurations except for the refinement scheme in use. The results appear in Fig. 8. As before, we compared the number of required refinement steps (left plot) and the time required by Marabou to solve the generated queries (right plot). The results indicate the superiority of the counterexample-guided approach (x axis), especially in that it leads to far fewer timeouts.

Based on the aforementioned experiments, we chose the naïve approach for generating initial abstractions and the counterexample-guided refinement strategy as the optimal configuration of the tool for the ACAS Xu set of benchmarks. We then used this configuration in comparing our abstraction-enhanced Marabou to the vanilla version. The plot in Fig. 9 compares the total query

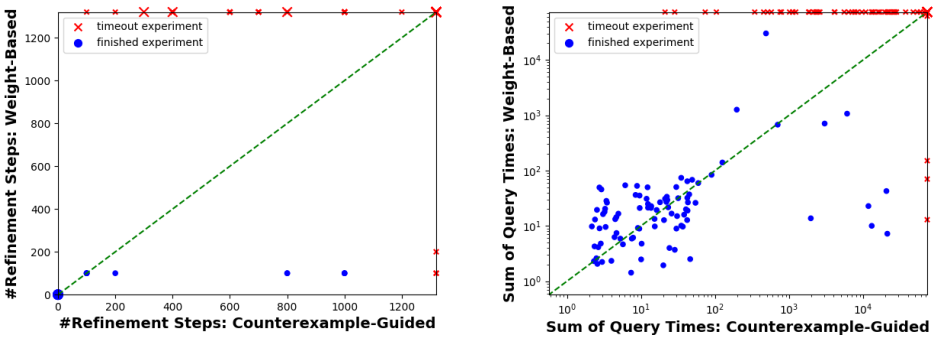


Fig. 8. Weight-based and counterexample-guided refinement steps.

solving time of vanilla Marabou (y axis) to that of our approach (x axis), on 90 difficult ACAS Xu benchmarks. We observe that the abstraction-enhanced version significantly outperforms vanilla Marabou on average — often solving queries orders-of-magnitude more quickly, and timing out on fewer benchmarks (64 timeouts for the vanilla version, versus 51 for the abstraction-enhanced version). These results clearly indicate the usefulness of combining our technique with an existing verification engine, in order to boost its performance.

Next, we used our abstraction-enhanced Marabou to verify *adversarial robustness* properties [28]. Intuitively, an adversarial robustness property states that slight input perturbations cannot cause sudden spikes in the network's output. This is desirable because such sudden spikes can lead to misclassification of inputs. Unlike the ACAS Xu domain-specific properties [15], whose formulation required input from human experts, adversarial robustness is a *universal property*, desirable for every DNN. Consequently it is easier to formulate, and has received much attention (e.g., [1,8,15,29]).

In order to formulate adversarial robustness properties for the ACAS Xu networks, we randomly sampled the ACAS Xu DNNs to identify input points where the selected output advisory, indicated by an output neuron y_i, received a much lower score than the second-best advisory, y_j (recall that the advisory with the lowest score is selected). For such an input point x_0, we then posed the verification query: does there exist a point x that is close to x_0, but for which y_j receives a lower score than y_i? Or, more formally:

$$(\|x - x_0\|_{L_\infty} \leq \delta) \wedge (y_j \leq y_i)$$

If this query is `SAT` then there exists an input x whose distance to x_0 is at most δ, but for which the network assigns a better (lower) score to advisory y_j than to y_i. However, if this query is `UNSAT`, no such point x exists. Because we select point x_0 such that y_i is initially much smaller than y_j, we expect the query to be `UNSAT` for small values of δ.

For each of the 45 ACAS Xu networks, we created robustness queries for 20 distinct input points — producing a total of 900 verification queries (we arbi-

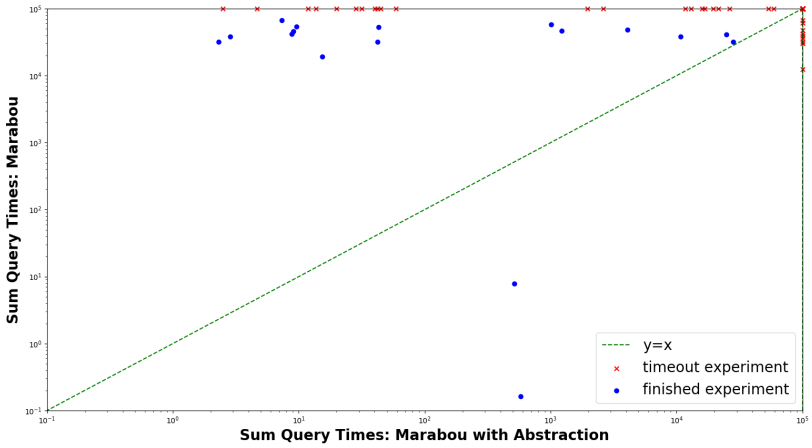


Fig. 9. Comparing the run time (in seconds, logscale) of vanilla Marabou and the abstraction-enhanced version on the ACAS Xu benchmarks.

trarily set $\delta = 0.1$). For each of these queries we compared the runtime of vanilla Marabou to that of our abstraction-enhanced version (with a 10-hour timeout). The results are depicted in Fig. 10. Again, we can see that the abstraction-enhanced version generally outperforms vanilla Marabou by a significant margin; specifically, the vanilla version timed out on 675 experiments, versus 470 timeouts for the abstraction-enhanced version.

6 Related Work

In recent years, multiple schemes have been proposed for the verification of neural networks. These include SMT-based approaches, such as Marabou [17,18], Reluplex [15], DLV [13] and others; approaches based on formulating the problem as a mixed integer linear programming instance (e.g., [7,29,3,6]); approaches with sophisticated deduction schemes such as symbolic interval propagation (the ReluVal solver [30]) or abstract interpretation (the AI2 solver [8]); and others (e.g., [21,24,31]). Our approach can be integrated with any sound and complete solver as its engine; incomplete approaches could also be used and might afford better performance, but could result in non-termination.

Some existing DNN verification techniques incorporate abstraction elements. In [25], the authors use abstraction to over-approximate the Sigmoid activation function with a collection of rectangles. If the abstract verification query they produce is `UNSAT`, then so is the original. When a spurious counterexample is found, an arbitrary refinement step is performed. The authors report limited scalability, tackling only networks with a few dozen neurons. Abstraction tech-

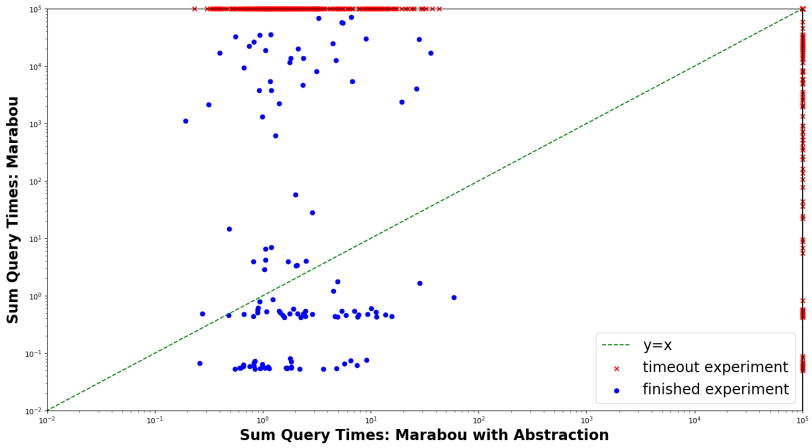


Fig. 10. Comparing the run time (seconds, logscale) of vanilla Marabou and the abstraction-enhanced version on the ACAS Xu adversarial robustness properties.

niques also appear in the AI2 approach [8], but there it is the input property and reachable regions that are over-approximated, as opposed to the DNN itself. Combining this kind of input-focused abstraction with our network-focused abstraction is an interesting avenue for future work.

7 Conclusion

With deep neural networks becoming widespread and with their forthcoming integration into safety-critical systems, there is an urgent need for scalable techniques to verify and reason about them. However, the size of these networks poses a serious challenge. Abstraction-based techniques can mitigate this difficulty, by replacing networks with smaller versions thereof to be verified, without compromising the soundness of the verification procedure. The abstraction-based approach we have proposed here can provide a significant reduction in network size, thus boosting the performance of existing verification technology.

In the future, we plan to continue this work along several axes. First, we intend to investigate refinement heuristics that can split an abstract neuron into two arbitrary sized neurons. In addition, we will investigate abstraction schemes for networks that use additional activation functions, beyond ReLUs. Finally, we plan to make our abstraction scheme parallelizable, allowing users to use multiple worker nodes to explore different combinations of abstraction and refinement steps, hopefully leading to faster convergence.

References

1. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.
2. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.
3. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and M. Kumar. Piecewise Linear Neural Network Verification: A Comparative Study, 2017. Technical Report. https://arxiv.org/abs/1711.00455v1.
4. N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. https://arxiv.org/abs/1709.10207.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2010.
6. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari. Output Range Analysis for Deep Neural Networks. In *Proc. 10th NASA Formal Methods Symposium (NFM)*, pages 121–138, 2018.
7. R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
8. T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
9. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
10. D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
11. J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. Tenenbaum, and T. Mattson. The Three Pillars of Machine Programming. In *Proc. 2nd ACM SIGPLAN Int. Workshop on Machine Learning and Programming Languages (MALP)*, pages 69–80, 2018.
12. G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
13. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
14. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
15. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.

16. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.
17. G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, 2019. To appear.
18. Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, 2019.
19. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
20. A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial Examples in the Physical World, 2016. Technical Report. http://arxiv.org/abs/1607.02533.
21. A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. https://arxiv.org/abs/1706.07351.
22. H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proc. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 197–210, 2017.
23. V. Nair and G. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th Int. Conf. on Machine Learning (ICML)*, pages 807–814, 2010.
24. N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. Verifying Properties of Binarized Deep Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1709.06662.
25. L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.
26. W. Ruan, X. Huang, and M. Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proc. 27th Int. Joing Conf. on Artificial Intelligence (IJACI)*, pages 2651–2659, 2018.
27. D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
28. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.
29. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *Proc. 7th Int. Conf. on Learning Representations (ICLR)*, 2019.
30. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.
31. W. Xiang, H.-D. Tran, and T. Johnson. Output Reachable Set Estimation and Verification for Multilayer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 99:1–7, 2018.

Simplifying Neural Networks using Formal Verification

Sumathi Gokulanathan[1], Alexander Feldsher[1], Adi Malca[1], Clark Barrett[2], and Guy Katz[1(✉)]

[1] The Hebrew University of Jerusalem, Israel
{sumathi.giokolanat, feld, adimalca, guykatz}@cs.huji.ac.il
[2] barrett@cs.stanford.edu

Abstract. Deep neural network (DNN) verification is an emerging field, with diverse verification engines quickly becoming available. Demonstrating the effectiveness of these engines on real-world DNNs is an important step towards their wider adoption. We present a tool that can leverage existing verification engines in performing a novel application: neural network simplification, through the reduction of the size of a DNN without harming its accuracy. We report on the work-flow of the simplification process, and demonstrate its potential significance and applicability on a family of real-world DNNs for aircraft collision avoidance, whose sizes we were able to reduce by as much as 10%.

Keywords: Deep Neural Networks, Simplification, Verification, Marabou

1 Introduction

Deep neural networks (*DNNs*) are revolutionizing the way complex software is produced, obtaining unprecedented results in domains such as image recognition [28], natural language processing [5], and game playing [27]. There is now even a trend of using DNNs as controllers in autonomous cars and unmanned aircraft [2,18]. With DNNs becoming prevalent, it is highly important to develop automatic techniques to assist in creating, maintaining and adjusting them.

As DNNs are used in tackling increasingly complex tasks, their sizes (i.e., number of neurons) are also increasing — to a point where modern DNNs can have millions of neurons [13]. DNN size is thus becoming a liability, as deploying larger networks takes up more space, increases energy consumption, and prolongs response times. Network size can even become a limiting factor in situations where system resources are scarce. For example, consider the ACAS Xu airborne collision avoidance system for unmanned aircraft, which is currently being developed by the Federal Aviation Administration [18]. This is a highly safety-critical system, for which a DNN-based implementation is being considered [18]. Because this system will be mounted on actual drones with limited memory, efforts are being made to reduce the sizes of the ACAS Xu DNNs as much as possible, without harming their accuracy [17, 18].

Most work to date on DNN simplification uses various heuristics, and does not provide formal guarantees about the simplified network's resemblance to the original. A common approach is to start with a large network, and reduce its size by removing some of its components (i.e., neurons and edges) [12, 15]. The parts to be removed from the network are determined heuristically, and network accuracy may be harmed, sometimes requiring additional training after the simplification process has been performed [12].

Here, we propose a novel simplification technique that harnesses recent advances in DNN verification (e.g., [9,19,32]). Using verification queries, we propose to identify components of the network that *never affect its output*. These components can be safely removed, creating a smaller network that is completely equivalent to the original. We empirically demonstrate that many such removable components exist in networks of interest.

We implement our technique in a proof-of-concept tool, called *NNSimplify*. The tool uses the following work-flow: (i) it performs lightweight simulations to identify parts of the DNN that are candidates for removal; (ii) it invokes an underlying verification engine to dispatch queries that determine which of those parts can indeed be removed without affecting the network's outputs; and (iii) it constructs the simplified network, which is equivalent to the original. A major benefit of the proposed verification-based simplification is that it does not require any retraining of the simplified network, which may be expensive.

Our implementation of NNSimplify (available online [10]) can use existing DNN verification tools as a backend. For the evaluation reported here, we used the recently published Marabou framework [21] as the underlying verification engine. We evaluated our approach on the ACAS Xu family of DNNs for airborne collision avoidance [18], and were able to reduce the sizes of these DNNs by up to 10% — a highly significant reduction for systems where resources are scarce.

The rest of the paper is organized as follows. In Section 2, we provide a brief background on DNNs and their verification and simplification. Next, we describe our verification-based approach to simplification in Section 3, followed by an evaluation in Section 4. We then conclude in Section 5.

2 Background: DNNs, Verification and Simplification

DNNs are comprised of an input layer, an output layer, and multiple hidden layers in between. A layer is comprised of multiple nodes (neurons), each connected to nodes from the preceding layer using a predetermined set of weights (see Fig. 1). By assigning values to inputs and then feeding them forward through the network, values for each layer can be computed from the values of the previous layer, finally resulting in values for the outputs.

As DNNs are increasingly used in safety-critical applications (e.g., [2, 18]), there is a surge of interest in verification methods that can provide formal guarantees about DNN behavior. A DNN verification query consists of a neural network and a property to be checked; and it results in either a formal guarantee that the network satisfies the property, or a concrete input for which the property is

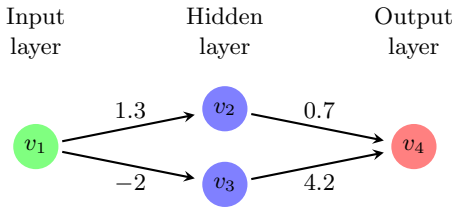


Fig. 1: A small neural network with 2 hidden nodes in one hidden layer. Weights are denoted over the edges. Hidden node values are typically determined by computing a weighted sum according to the weights, and then applying a non-linear activation function to the result.

violated (a counter-example). Verification queries can encode various properties about DNNs; e.g., that slight perturbations to a network's inputs do not affect its output, and that it is thus robust to *adversarial perturbations* [1, 4, 30].

Recently, there has been significant progress on DNN verification tools that can dispatch such queries (see a recent survey [24]). Some of the proposed approaches for DNN verification include the use of specialized SMT solvers [14, 19, 21], the use of LP and MILP solvers [7, 31], symbolic interval propagation [32], abstract interpretation [9], and many others (e.g., [3, 6, 8, 25, 26]). This new technology has been applied in a variety of contexts, such as collision avoidance [19], adversarial robustness [11, 14, 20], hybrid systems [29], and computer networks [22]. Although DNN verification technology is improving rapidly, scalability remains a major limitation of existing approaches. It has been shown that a common variant of the DNN verification problem is NP-complete, and becomes exponentially harder as the network size increases [19, 23].

In recent years, enormous DNNs have been appearing in order to tackle increasingly complex tasks — to a point where DNN size is becoming a liability, because large networks take longer to train and even to evaluate when deployed. Techniques for neural network minimization and simplification have thus started to emerge: typically, these take an initial, large network, and reduce its size by removing some of its components [12]. The pruning phase involves the removal of edges from the network. The selection of which edges to remove is done heuristically, often by selecting edges that have very small weights, because these edges are less likely to significantly affect the network's outputs. If all edges connecting a node to the preceding layer or to the succeeding layer are removed, then the node itself can be removed. After the pruning phase, the reduced network is retrained [12, 15].

3 Simplification using Verification

Despite the demonstrated usefulness of pruning-based DNN simplification [12, 15], heuristic-based approaches might miss removable edges, if these edges do

not have particularly small weights. However, such edges can be identified using verification. For example, consider the network shown in Fig. 2. As all edge weights have identical magnitudes, none of them would be pruned by a heuristic-based approach. However, using a verification engine, it is possible to check the property: "does there exist an input for which v_4 takes a non-zero value?". If the verification tool answers "no", as is the case for the network in Fig. 2 (because $v_4 = v_2 - v_3$ and $v_2 = v_3$), then we are guaranteed that v_4 is always assigned 0, regardless of the input. In turn, this means that v_4 can never affect nodes in subsequent layers. In this case, v_4 and all its edges can be safely removed from the network (rendering the network's output constant). Due to the soundness of the verification process, we are guaranteed that the simplified DNN is completely equivalent to the original DNN, and thus no retraining is required.

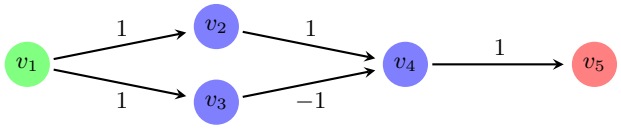


Fig. 2: Using verification, we can discover that node v_4 can safely be removed from the network.

Using verification to identify nodes that are always assigned 0 for every possible input, and can thus be removed, is the core of our technique. However, because verification is costly, posing this query for every node of the DNN might take a long time. To mitigate this difficulty, we propose the following work-flow:

1. Use lightweight simulations to identify nodes that are candidates for removal. Initially, all hidden nodes are such candidates. We then evaluate the network for random input values, and remove from the list of candidates any hidden node that is assigned a non-zero value for some input. With each simulation, the number of candidates for removal decreases.
2. For each remaining candidate node v, we create a separate verification query stating that $v \neq 0$, and use the underlying verification engine to dispatch it. If we get an `UNSAT` answer, we mark node v for removal. The candidates are explored in a layer-by-layer order, which allows us to only examine a part of the DNN for every query. For example, when addressing a candidate in layer #2, we do not encode layers #3 and on as part of our verification query, as a node's assignment can only be affected by nodes in preceding layers. Because verifying smaller networks is generally easier, this layer-by-layer approach accelerates the process as a whole. In addition, this process naturally lends itself to parallelization, by running each verification query on a separate machine.
3. Finally, we construct the simplified network, in which the nodes marked for removal and all their incoming and outgoing edges are deleted. We can also

remove any nodes that subsequently become irrelevant due to the removal of all of their incoming or outgoing edges (e.g., for the DNN in Fig. 2, after removing v_4 we can also remove v_2 and v_3, as neither has any remaining outgoing edges).

We note that our technique can be extended to simplify DNNs in additional ways, by using different verification queries. For example, it can identify separate nodes that are always assigned identical, non-zero values (duplicates) and unify them, thus reducing the overall number of nodes. It can also identify and remove nodes that can be expressed as linear combinations of other nodes.

4 Evaluation

Our proof-of-concept implementation of the approach, called NNSimplify, is comprised of three Python modules, one for performing each of the aforementioned steps. The tool is general, in two ways: (1) it can be applied to simplify any DNN, regardless of its application domain; and (2) it can use any DNN verification engine as a backend, benefiting from any future improvement in verification technology. For our experiments we used the Marabou [21] verification engine. In practice, it is required that the DNN in question be supported by the backend verification engine — for example, some engines may not support certain network topologies. Additionally, the DNN needs to be provided in a format supported by NNSimplify; currently, the tool supports the NNet format [16], and we plan to extend it to additional formats. The tool, additional documentation, and all the benchmarks reported in this section are available online [10].

We evaluated NNSimplify on the ACAS Xu family of DNNs for airborne collision avoidance [18]. This set contains 45 DNNs, each with 5 input neurons, 5 output neurons, and 300 hidden neurons spread across 6 hidden layers. The ACAS Xu networks are fully connected, and use the ReLU activation function in each of their hidden nodes — and are thus supported by Marabou.

For each of the 45 ACAS Xu DNNs, we ran the first Python module of NNSimplify (random simulations), resulting in a list of candidate nodes for removal. For each DNN we performed 20000 simulations, and this narrowed down the list of nodes that are candidates for removal to about 7% of all hidden nodes (see Fig. 3). The simulations were performed on points sampled uniformly at random, although other distributions could of course be used.

Next, for each candidate for removal we ran the second Python module, which takes as input a DNN and a node v that is a candidate for removal. This module constructs a temporary, smaller DNN, where the candidate node v is the only output node (subsequent layers are omitted). These temporary DNNs were then passed to the underlying verification engine, with the query $v \neq 0$. Here, we encountered the following issue: the Marabou framework, like many linear-programming based tools, does not provide a way to directly specify that $v \neq 0$, but rather only to state that $v \geq \varepsilon$ for some $\varepsilon > 0$ (we assume all hidden nodes are, by definition, never negative, which is the case for the ACAS Xu DNNs).

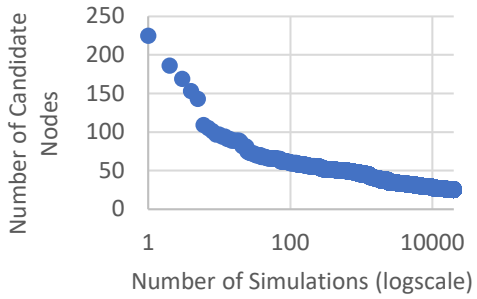


Fig. 3: Using simulation to identify nodes that are candidates for removal, on one ACAS Xu network.

We experimented with various values of ε (see Fig. 4), and concluded that the choice of ε has very little effect on the outcome of the experiment — i.e., nodes tend to either be obsolete, or take on large values. The set of removed nodes was almost identical in all experiments, with minor differences due to different queries timing out for different values of ε.

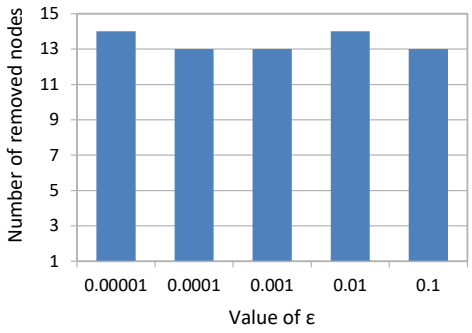


Fig. 4: Number of removed nodes as a function of the value of ε, on one of the ACAS Xu networks.

Finally, we ran the third Python module that uses the results of the previous steps to construct the simplified network.

We performed this process for each of the 45 DNNs. We ran the experiments on machines with Intel Xeon E5-2670 CPUs (2.60GHz) and 8GB of memory, and used $\varepsilon = 0.01$. Each verification query was given a 4-hour timeout. Out of 1069 verification queries (1 per candidate node), 535 were `UNSAT` (node marked for

removal), 15 were SAT, and 519 timed out (node not marked for removal). Thus, on average, 4% of the nodes were marked for removal (535 nodes out of 13500). Fig. 5 depicts their distribution across the 45 DNNs. In most networks, between 11 and 15 nodes (out of 300) could be removed; but for a few networks, this number was higher. For one of the networks we discovered 29 neurons that could be removed — approximately 10% of that network's total number of neurons.

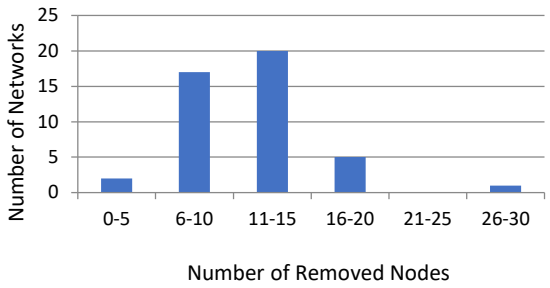


Fig. 5: Total number of removed nodes in the ACAS Xu networks.

5 Conclusion

DNN verification is an emerging field, and we are just now beginning to tap its potential in assisting engineers in DNN development. We presented here the NNSimplify tool, which uses black-box verification engines to simplify neural networks. We demonstrated that this approach can lead to a substantial reduction in DNN size. Although our experiments show that the tool is already applicable to real-world DNNs, its scalability is limited by the scalability of its underlying verification engine; but as the scalability of verification technology improves, that limitation will diminish. In the future, we plan to extend this work along several axes. First, we intend to explore additional verification queries, which would allow to simplify DNNs in more sophisticated ways — for example by revealing that some neurons can be expressed as linear combinations of other neurons, or that some neurons are always assigned identical values and can be merged. In addition, we plan to investigate more aggressive simplification steps, which may change the DNN's output, while using verification to ensure that these changes remain within acceptable bounds. Finally, we intend to apply the technique to additional real-world DNNs and case studies.

Acknowledgements. This project was partially supported by grants from the Binational Science Foundation (2017662), the Israel Science Foundation (683/18), and the National Science Foundation (1814369).

References

1. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.
2. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.
3. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and M. Kumar. Piecewise Linear Neural Network Verification: A Comparative Study, 2017. Technical Report. https://arxiv.org/abs/1711.00455v1.
4. N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. https://arxiv.org/abs/1709.10207.
5. R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research (JMLR)*, 12:2493–2537, 2011.
6. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari. Output Range Analysis for Deep Neural Networks. In *Proc. 10th NASA Formal Methods Symposium (NFM)*, pages 121–138, 2018.
7. R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
8. Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification, 2019. Technical Report. http://arxiv.org/abs/1910.14574.
9. T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
10. S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. The NNSimplify Code, 2020. https://drive.google.com/open?id=19TbPS7P9fo-2tRXo8ENnggLY1LxxPCd1.
11. D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
12. S. Han, H. Mao, and W. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, 2015. Technical Report. http://arxiv.org/abs/1510.00149.
13. A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, 2017. Technical Report. http://arxiv.org/abs/1704.04861.
14. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
15. F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and $<$ 0.5MB Model Size, 2016. Technical Report. http://arxiv.org/abs/1602.07360.

16. K. Julian. NNet Format, 2018. https://github.com/sisl/NNet.
17. K. Julian, M. Kochenderfer, and M. Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.
18. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
19. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
20. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.
21. G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
22. Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 2019.
23. L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. https://arxiv.org/abs/1801.05950.
24. C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2019. Technical Report. http://arxiv.org/abs/1903.06758.
25. A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1706.07351.
26. N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. Verifying Properties of Binarized Deep Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1709.06662.
27. D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
28. K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. http://arxiv.org/abs/1409.1556.
29. X. Sun, H. Khedr, and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, pages 147–156, 2019.
30. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.
31. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *Proc. 7th Int. Conf. on Learning Representations (ICLR)*, 2019.
32. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals, 2018. Technical Report. http://arxiv.org/abs/1804.10829.

NeVer 2.0: Learning, Verification and Repair of Deep Neural Networks

Dario Guidotti[1], Luca Pulina[2], and Armando Tacchella[1]

[1] University of Genoa, Italy
[2] University of Sassari, Italy
dario.guidotti@edu.unige.it,
lpulina@uniss.it, armando.tacchella@unige.it

Abstract. In this work we present an early prototype of NeVer 2.0, a new system for automated synthesis and analysis of deep neural networks. NeVer 2.0 borrows its design philosophy from NeVer, the first package that integrated learning, automated verification and repair of (shallow) neural networks in a single tool. The goal of NeVer 2.0 is to provide a similar integration for deep networks by leveraging a selection of state-of-the-art learning frameworks and integrating them with verification algorithms to ease the scalability challenge and make repair of faulty networks possible.

Keywords: Deep Neural Networks · Network Pruning · Network Verification.

1 Introduction

Adoption and successful application of deep neural networks (DNNs) in various domains have made them one of the most popular machine-learned models to date — see, e.g., [27] on image classification, [35] on speech recognition, and [15] for the general principles and a catalog of success stories. Despite the impressive progress that the learning community has made with the adoption of DNNs, it is well known that their application in safety- or security-sensitive contexts is not yet hassle-free. From their well-known sensitivity to *adversarial perturbations* [26, 6], i.e., minimal changes to correctly classified input data that cause a network to respond in unexpected and incorrect ways, to other less-investigated, but possibly significant properties — see, e.g., [18] for a catalog — the need for tools to analyze and possibly repair DNNs is strong.

As witnessed by an extensive survey [10] of more than 200 recent papers, the response from the scientific community has been equally strong. As a result, many algorithms have been proposed for verification of neural networks and tools implementing them have been made available. Some examples of well-known and fairly mature verification tools are Marabou [13], an SMT-based tool that answers queries regarding the properties of a DNN by transforming the queries into constraint satisfiability problems; ERAN [25], a robustness analyzer based on abstract interpretation and MIPVerify [28], another robustness analyzer based

on mixed integer programming (MIP). Other widely-known verification tools are Neurify [31], a robustness analyzer based on symbolic interval analysis and linear relaxation, NNV [30], a tool implementing different methods for reachability analysis, Sherlock [4], an output range analysis tool and NSVerify [2], also for reachability analysis. A number of verification methodologies — without a corresponding tool — is also available like [32], a game based methodology for evaluating pointwise robustness of neural networks in safety-critical applications. Most of the above-mentioned tools and methodologies work only for feedforward fully-connected neural networks with ReLU activation functions, with some of them featuring verification algorithms for convolutional neural networks with different kinds of activation function. To the best of our knowledge, current state-of-the-art tools are restricted to verification/analysis tasks, in some cases they are limited to specific network architectures and they might prove difficult to use for the non-initiated.

In this work we present an early prototype of NeVer 2.0, a new system that aims to bridge the gap between learning and verification of DNNs and solve some of the above mentioned issues. NeVer 2.0 borrows its design philosophy from NeVer [22], the first tool for automated learning, analysis and repair of neural networks. NeVer was designed to deal with multilayer perceptrons (MLPs) and its core was an abstraction-refinement mechanism described in [21, 23]. As a system, one peculiar aspect of NeVer was that it included learning capabilities through the shark [11] library. Concerning the verification part, NeVer could utilize any solver integrating Boolean reasoning and linear arithmetic constraint solving — HySAT [5] at the time. A further peculiarity of the approach was that NeVer could leverage abstract counterexamples to (try to) repair the MLP, i.e., retrain it to eliminate the causes of misbehaviour.

Our goal for NeVer 2.0 is to provide the same features of NeVer, but in an updated package that has the following features:

- Loading of datasets, trained and untrained models provided in a variety of formats; currently NeVer 2.0 supports directly popular datasets, e.g., MNIST [16] and Fashion MNIST [33], but support for further datasets can be added through a common interface; models (either trained or not) can be supplied to NeVer 2.0 using ONNX[3] and PyTorch[4] [20] formats — TensorFlow[5] [1] support is under development.
- Training of DNNs through state-of-the-art frameworks; currently NeVer 2.0 is based on PyTorch, but further extensions are planned to handle different kinds of learning models (e.g., kernel-based machines) that are not handled natively by PyTorch, or to leverage specific capabilities of other learning frameworks.
- Manipulation of DNNs including, but not limited to, pruning [24], quantization [34], and transfer learning [29]; currently NeVer 2.0 builds on Py-

[3] https://onnx.ai/
[4] https://pytorch.org/
[5] https://www.tensorflow.org/

Torch to manipulate DNNs, and only two mainstream pruning techniques are implemented, namely network slimming [24] and weight pruning [17].
- Verification of DNNs: currently, NeVer 2.0 leverages external tools as backends to provide verification capabilities; connectors to Marabou [13], ERAN [25] and MIPVerify [28] are currently implemented; we plan to add abstraction-refinement algorithms that improve on and extend those available in NeVer, but their development is still underway.
- Repair of DNNs should enable the results of verification to improve on the results of learning; currently, NeVer 2.0 features the same mechanism of NeVer, i.e., it relies on the capability of the embedded learning algorithms to exploit counterexamples and retrain the network in a better way — a sort of adversarial training guided by verification; we expect to reach tighter integration of verification and learning once our custom verification algorithms are implemented.

The version of NeVer 2.0 corresponding to this work is available online [8] under the Commons Clause (GNU GPL v3.0) license.
The rest of the paper is structured as follows. In Section 2 we introduce some basic notations and definition to be used through the paper. In Section 3 we describe the architecture and the current implementation of NeVer 2.0. In Section 4 we show some early results obtained with NeVer 2.0 prototype using MNIST datasets to learn and verify fully-connected ReLU networks. We conclude the paper with and our future research agenda in Section 5.

2 Preliminaries

A *neural network* is a system of interconnected computing units called *neurons*. In *fully connected feed-forward* networks, neurons are arranged in disjoint layers, with each layer being fully connected only with the next one, and without connection between neurons in the same layer. Given a feed-forward neural network N with n layers, we denote the i-th layer of N as $\mathbf{h}^{(i)}$. We call a layer without incoming connections *input* layer $\mathbf{h}^{(1)}$, a layer without outgoing connections *output* layer $\mathbf{h}^{(n)}$, while all other layers are referred to as *hidden* layers . Each hidden layer performs specific transformations on the inputs it receives. In this work we consider hidden layers that make use of *linear* and *batch normalization* modules. Given an input vector $\mathbf{x}$, a linear module computes a linear combination of its values as follows:

$$\mathbf{L}^{(i)} = \mathbf{W}^{(i)} \cdot \mathbf{x} + \mathbf{b}^{(i)} \tag{1}$$

where $\mathbf{W}^{(i)}$ is the matrix of weights and $\mathbf{b}^{(i)}$ is the vector of the biases associated with the linear module in the i-th layer and $\mathbf{L}^{(i)}$ is the corresponding output. Entries of both $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ are learned parameters. In our target architectures, each linear module is followed by a batch normalization module. This is done to address the so-called *internal covariate shift* problem, i.e., the change

of the distribution of each layer's input during training [12]. The mathematical formulation of batch normalization layers can be expressed as

$$\mathbf{BN}^{(i)} = \frac{\boldsymbol{\gamma}^{(i)}}{\sqrt[2]{\boldsymbol{\sigma}^{(i)} + \epsilon^{(i)}}} \odot (\mathbf{L}^{(i)} - \boldsymbol{\mu}^{(i)}) + \boldsymbol{\beta}^{(i)} \tag{2}$$

All the operators in this equation are element-wise operators: in particular $\odot$ and the fractional symbol represent respectively the Hadamard product and division. $\mathbf{BN}^{(i)}$ and $\mathbf{L}^{(i)}$ are the output and the input vectors of the module, respectively. $\boldsymbol{\gamma}^{(i)}$, $\boldsymbol{\mu}^{(i)}$, $\boldsymbol{\sigma}^{(i)}$, $\boldsymbol{\beta}^{(i)}$ are vectors, whereas $\epsilon^{(i)}$ is a scalar value. These are learned parameters of the batch normalization layer. In particular $\boldsymbol{\mu}^{(i)}$ and $\boldsymbol{\sigma}^{(i)}$ are the estimated mean and variance of the inputs computed during training.
Finally, the output of hidden layer i is computed as $\mathbf{h}^{(i)} = \Phi^{(i)}(\mathbf{BN}^{(i)})$, where $\Phi^{(i)}$ is the *activation function* associated to the neurons in the layer. We consider only networks utilizing *Rectified Linear Unit (ReLU)* activation functions, i.e., $\Phi^{(i)} = \max(0, \mathbf{BN}^{(i)})$. Given an input vector $\mathbf{x}$, the network N computes an output vector $\mathbf{y}$ by means of the following computations

$$\begin{aligned} &\mathbf{h}^{(1)} = \mathbf{x} \\ &\mathbf{h}^{(i)} = \Phi^{(i)}(\mathbf{BN}^{(i)}(\mathbf{L}^{(i)}(\mathbf{h}^{(i-1)}))) \quad i = 2, \ldots, n-1 \\ &\mathbf{y} = \mathbf{h}^{(n)} = \mathbf{L}(\mathbf{h}^{(n-1)}) \end{aligned} \tag{3}$$

A neural network can be considered as a non-linear function $f_{\mathbf{w}} : \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ is the input space of the network, $\mathcal{Y}$ is the output space and $\mathbf{w}$ is the vector representing the weights of all the connections. We consider neural network applied to classification of d-dimensional vectors of real numbers, i.e., $\mathcal{X} \subseteq \mathbb{R}^d$ and $\mathcal{Y} \subseteq \mathbb{R}^m$, where d is the dimension of the input vector and m is the dimension of the output vector and thus also the number of possible classes of interest. We assume that given an input sample $\mathbf{x}$ the output vector $f_{\mathbf{w}}(\mathbf{x})$ contains the likelihood that $\mathbf{x}$ belongs to one of the m classes. The specific class can be computed as

$$\underset{c \in \{1,\ldots,m\}}{\arg\max} \, (f_{\mathbf{w}}(\mathbf{x}))_c$$

where $(f_{\mathbf{w}}(\mathbf{x}))_c$ denotes the c-th element of $f_{\mathbf{w}}$. Training of (deep) neural networks poses substantial computational challenges since for state-of-the-art models the size of $\mathbf{w}$ can be in the order of millions. As in any machine learning task, training must select weights to maximize the likelihood that the network responds correctly, i.e., if the input $\mathbf{x}$ is of class k, the chance of *misclassification* should be as small as possible, where misclassification occurs whenever the following holds

$$\underset{c \in \{1,\ldots,m\}}{\arg\max} \, (f_{\mathbf{w}}(\mathbf{x}))_c \neq k$$

Training can be achieved through minimization of some kind of *loss function* whose value is low when the chance of misclassification is also low. While there

are many different kinds of loss functions, in general they are structured in the following way:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{k=0}^{n} Err(y_k, \underset{c\in\{1,\dots,m\}}{\arg\max}\,(f_{\mathbf{w}}(\mathbf{x}_k))_c) + \lambda \cdot Reg(\mathbf{w}) \tag{4}$$

where n is the number of training pairs $(\mathbf{x}_k, y_k)$, y_k is the correct class label of $\mathbf{x}_k$, Err represents the loss caused by misclassification, Reg is a *regularization* function, and λ is the parameter controlling the effect of Reg on J. The regularization function is needed to avoid *overfitting*, i.e., the high variance of the training results with respect to the training data. The regularization function usually penalizes models with high complexity by smoothing out sharp variations induced in the trained network by the Err function. A common regularization function is, for example, the L2 norm:

$$Reg(\mathbf{w}) = \frac{1}{2n}||\mathbf{w}||_2 \tag{5}$$

3 System architecture and implementation

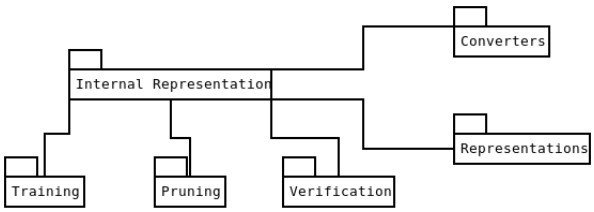


Fig. 1. High level UML diagram of NeVer 2.0.

NeVer 2.0 is conceived as a modular API to manage DNNs, from training to verification and repair. In Figure 1 we present an overview of the architecture divided in six main packages. The main elements we consider in this work are training, pruning and verification: these packages are organized mostly around *Strategy* patterns that define general interfaces to perform network operations, and specialized subclasses that actually support those operations. Additionally, to have full control of the internal model and to separate the main elements from implementation details, we designed our own network representation structured as a graph whose nodes correspond to disjoint layers. To leverage the capabilities of current learning frameworks, we designed a set of conversion strategies to/from our internal representation and the representations whereon learning frameworks are based. The aims and the internal structures of the packages shown in Figure 1 are described in detail in the remainder of this Section.

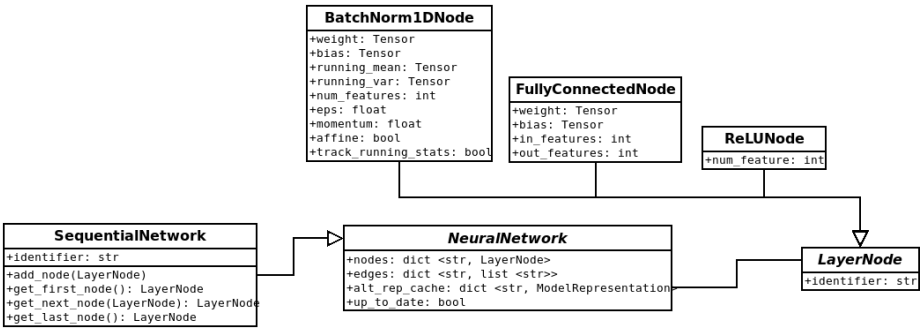


Fig. 2. UML diagram of internal network representation.

3.1 Internal Representation

The classes supporting the internal representation are shown in Figure 2. There are two abstract base classes, namely *NeuralNetwork* and *LayerNode*. Conceptually, *NeuralNetwork* is a container of *LayerNode* objects orgnaized inside *NeuralNetwork* as a graph. A list of *ModelRepresentation* objects is kept for internal usage — see subsection 3.2 for details. In the current prototype the only concrete subclass of *NeuralNetwork* is SequentialNetwork which represents networks whose corresponding graph is a list, i.e., each layer is connected only to the next one. More complex topologies for concrete architectures can be implemented, should the need arise. The concrete subclasses of *LayerNode* are the building blocks that we currently support: BatchNorm1DNode, FullyConnectedNode and ReLUNode, i.e., batch normalization layers, fully connected layers and ReLU layers, respectively. These building blocks are sufficient to encode the DNNs that we introduced in Section 2. It should be noted that our representation is not an "executable" representation, i.e., it does not provide the capability to compute the output of a DNN given the input, therefore our nodes contain only enough information to create the corresponding executable representations in different learning frameworks and/or support the encoding for verification purposes. The class *Tensor* is our utility class for tensorial data. Currently it is simply an alias for the `ndarray` class in `numpy`, but we have added it as a wrapper to isolate NeVer 2.0 classes from implementation details.

3.2 Converters and Representations

The design of a model representation to generalize those used in different learning frameworks is based on the *Adapter* design pattern, as shown in Figure 3. We have defined the abstract class *ModelRepresentation*. which is then specialized by PyTorchNetwork and ONNXNetwork to encode PyTorch and ONNX models, respectively. The concrete subclasses wrap the actual network model in the corresponding learning framework or interchange format, as in the case of ONNX. Conversion between our internal representation and the concrete subclasses of

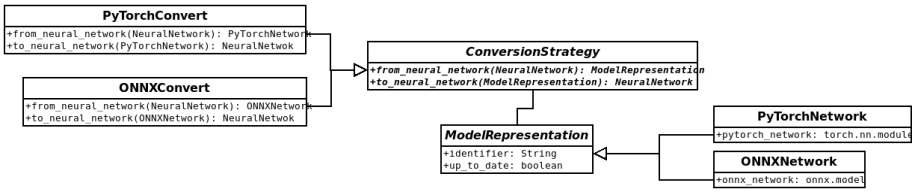


Fig. 3. UML diagram of the classes related to the representations and converters of the different learning frameworks.

ModelRepresentation are provided by subclasses of *ConversionStrategy* — we may consider this also as a *Builder* pattern implementation. *ConversionStrategy* defines an interface with two functions: one for converting from our internal representation to a specific model representation, and the other for performing the inverse task. The concrete subclasses of *ConversionStrategy* implement the functions for the corresponding concrete subclasses of *ModelRepresentation*. As new type of learning frameworks/model are added to NeVer 2.0, new concrete subclasses of *ModelRepresentation* should be added to support conversion.

3.3 Training

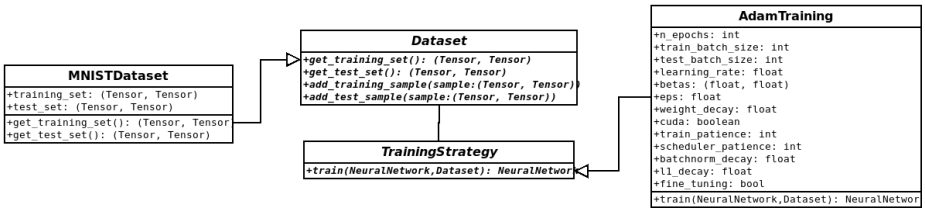


Fig. 4. UML diagram of the classes related to the training strategies.

In Figure 4 we show the internal design of the Training package whose main element is the abstract class *TrainingStrategy*. The current abstraction of a training strategy features a single function which requires a *NeuralNetwork* and a *Dataset* and returns a (trained) *NeuralNetwork*. The concrete subclasses of *TrainingStrategy* provide the actual training procedures. Currently, we have designed and implemented a single training procedure based on the Adam optimizer [14] and adapted to the concrete pruning procedures we have implemented. Our implementation requires a PyTorch representation to train the network, but this is handled transparently by NeVer 2.0 architecture. The class *Dataset* is meant to represent a generic dataset. As such it features four functions: one for loading the training set — the set of data considered to train the network — one for loading the test set – the set of data considered to assess the accuracy of the

network and two for adding a data sample to the training set and to the test set respectively. The actual datasets are represented by concrete subclasses of *Dataset*. Currently, we have implemented the corresponding concrete class for the MNIST dataset MNISTDataset and for the FMNIST dataset FMNISTDataset.

3.4 Pruning

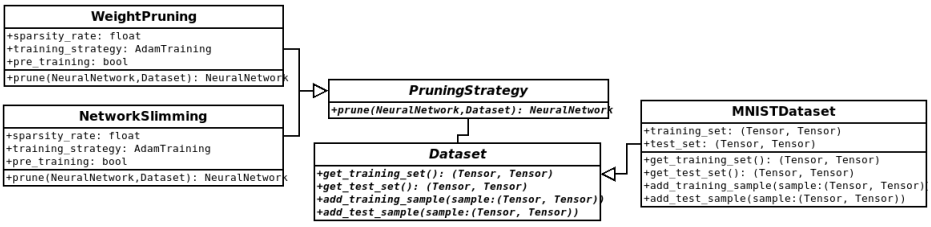


Fig. 5. UML diagram of the classes related to the pruning strategies.

As mentioned in our paper [7], we believe that pruning can be one of the keys to ease the verification of DNNs, therefore we decided to include abstractions and concrete classes to support pruning in the current realization of NeVer 2.0. In Figure 5 we show the architecture, where the abstract class *PruningStrategy* is meant to represent a generic pruning methodology, and consists of a single function which requires a *NeuralNetwork* and a *Dataset* and returns the pruned *NeuralNetwork*. Concrete subclasses implement the actual pruning procedures: currently we have designed and implemented two concrete strategies, namely WeightPruning and NetworkSlimming — both based on PyTorch representations. In particular, the former strategy selects all the weights which are smaller than a certain threshold and sets them to 0. The latter strategy leverages the weights of the batch normalization layers to identify low-importance neurons and remove them from the network — more details can be found in [9] and [19]. The distinctive parameters of the pruning strategies are provided as attributes in the concrete classes. In particular, if pre-training and/or fine-tuning are required for the pruning procedure then a suitable training strategy must be provided to the pruning strategy as an attribute.

3.5 Verification

As shown in Figure 6, we have designed the abstract class *VerificationStrategy* to represent a generic verification methodology. This abstract class defines an interface consisting of a single function which requires a *NeuralNetwork* and a *Property* and returns a Boolean value depending on whether the property is verified or not and a counterxample (if available). The abstract class *Property* represents a generic property that should be verified. Currently we have two

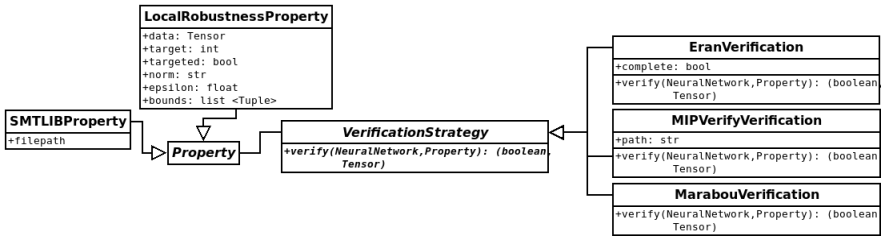


Fig. 6. UML diagram of the classes related to the verification strategies.

concrete classes: SMTLIBProperty and LocalRobustnessProperty. SMTLIBProperty represent a generic property which NeVer 2.0 reads from a file formatted according to SMTLIB [6] syntax [3]. LocalRobustnessProperty is a "pre-cooked" property encoding the search of an adversarial example corresponding to a specific data sample. The concrete subclasses of *VerificationStrategy* that we have implemented so far are EranVerification, MarabouVerification and MIPVerifyVerification which leverage, respectively, ERAN [25], Marabou [13] and MIPVerify to verify the property.

4 Preliminary experimental analysis

We test the current capabilities of NeVer 2.0 by replicating the setup of the experiment reported in [7]. In this experiment we analyze how the integration of pruning and verification can ease analysis of DNNs — currently, a distinctive capability that NeVer 2.0 offers. We test two different network architectures and two different pruning methods considering all three verification backends available in NeVer 2.0. The DNNs we consider are both fully connected networks with three hidden layers: one with 64, 32, 16 hidden neurons, and the other with 128, 64, 32 hidden neurons. In both networks the activation function is the ReLU. We experimented with weight pruning (based on [9]) and network slimming (based on [19]). To analyze the performances of the different pruning methods we test them with three different sets of pruning parameters which correspond to increasing magnitudes of pruning. The results of our experiment are summarized in Table 1. We consider three versions of each DNN: the version before pruning (baseline), the one obtained after a specialized training for network slimming (sparse), the version after the application of weight pruning (WP) and the one obtained after network slimming (NS). The results of our experiment prove that NeVer 2.0 — albeit still at the prototypical stage – is ready to verify networks of some practical interest, and its combination of pruning and verification may offer some advantage over the straight usage of its backends.

[6] http://smtlib.cs.uiowa.edu/

			MNIST			FMNIST		
Base	**Param**	**Network**	**Marabou**	**MIPVerify**	**ERAN**	**Marabou**	**MIPVerify**	**ERAN**
NET1		Baseline	0	0	0	0	0	1
	SET1	Sparse	0∗	15∗	20	0∗	20	20
		WP	8	5∗	14	2	5	19
		NS	18	16∗	20	19	20	20
	SET2	Sparse	0	0	1	0	0	0
		WP	0	0	7	0	0	5
		NS	5	15	17	6	4	20
	SET3	Sparse	0	0	1	0	0	0
		WP	0	0	1	3	0	3
		NS	6	0	7	0	0	4
NET2		Baseline	0	0	0	0	0	1
	SET1	Sparse	0∗	17∗	20	0∗	19∗	20
		WP	9	0	0	0	0	0
		NS	18	14∗	19	0∗	17	20
	SET2	Sparse	0∗	0	0	0∗	0	0
		WP	0	0	0	0	0	0
		NS	0	0	1	0	0	0
	SET3	Sparse	0	0	0	0	0	0
		WP	0	0	0	0	0	0
		NS	1	0	0	0	0	0

Table 1. Results — originally reported in [7] — obtained by running NeVer 2.0 with Marabou, ERAN and MIPVerify. The values reported represent the number of problems which were solved successfully within the timeout of 600 CPU seconds. The column **Base** represent the base architecture, **Param** represent the set of parameters used for increasing magnitude of pruning and **Network** represent the kind of network considered. **Marabou**, **MIPVerify**, and **ERAN** represent the number of problems solved by Marabou, MIPVerify and ERAN, respectively.

5 Planned extensions

NeVer 2.0 is an ongoing project and we have already planned various extensions. First, we aim to increase the variety of networks that can be represented by adding more concrete subclasses to *LayerNode*. In particular, we expect to be able to design and implement convolutional layers, the related batch normalization layers, different kinds of pooling layers and different kinds of activation functions. With these extensions, that should be matched by corresponding training, pruning and verification enhancements, NeVer 2.0 should be able to represent all the main kinds of DNNs which current state-of-the-art verification methodologies can deal with.

The second addition that we have already planned, relates to the addition of converters to/from other major learning frameworks, starting from TensorFlow. This addition should include also the capability of visualizing and modifying the network architecture through a graphical user interface, in the hope that NeVer 2.0 becomes more easily accessible also to the non-initiated.

Further additions that we wish to add include quantization, which we believe would create interesting synergies with pruning, and repair, i.e, the capability to modify a neural network to make it compliant to the property of interest. In particular, besides basic form of repair that are already supported by NeVer 2.0, i.e., verification-based adversarial learning, we expect to provide tighter integration between learning and verification.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P.A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zhang, X.: Tensorflow: A system for large-scale machine learning. CoRR **abs/1605.08695** (2016)
2. Akintunde, M., Lomuscio, A., Maganti, L., Pirovano, E.: Reachability analysis for neural agent-environment systems. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018. pp. 184–193. AAAI Press (2018)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Dutta, S., Chen, X., Jha, S., Sankaranarayanan, S., Tiwari, A.: Sherlock - A tool for verification of neural network feedback systems: demo abstract. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019. pp. 262–263 (2019)
5. Franzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation **1**, 209–236 (2007)
6. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)
7. Guidotti, D., Leofante, F., Pulina, L., Tacchella, A.: Verification of neural networks: Enhancing scalability through pruning. In: ECAI 2020 - 24th European Conference on Artificial Intelligence (to appear)
8. Guidotti, D., Tacchella, A., Pulina, L.: NeVer 2.0 (2020), https://gitlab.sagelab.it/dguidotti/fomlas2020-code
9. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural networks. CoRR **abs/1506.02626** (2015)
10. Huang, X., Kroening, D., Kwiatkowska, M., Ruan, W., Sun, Y., Thamo, E., Wu, M., Yi, X.: Safety and trustworthiness of deep neural networks: A survey. arXiv preprint arXiv:1812.08342 (2018)
11. Igel, C., Glasmachers, T., Heidrich-Meisner, V.: Shark. Journal of Machine Learning Research **9**, 993–996 (2008)
12. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. pp. 448–456 (2015)
13. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.W.: The marabou framework for verification and analysis of deep neural networks. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. pp. 443–452 (2019)
14. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015)

15. LeCun, Y., Bengio, Y., Hinton, G.E.: Deep learning. Nature **521**(7553), 436–444 (2015)
16. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
17. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]. pp. 598–605 (1989)
18. Leofante, F., Narodytska, N., Pulina, L., Tacchella, A.: Automated verification of neural networks: Advances, challenges and perspectives. CoRR **abs/1805.09938** (2018)
19. Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., Zhang, C.: Learning efficient convolutional networks through network slimming. In: IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017. pp. 2755–2763 (2017)
20. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. pp. 8024–8035 (2019)
21. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. pp. 243–257 (2010)
22. Pulina, L., Tacchella, A.: Never: a tool for artificial neural networks verification. Annals of Mathematics and Artificial Intelligence **62**(3-4), 403–425 (2011)
23. Pulina, L., Tacchella, A.: Challenging SMT solvers to verify neural networks. AI Commun. **25**(2), 117–135 (2012)
24. Sietsma, J., Dow, R.J.F.: Neural net pruning-why and how. In: Proceedings of International Conference on Neural Networks (ICNN'88), San Diego, CA, USA, July 24-27, 1988. pp. 325–333 (1988)
25. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: Boosting robustness certification of neural networks. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019 (2019)
26. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings (2014)
27. Taigman, Y., Yang, M., Ranzato, M., Wolf, L.: Deepface: Closing the gap to human-level performance in face verification. In: 2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014. pp. 1701–1708 (2014)
28. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019 (2019)
29. Torrey, L., Shavlik, J.: Transfer learning. In: Handbook of research on machine learning applications and trends: algorithms, methods, and techniques, pp. 242–264. IGI Global (2010)

30. Tran, H., Yang, X., Lopez, D.M., Musau, P., Nguyen, L.V., Xiang, W., Bak, S., Johnson, T.T.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. CoRR **abs/2004.05519** (2020)
31. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 6369–6379 (2018)
32. Wu, M., Wicker, M., Ruan, W., Huang, X., Kwiatkowska, M.: A game-based approximate verification of deep neural networks with provable guarantees. CoRR **abs/1807.03571** (2018)
33. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. CoRR **abs/1708.07747** (2017)
34. Xie, Y., Jabri, M.A.: Analysis of the effects of quantization in multilayer neural networks using a statistical model. IEEE Trans. Neural Networks **3**(2), 334–338 (1992)
35. Yu, D., Hinton, G.E., Morgan, N., Chien, J., Sagayama, S.: Introduction to the special section on deep learning for speech and language processing. IEEE Trans. Audio, Speech & Language Processing **20**(1), 4–6 (2012)

Verifying Recurrent Neural Networks using Invariant Inference

Yuval Jacoby[1], Clark Barrett[2], and Guy Katz[1]

[1] The Hebrew University of Jerusalem, Israel
{yuval.jacoby, g.katz}@mail.huji.ac.il
[2] Stanford University, USA
clarkbarrett@stanford.edu

Abstract. Deep neural networks are revolutionizing the way complex systems are developed. However, these automatically-generated networks are opaque to humans, making it difficult to reason about them and guarantee their correctness. Here, we propose a novel approach for verifying properties of a widespread variant of neural networks, called *recurrent neural networks*. Recurrent neural networks play a key role in, e.g., natural language processing, and their verification is crucial for guaranteeing the reliability of many critical systems. Our approach is based on the inference of *invariants*, which allow us to reduce the complex problem of verifying recurrent networks into simpler, non-recurrent problems. Experiments with a proof-of-concept implementation of our approach demonstrate that it performs orders-of-magnitude better than the state of the art.

1 Introduction

The use of *deep neural networks* (*DNNs*) [19] is on the rise. In recent years, DNNs have successfully been applied in domains such as image classification [35], speaker recognition [25], and game playing [47], often achieving much better performance than hand-crafted software. This trend is likely to continue, and we can already observe first signs of safety-critical systems being designed with DNN components [4,28].

We focus here on *recurrent neural networks* (*RNN*s), which are a particular kind of DNNs. Unlike *feed-forward neural networks* (*FFNNs*), where an evaluation of the network is performed independently of past evaluations, RNNs contain memory units that allow them to retain information from previous evaluations. This renders RNNs particularly suited for tasks that involve context, such as text interpretation. Consider, e.g., the sentence "you only live once, but if you do *it* right, once is enough" (Mae West). In order for a neural network that reads the sentence word-by-word to be able to associate the word *it* with the preceding word *live*, it must retain information about words it had previously encountered. Because of this trait, RNNs are increasingly used in machine translation [11], text summarization [40], health applications [38], speaker recognition [24] and game playing [37].

Part of the success of DNNs is attributed to their very attractive generalization properties: after being trained on a finite set of examples, they generalize well to inputs they have not encountered before [19]. Unfortunately, while this works well on average, it is known that DNNs may react in highly undesirable ways to certain inputs. For instance, it has been observed that many DNNs are vulnerable

to *adversarial attacks* [20], where small, carefully-crafted perturbations are added to an input in order to fool the network into performing significant classification errors. In one recent paper, Cisse et al. [9] proposed an approach for producing such adversarial perturbations that could fool RNNs for automatic speaker recognition. This example, and others, highlight the need to *formally verify* the correctness of RNNs, so that they can be reliably deployed in safety-critical settings. However, while DNN verification has received significant attention in recent years (e.g., [5,13,17,27,31,39,41,50,51,53]), almost all of these efforts have been focused on FFNNs, with very little work done on RNN verification.

To the best of our knowledge, the only existing general approach for RNN verification is via *unrolling* [1]: the RNN is duplicated and concatenated onto itself, creating an equivalent feed-forward network that operates on the sequence of inputs simultaneously, as opposed to one at a time. The FFNN can then be verified using existing FFNN verification technology. The main limitation of this approach is that the transformation greatly increases the network size: if the RNN is to be evaluated over k consecutive inputs, the resulting FFNN is k times larger than the original RNN. Because the complexity of FFNN verification is known to be exponential in network size [29], this reduction gives rise to FFNNs that are difficult to verify — and is hence applicable primarily to small RNNs with short input sequences.

Here, we propose a novel approach for RNN verification, which affords far greater scalability than unrolling. Our approach advocates reducing the RNN verification problem to FFNN verification, but does so in a way that is independent of the number of inputs that the RNN is to be evaluated on, and without duplicating the RNN or otherwise increasing its size. Specifically, our approach consists of two main steps: (i) create an FFNN that *over-approximates* the RNN, but which is the same size as the RNN; and (ii) verify this over-approximation using existing techniques for FFNN verification. In order to perform step (i) we leverage the well-studied notion of *inductive invariants*: our FFNN encodes time-invariant properties of the RNN, which hold initially and continue to hold after the RNN is evaluated on each additional input. Using such invariants allows us to circumvent any duplication of the network or its inputs.

Of course, coming up with meaningful inductive invariants is crucial for the success of this approach. At first we present a general framework that receives the inductive invariants from an oracle, proves that they are indeed invariants, and then uses them in over-approximating the RNN using an FFNN. This semi-automatic framework can be useful, e.g., if a human expert can provide meaningful invariants for the system at hand. Next, in order to render the approach fully automatic, we present an approach for *invariant inference* in RNNs. Automated inference of inductive invariants has been studied extensively in the context of program verification, and is known to be undecidable in general. To mitigate this difficulty, our approach attempts to infer invariants according to predefined *templates*. By instantiating these templates, we automatically generate a candidate invariant I, and then: (i) use our underlying FFNN verification engine to prove that I is indeed an invariant; and (ii) use I in creating the FFNN over-approximation of the RNN, in order to prove the desired property. If either of these steps fail, we refine the invariant I (either strengthening or weakening it, depending on the point of failure), and repeat the process. The process terminates when the property is proven correct, when a counter-example is found, or when a certain timeout value is reached.

In order to evaluate our approach, we created a proof-of-concept implementation in Python. As a feed-forward verification back-end we used the Marabou tool [31] (other engines could also be used). When compared to the state of the art on a set of benchmarks from the domain of speaker recognition [24], our approach performs orders-of-magnitude faster. We intend to make our code and benchmarks publicly available online with the final version of this paper.

To summarize, our contributions are:

1. We propose a general framework for the invariant-based verification of recurrent neural networks. The framework reduces the RNN verification problem to the widely studied FFNN verification problem. The complexity of our procedure is independent of the number of time steps that the RNN is to be evaluated.
2. We propose approaches for automatically inferring invariants for RNNs.
3. We provide an implementation of our technique, as well as a new set of benchmark problems.

The rest of this paper is organized as follows. In Section 2, we provide a brief background on DNNs and their verification. In Section 3, we describe our approach for verifying RNNs via a reduction to FFNN verification, using invariants. We describe automated methods for RNN invariant inference in Section 4, followed by an evaluation of our approach in Section 5. We then discuss related work in Section 6, and conclude with Section 7.

2 Background

2.1 Feed-Forward Neural Networks and their Verification

Feed-forward neural networks (FFNNs) are comprised of an input layer, an output layer, and multiple hidden layers in between. A layer is comprised of multiple nodes (neurons), each connected via edges to nodes from the preceding layer. Each node is assigned a bias value, and each edge is assigned a weight value — both of which are determined when the FFNN is trained. The FFNN is evaluated by assigning values to neurons in the input layer (input neurons), and propagating these values forward through the network (hence the name "feed-forward"). The value of each neuron is computed as a weighted sum of values from the preceding layer, according to the edge weights, plus its bias value. This weighted sum is then passed to a non-linear activation function, and this function's output becomes the value for the new node.

A simple example appears in Figure 1. The FFNN depicted therein has a single input neuron $v_{1,1}$, a single output neuron $v_{3,1}$, and two hidden neurons $v_{2,1}$ and $v_{2,2}$ in a single hidden layer. All bias values are assumed to be 0. When the input neuron is assigned $v_{1,1} = 4$, the weighted sums for $v_{2,1}$ and $v_{2,2}$ are 4 and -4, respectively. We use the common $\text{ReLU}(x) = \max(0, x)$ function as our activation function, which yields $v_{2,1} = \text{ReLU}(4) = 4$ and $v_{2,2} = \text{ReLU}(-4) = 0$. Finally, we obtain the output $v_{3,1} = 4$.

More formally, given a DNN N, we use n to denote the number of layers of N. We will use s_i to denote the number of neurons in layer i, also called the *dimension* of layer i. Each hidden layer is associated with a weight matrix W_i of size $s_{i-1} \times s_i$, and a bias vector b_i of size s_i. The DNN input vector will be denoted

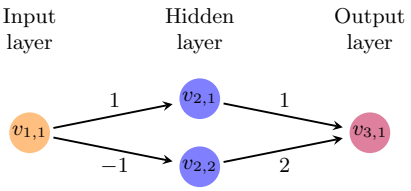


Fig. 1: A simple feed-forward neural network.

as v_1 and the output vector of each hidden layer $1 < i < n$ as $v_i = f(W_i v_{i-1} + b_i)$, where f is some element-wise activation function (such as $\mathrm{ReLU}(x) = \max(0, x)$). The output layer is evaluated similarly, but without an activation function: $v_n = W_{n-1} v_{n-1} + b_n$. We will use $v_{i,j}$ to point to the j'th neuron in the i'th layer. Given some input vector v_1 the network then can be evaluated by sequentially calculating v_i for $i = 2, 3, \ldots, n$, and v_n will be the output of the network. In the remainder of the paper, we will use x and y to denote the input and output layers, respectively; and $|x|$ to denote the size of the input vector, which is just s_1. In addition, unless otherwise stated, for simplicity we will assume that all bias values are 0.

Verifying FFNNs. The goal of verifying an FFNN is to establish whether there exist inputs that satisfy certain constraints, such that their corresponding outputs also satisfy certain constraints. Many interesting problems can be cast into this formulation. Looking again at the network from Figure 1, we might wish to know whether it is always the case that $v_{1,1} \leq 5$ entails $v_{3,1} < 20$. Negating the output property, we can use a verification engine to check whether it is possible that $v_{1,1} \leq 5$ and $v_{3,1} \geq 20$. If this query is unsatisfiable (`UNSAT`), then the original property is bound to hold; but if the query is satisfiable (`SAT`), as is the case here, then the verification engine will provide us with a counter-example, such as $v_{1,1} = -10, v_{3,1} = 20$.

Formally, we define an FFNN verification query as a triple $\langle P, N, Q \rangle$, where N is an FFNN, P is a predicate over the input variables x, and Q is a predicate over the output variables y. Solving this query entails deciding whether there exists a specific input assignment x_0 such that $P(x_0) \wedge Q(N(x_0))$ holds (where $N(x_0)$ is the output of N for the input x_0). It has been shown that even for simple FFNNs and for predicates P and Q that are conjunctions of linear constraints, the verification problem is NP-complete [29]: in the worst-case, solving it requires a number of operations that is exponential in the number of neurons in N.

2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are similar to FFNNs, but have an additional construct called a *memory unit*. These memory units are typically introduced for the purpose of modeling data that changes over discrete units of time (time-series data) [15,52]. Memory units achieve this by allowing a hidden neuron to *store* its assigned values for a specific evaluation of the network, and have that value become part of the neuron's weighted sum computation in the *next* evaluation. Thus, when

evaluating the RNN in time step $t+1$, e.g. when the RNN reads the $t+1$'th word in a sentence, the results of previous evaluations can affect the current result. We say that a DNN is an RNN if at least one of its neurons has a memory unit. An RNN can also contain "regular", memory-less neurons.

A simple example of an RNN appears in Figure 2. There, node $\tilde{v}_{2,1}$ represents node $v_{2,1}$'s memory unit (throughout this paper, we will often draw the memory units as squares, and mark them using the tilde sign). When computing the weighted sum for node $v_{2,1}$, the value of $\tilde{v}_{2,1}$ is also added to the sum, according to its listed weight (1, in this case). Then, once the value of $v_{2,1}$ has been computed, it is stored in $\tilde{v}_{2,1}$ for the next round. In other words, when evaluating $v_{2,1}$ in time step $t+1$, its value from iteration t is added to the weighted sum; and the final value assigned to $v_{2,1}$ is then stored back in $\tilde{v}_{2,1}$, to be used in iteration $t+2$. By convention, we assume that all memory units are initialized to 0 for the first evaluation, at time step $t=1$.

Time Step	$v_{1,1}$	$v_{2,1}$	$\tilde{v}_{2,1}$	$v_{3,1}$
1	0.5	0.5	0	0.5
2	1.5	2	0.5	2
3	-1	1	2	1
4	-3	0	1	0

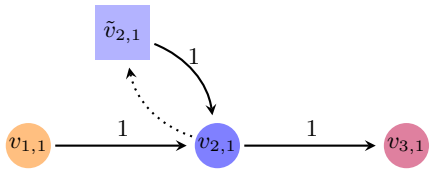


Fig. 2: An illustration of a toy RNN with ReLU activation functions. Each row of the table represents a single time step, and depicts the value of each neuron for that step. Observe that due to the ReLUs functions, the value of $v_{2,1}^t$ is computed as $\max\left(0, \tilde{v}_{2,1}^{t-1} + v_{1,1}^t\right)$, where the t superscript represents time step t.

Formally, we define an RNN as follows. We use the same terminology that we did for FFNNs, but add to each node a superscript t to indicate the timestamp of the RNN's evaluation: for example, $v_{3,2}^4$ indicates the value that node $v_{3,2}$ is assigned in the 4'th evaluation of the RNN. Next, we associate each hidden layer of the RNN with a square matrix H_i of dimension s_i, that represents the weights on edges from memory units to neurons. Observe that each memory unit in layer i can contribute to the weighted sums of all neurons in layer i, and not just to the neuron whose values it stores. For time step $t > 0$, the evaluation of each hidden layer $1 < i < n$ is now computed by:

$$v_i^t = f\left(W_i v_{i-1}^t + H_i v_i^{t-1} + b_i\right) \tag{1}$$

And by convention, we set v_i^0 to be the zero vector for all layers. The output values are computed by:

$$v_n^t = W_n v_{n-1}^t + H_n v_n^{t-1} + b_n$$

To keep the definition simple, we assume that each hidden neuron in the network has a memory unit. This definition captures also "regular" neurons, by setting the appropriate entries of H to 0, effectively cutting off the memory unit.

For simplicity, in our formulation we focus on activation functions that are applied element-wise. There exist more complex functions, such as Long-Short term Memory (LSTM) [26] or Gated Recurrent Unit (GRU) [7], that are applied to multiple nodes at once. Our technique can be extended to this case, as well; we leave this for future work.

Verifying RNNs. Similarly to the FFNN case, we define an RNN verification query as a tuple $\langle P, N, Q, T_{\max} \rangle$, where P is an input property, Q is an output property, N is an RNN, and $T_{\max} \in \mathbb{N}$ is a bound on the time interval for which the property should hold. P and Q include linear constraints over the network's inputs and outputs; only now they may also use the notion of time, stipulating, e.g., that output y_2 at the 5'th time step is at most 10: $y_2^5 \leq 10$.

As a running example, consider the network from Figure 2, denoted by N, the input predicate $P = \bigwedge_{t=1}^{5}(-3 \leq v_{1,1}^t \leq 3)$, the output predicate $Q = \bigvee_{t=1}^{5}(v_{3,1}^t \geq 16)$, and the time bound $T_{\max} = 5$. This query searches for an evaluation of N with 5 time steps, in which all input values are in the range $[-3, 3]$, and such that at some time step the output value is at least 16. By the weights of N, it can be proved that $v_{3,1}^t$ is at most the sum of the ReLUs of inputs so far, $v_{3,1}^t \leq \sum_{i=1}^{t} \text{ReLU}(v_{1,1}^i) \leq 3t$; and so $v_{3,1}^t \leq 15$ for all $1 \leq t \leq 5$, and the query is `UNSAT`.

2.3 Inductive Invariants

Inductive invariants [16] are a well-established way to reason about software with loops. As we later demonstrate, such invariants can be useful in reasoning about RNNs, as these networks perform a loop-like computation.

Formally, let $\langle Q, q_0, T \rangle$ be a transition system, where Q is the set of states, $q_0 \in Q$ is an initial state, and $T \subseteq Q \times Q$ is a transition relation. An invariant I is a logical formula defined over the states of Q, with two properties: (i) I holds for the initial state, i.e. $I(q_0)$ holds; and (ii) I is closed under T, i.e. $(I(q) \wedge \langle q, q' \rangle \in T) \Rightarrow I(q')$. If it can be proved (in a given proof system) that formula I is an invariant, we say that I is an inductive invariant. We use S_I to denote the support of I, i.e. the set $S_I = \{q \in Q \mid I(q)\}$.

Invariants are particularly useful when attempting to verify that a given transition system satisfies a *safety property*. There, we are given a set of bad states B, and seek to prove that none of these states is reachable. We can do so by showing that $S_I \cap B = \emptyset$. Unfortunately, automatically discovering invariants for which the above holds is typically an undecidable problem. Thus, a common approach is to restrict the search space — i.e., to only search for invariants with a certain syntactic form. As we later discuss, in the context of RNNs such an approach is often sufficient for coming up with useful inductive invariants.

3 Reducing RNN Verification to FFNN Verification

3.1 Unrolling

To date, the only available general approach for verifying RNNs [1] is to transform the RNN in question into a *completely equivalent*, feed-forward network, using *unrolling*. An example appears in Figure 3. The idea is to leverage $T_{\max}$, which is an upper bound on the number of times that the RNN will be evaluated. The RNN is duplicated $T_{\max}$ times, once for each time step in question, and its memory units are removed. Finally, the nodes in the i'th copy are used to fill the role of memory units for the $i+1$'th copy of the network.

While unrolling gives a sound reduction from RNN verification to FFNN verification, it unfortunately tends to produce very large networks. When verifying a

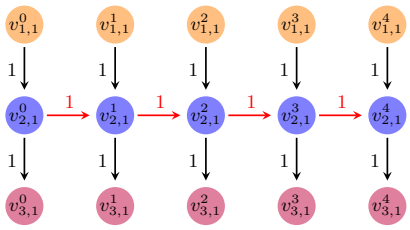


Fig. 3: Unrolling of the network from Figure 2, for $T_{\max} = 5$ time units. The edges in red fill the role of the memory units of the original RNN. The number of neurons in the unrolled network is 5 times the number of neurons in the original.

property that involves t time steps, an RNN network with n memory units will be transformed into an FFNN with $(t-1) \cdot n$ new nodes. Because the FFNN verification problem becomes exponentially more difficult as the network size increases [29], this renders the problem infeasible for large values of t. As scalability is a major limitation of existing FFNN verification technology, unrolling can currently only be applied to small networks that are evaluated for a small number of time steps.

3.2 Circumventing Unrolling

We propose a novel alternative to unrolling, which can reduce RNN verification to FFNN verification without the blowup in network size. The idea is to transform a verification query $\varphi = \langle P, N, Q, T_{\max} \rangle$ over an RNN N into a different verification query $\hat{\varphi} = \langle \hat{P}, \hat{N}, \hat{Q} \rangle$ over an FFNN $\hat{N}$. $\hat{\varphi}$ is not equivalent to φ, but rather *over-approximates* it. The approximation is constructed in a way that guarantees that if $\hat{\varphi}$ in `UNSAT`, then φ is also `UNSAT`. In other words, if the modified property holds for $\hat{N}$, then the original property holds for N. As is often the case, if $\hat{\varphi}$ is `SAT`, either the original property truly does not hold for N, or the invariant I was *too weak*. In the latter case, we can strengthen I and try again; we discuss this case later.

A key point in our approach is that $\hat{\varphi}$ is created in a way that captures the notion of time in the FFNN setting, and without increasing the network size. This is done by incorporating into $\hat{P}$ an *invariant*, that puts bounds on the memory units as a function of the time step t. This invariant does not precisely compute the values of the memory units — instead, it bounds each of them in an interval. This inaccuracy is what makes $\hat{\varphi}$ an over-approximation of φ. More specifically, the construction is performed as follows:

1. $\hat{N}$ is constructed from N by adding a new input neuron, t, to represent time. Because FFNNs typically deal with continuous inputs, we will treat t as a real number.
2. For every node v with memory unit $\tilde{v}$, in $\hat{N}$ we *replace* $\tilde{v}$ with a regular neuron, v^m, which is placed in the input layer. The m superscript signifies that this neuron replaces a memory unit. Neuron v^m will be connected to the network's original neurons, just as $\tilde{v}$ was, and with the same weights as before.[3]

[3] Note that we slightly abuse the definitions from Section 2, by allowing an input neuron to be connected to neurons in layers other than its preceding layer.

3. We set $\hat{P} = P \wedge (1 \leq t \leq T_{\max}) \wedge I$, where I is a formula that bounds the values of each new v^m node as a function of the time step t. The constraints in I constitute the invariant over the memory units' values.
4. The output property is unchanged: $\hat{Q} = Q$.

We name $\hat{\varphi}$ and $\hat{N}$ constructed in this way the *snapshot query* and the *snapshot network*, respectively, and denote $\hat{\varphi} = \mathcal{S}(\varphi)$ and $\hat{N} = \mathcal{S}(N)$. The intuition behind this construction is that query $\hat{\varphi}$ encodes a snapshot (an assignment of t) in which all constraints are satisfied. At this point in time, the v^m nodes represent the values stored in the memory units (whose assignments are bounded by the invariant I); and the input and output nodes represent the network's inputs and outputs at time t. Clearly, a satisfying assignment for $\hat{\varphi}$ does not necessarily indicate a counter-example for φ; e.g., because the values assigned to v^m might be impossible to obtain at time t in the original network. However, if $\hat{\varphi}$ is `UNSAT` then so is φ, because there does not exist a point in time in which the query might be satisfied. Note that the construction only increases the network size by 1 (the v^m neurons replace the memory units, and we add a single neuron t).

Time-Agnostic Properties. In the aforementioned construction of $\hat{\varphi}$, the original properties P and Q appear, either fully or as a conjunct, in the new properties $\hat{P}$ and $\hat{Q}$. It is not immediately clear that this is possible, as P and Q might also involve time. For example, if P is the formula $v^7_{1,2} \geq 10$, it cannot be directly incorporated into $\hat{P}$, because $\hat{N}$ has no notion of time step 7.

To overcome this difficulty, we make the following simplifying assumption: we assume that P and Q are time-agnostic, and are given in the following form: $P = \bigwedge_{t=1}^{T_{\max}} \psi_1$ and $Q = \bigvee_{t=1}^{T_{\max}} \psi_2$, where ψ_1 and ψ_2 are formulas that include linear constraints over the inputs and outputs of N, respectively, at time stamp t. This formulation can express queries in which the inputs are always in a certain interval, and a bound violation of the output nodes is sought. Our running example from Figure 2 fits this form. When the properties are given in this form, we set $\hat{P} = \psi_1$ and $\hat{Q} = \psi_2$, with the t superscripts omitted for all neurons. Later, in Section 4.4, we relax this limitation significantly.

Example. In Figure 4 we demonstrate our approach on the running example from Figure 2. Recall that in that example, our constraints were $P = \bigwedge_{t=1}^{5}(-3 \leq v^t_{1,1} \leq 3)$, and $Q = \bigvee_{t=1}^{5}(v^t_{3,1} \geq 16)$. First, we add a new input neuron, marked t, to represent the time step. Next, we replace the memory unit $\tilde{v}_{2,1}$ with a regular neuron, $v^m_{2,1}$. Node $v^m_{2,1}$ is connected to node $v_{2,1}$ with weight 1 (the same weight previously found on the edge from $\tilde{v}_{2,1}$ to $v_{2,1}$). Next, we set $\hat{P}$ to be the conjunction of (i) P, with its internal conjunction and t superscripts omitted; (ii) the time constraint $1 \leq t \leq 5$; and (iii) the invariant that bounds the values of $v^m_{2,1}$ as a function of time: $v^m_{2,1} \leq 3(t-1)$. Our new verification query is thus:

$$\langle \underbrace{(-3 \leq v_{1,1} \leq 3) \wedge (1 \leq t \leq 5) \wedge (v^m_{2,1} \leq 3(t-1))}_{\hat{P}}, \hat{N}, \underbrace{v_{3,1} \geq 16}_{\hat{Q}} \rangle$$

where $\hat{N}$ is the network in Figure 4. This query is, of course, `UNSAT`, indicating that the original query is also `UNSAT`. Note that the new node t is not connected

to any other node; it is added solely for the purpose of including it in constraints that appear in $\hat{P}$.

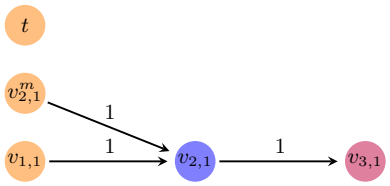


Fig. 4: The feed-forward snapshot network $\hat{N}$ for the RNN from Figure 2. We introduce two new nodes: t to represent the time step, and $v^m_{2,1}$ to take over the role of the memory unit $\tilde{v}_{2,1}$.

The requirement that I be an invariant over the memory units of N ensures that our approach is sound. Specifically, it guarantees that I allows any assignment for v^m that the original memory unit $\tilde{v}$ might be assigned. For instance, looking at the example from Figure 2, consider the input sequence $v^1_{1,1} = 2, v^2_{1,1} = -1, v^3_{1,1} = 3$. This sequence implies that at the beginning of the fourth evaluation, $\tilde{v}^4_{2,1} = v^3_{2,1} = 2 - 1 + 3 = 4$. Our invariant I, which states that $v^m_{2,1} \leq 3(t-1)$ allows this, because $4 < 3 \cdot 3 = 9$. On the other hand, $I' = (v^m_{2,1} \leq t - 1)$ is not a valid invariant for this example, because it forbids a possible assignment of $v^m_{2,1}$.

The soundness guarantee is formulated in the following lemma (whose proof, by induction, is straightforward and is omitted due to lack of space):

Lemma 1. *Let $\varphi = \langle P, N, Q, T_{\max} \rangle$ be an RNN verification query, and let $\hat{\varphi} = \langle \hat{P}, \hat{N}, \hat{Q} \rangle$ be the snapshot query $\hat{\varphi} = \mathcal{S}(\varphi)$. Specifically, let $\hat{N}$ be constructed from N by adding a time neuron t and by replacing each memory unit $\tilde{v}$ with an input neuron v^m; and let $\hat{P} = P \wedge (1 \leq t \leq T_{\max}) \wedge I$. If I is an invariant that bounds the values of each v^m, and if $\hat{\varphi}$ is* `UNSAT`*, then φ is also* `UNSAT`*.*

3.3 Constructing $\hat{\varphi}$: Verifying the Invariant

In Section 3.2 we described a reduction from RNN verification to FFNN verification. A key assumption was that the oracle-provided formula I, which bounds the memory units as a function of the time t, was truly an invariant. This assumption is risky, as a malicious (or simply mistaken) oracle could provide a bogus invariant, jeopardizing the soundness of the process as a whole. In this section we make our method more robust, by including a step that verifies that I is indeed an invariant. This step, too, is performed by creating an FFNN verification query, which can then be dispatched using the back-end FFNN verification engine.

In Section 2.3 we defined the notion of an inductive invariant. In the context of an RNN N, we define the state space Q as the set of states $q = \langle \mathcal{A}, t \rangle$ where $\mathcal{A}$ is the current assignment to the nodes of N (including the assignments of the memory units), and t is a natural number representing the time step. For another state $q' = \langle \mathcal{A}', t' \rangle$, the transition relation $T(q, q')$ holds if and only if:

1. $t' = t + 1$; i.e., the time step has advanced by one;

2. for each memory unit $\tilde{v}$ associated with neuron v, it holds that $\mathcal{A}'[\tilde{v}] = \mathcal{A}[v]$; i.e., the assignment of each neuron in $\mathcal{A}$ is stored in its corresponding memory unit in $\mathcal{A}'$; and
3. the assignment $\mathcal{A}'$ of all of the network's neurons constitutes a proper evaluation of the RNN according to Section 2; i.e., all weighted sums and activation functions are computed properly.

A state q_0 is initial if the time step is 1, all memory units are assigned to 0, and the assignment of all of the network's neurons constitutes a proper evaluation of the RNN (there may be multiple initial states).

Next, let I be a formula over the memory units of N, and suppose we wish to verify that I is an invariant. We slightly abuse notation, and treat I as a formula over both the RNN N and its snapshot network $\hat{N} = \mathcal{S}(N)$; for the latter, every occurrence of memory unit $\tilde{v}^t$ is renamed to v^m. Proving that I is in invariant amounts to proving that $I(q_0)$ holds for any initial state q_0, and that for every two states $q, q' \in Q$, if $I(q)$ then also $I(q')$. Checking whether $I(q_0)$ holds is trivial: in the initial step, all memory units are set to 0, and we can easily check that I holds for this assignment. The second check is more tricky; here, the key point is that because q and q' are consecutive states, the memory units of q' are simply the neurons of q. Thus, we can prove that I holds for q' by looking at the snapshot network, assuming that I holds initially, and proving that $I[v^m \mapsto v, t \mapsto t+1]$, i.e. the invariant with each memory unit v^m renamed to its corresponding neuron v and the time step advanced by 1, also holds. The resulting verification query, which we term φ_I, can be verified using the underlying FFNN verification back-end.

We use our running example from Figure 2 to illustrate this process. Let $I = v^m_{2,1} \leq 3(t-1)$ be our candidate invariant. We begin by checking that I holds at every initial state q_0; this is true because at time $t = 1$, $v^m_{2,1} = 0 \leq 3 \cdot 0$. Next, we assume that I holds for state $q = \langle \mathcal{A}, t \rangle$ and prove that it holds for $q = \langle \mathcal{A}', t+1 \rangle$. First, we create the snapshot FFNN $\hat{N}$, shown in Figure 4. We then extend the original input property $P = \bigwedge_{t=1}^{5}(-3 \leq v^t_{1,1} \leq 3)$ into a property P' that also captures our assumption that the invariant holds at time t:

$$P' = (-3 \leq v_{1,1} \leq 3) \wedge (v^m_{2,1} \leq 3(t-1)).$$

Finally, we prepare an output property Q' that asserts that the invariant does not hold for $v_{2,1}$ at time $t+1$, by renaming $v^m_{2,1}$ to $v_{2,1}$ and incrementing t:

$$Q' = \neg(v_{2,1} \leq 3(t+1-1)).$$

When the FFNN verification engine answers that $\varphi_I = \langle P', \mathcal{S}(N), Q' \rangle$ is `UNSAT`, we can conclude that I is indeed an invariant. In cases where the query turns out to be `SAT`, I is not an invariant, and the counter-example returned by the underlying verifier can be used to refine it.

The steps described in this section, namely (i) prove that a formula I is an invariant; (ii) use I to create a query $\hat{\varphi}$ that over-approximates the original query φ; and (iii) prove that is $\hat{\varphi}$ `UNSAT`, allow us to automatically reduce the RNN verification problem to the FFNN verification problem. The only part of the process that is not automated is coming up with the invariant I — which is the topic of the following section.

4 Invariant Inference

In order to reduce RNN verification to FFNN verification, we require an invariant I that bounds the values of each memory unit as a function of time t. In general, automatically discovering such invariants is an undecidable problem [44]. To mitigate this difficulty we restrict ourselves to invariants that follow a *linear template*.

4.1 Single Memory Unit

We begin with a simple case, in which the network has a single hidden node v with a memory unit (we will relax this limitation later). Note that this is the case in the running example depicted in Figure 2. Here, inferring an invariant according to a linear template means finding values α_l and α_u, such that:

$$\alpha_l \cdot (t-1) \leq \tilde{v}^t \leq \alpha_u \cdot (t-1) \tag{2}$$

Thus, the goal is to bound the value of $\tilde{v}^t$ from below (using α_l) and from above (using α_u), both as a function of time. For simplicity, we focus only on finding the upper bound; the lower bound case is symmetrical. For our running example, we have already seen such an upper bound, which was sufficiently strong for proving the desired property: $\tilde{v}^t_{2,1} \leq 3(t-1)$.

Once candidate α's are proposed, verifying that the invariant holds can be performed using the techniques outlined in Section 3. There are two places where the process might fail: (i) the proposed invariant cannot be proved (φ_I is `SAT`), because a counter-example exists. This means that our invariant is *too strong*, i.e. the bound is too tight. In this case we can weaken the invariant by increasing α_u; or (ii) the proposed invariant holds, but the FFNN verification problem that it leads to, $\hat{\varphi}$, is `SAT`. In this case, the invariant is *too weak*; it is indeed an invariant, but does not imply the desired output property. In this case, we can strengthen the invariant by decreasing α_u.

To illustrate these possible failures, we return to our running example from Figure 2. Assume we set $\alpha_u = 100$. This value gives rise to a valid upper bound (since we know that $\tilde{v}^t_{2,1} \leq 3(t-1) \leq 100t$). However, this invariant is too weak to prove that $\hat{\varphi}$ is `UNSAT`; for example, assigning $t = 4$, $v_{1,1} = 0$ and $v^m_{2,1} = 40$ will imply $v_{3,1} = 40$ which satisfies the output condition $Q = \bigvee_{t=1}^{5}(v^t_{3,1} \geq 16)$. We can conclude that a stronger invariant, i.e. a smaller α_u, is required. For the other possible failure, let us set $\alpha_u = 1$; in this case, the resulting formula is not an invariant, e.g. because of the input assignment $v_{1,1} = 3$, $t = 1$ and $v^m_{2,1} = 0$, which leads to $v_{2,1} = 3 > 1 \cdot (t+1)$. This implies that a greater α_u is required.

The aforementioned example leads us to binary search strategy. We define a search range $[lb, ub]$ for α_u. Initially, $lb = -M$ and $b = M$ for a very large constant M. We set $\alpha = \frac{1}{2}(lb + ub)$, and attempt our procedure. If the candidate invariant is too weak, ub is decreased; and if it is too strong, lb is increased. The search stops, i.e. an optimal invariant is found, when $ub - lb \leq \epsilon$ for a small constant ϵ. The result verification procedure (for networks with a single memory unit) is described in Alg. 1. The algorithm fails if the optimal linear invariant is found, but is insufficient for proving the property in question; this can happen if φ is indeed `SAT`, or if a more sophisticated invariant is required. The algorithm can be extended in a straightforward manner to incorporate lower bound invariants as well.

Algorithm 1 Automatic Single Memory Unit Verification$(P, N, Q, T_{\max})$

$lb \leftarrow -M$, $ub \leftarrow M$ ▷ M is a large constant
while $ub - lb \geq \epsilon$ **do**
  $\alpha_u \leftarrow \frac{ub+lb}{2}$
  $I \leftarrow v^m_{2,1} \leq \alpha_u \cdot t$
  if φ_I is `UNSAT` **then**
    Construct $\hat{\varphi}$ using invariant I
    if $\hat{\varphi}$ is `UNSAT` **then**
      return True
    $ub \leftarrow \alpha_u$ ▷ Invariant too weak
  else
    $lb \leftarrow \alpha_u$ ▷ Invariant too strong
return False

Linear Templates: Pros and Cons. Using linear invariant templates affords two key benefits. First, the generated invariants have a simple form, and approaches based on binary-search can be used to optimize them. Second, because linear constraints can be encoded into most FFNN verification tools, the resulting φ_I queries can be verified automatically; this would not have been the case, e.g., if we had used upper bounds that are quadratic in t, which FFNN verification tools typically do not support. The downside of using the linear template is that we might miss out on more complex invariants, which may be required to prove the property at hand. Identifying additional kinds of templates that are both expressive and supported by FFNN verification tools is an important avenue for future work.

4.2 Multiple Layers with Single Memory Units

We now extend our approach from RNNs with a single memory unit to RNNs with multiple memory units, each in a separate layer. The extension is performed in an iterative fashion. As before, we begin by constructing the snapshot network in which all memory units are replaced by regular neurons. Next, we work layer by layer and generate invariants that over-approximate each memory unit. As we go into deeper layers of the network, we use previously-proven invariants in order to bound the current memory unit. Eventually, all memory units are over-approximated using invariants, and we can attempt to prove the desired property.

An example appears in Figure 5. Let $P = \bigwedge_{t=1}^{5}(-3 \leq v^t_{1,1} \leq 3)$ and $Q = \bigvee_{t=1}^{5}(v^t_{4,1} \geq 60)$. First we construct the snapshot network shown in the figure. Next, we prove the invariant $v^m_{2,1} \leq 3(t-1)$, same as we did before. This invariant bounds the values of $v^m_{2,1}$. Next, we use this information in proving an invariant also for $v_{3,1}$; e.g., $v^m_{3,1} \leq 9 \cdot (t-1)$. To see why this is an invariant, observe that

$$v^t_{3,1} = \sum_{i=1}^{t} v^i_{2,1} \overset{*}{\leq} \sum_{i=1}^{t} 3i = (3+3t)\frac{t}{2} \leq (3+3T_{\max})\frac{t}{2} = 9t$$

Note that the invariant for $v_{2,1}$ was used in the $*$ transition. Once this invariant is proved, we can show that the original property holds, by using FFNN verification

to show that the following snapshot query in UNSAT:

$$\langle(-3 \leq v_{1,1} \leq 3) \wedge (1 \leq t \leq 5) \wedge (v^m_{2,1} \leq 3(t-1)) \wedge (v^m_{3,1} \leq 9(t-1)), \mathcal{S}(N), v_{4,1} \geq 60\rangle$$

where $\mathcal{S}(N)$ is the FFNN from Figure 5.

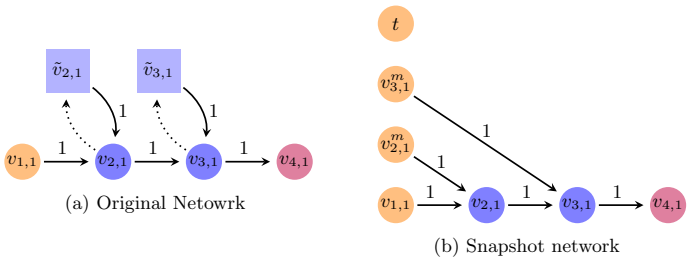
The general algorithm for this case is given as Alg. 2. We assume for simplicity that *every* hidden layer in the RNN has a (single) memory unit. Initially (Lines 2-5), the algorithm guesses and verifies a very coarse upper bound on each of the memory units. Next, it repeatedly attempts to solve the snapshot query $\hat{\varphi}$ using the current invariants. If successful, we are done (Line 9); and otherwise, we start another pass over the network's layers, attempting to strengthen each invariant in turn. We know we have reached an optimal invariant for a layer when the search range for that layer's α becomes smaller than some small ϵ constant (Line 13). The algorithm fails (Line 22) when (i) optimal invariants for all layers have been discovered; and (ii) these optimal invariants are insufficient for solving the snapshot query. As before, the algorithm only deals with upper bounds, and can be extended to incorporate lower bound invariants as well.

Algorithm 2 Automatic Multiple Memory Units Verification$(P, N, Q, T_{\max})$

1: Construct the snapshot network $\hat{N}$
2: **for** $i = 2$ to $n-1$ **do**
3: $lb_i \leftarrow -M, ub_i \leftarrow M, \alpha_i \leftarrow M$ ▷ M is a large constant
4: $I_i \leftarrow v^m_i \leq \alpha_i \cdot t$ ▷ Loose invariant, should hold
5: **if** φ_{I_i} is UNSAT **then return** False ▷ Loose invariant fails, give up
6: **while** True **do**
7: Construct $\hat{\varphi}$ using the invariant $\bigwedge_{i=2}^{n-1} I_i$
8: **if** $\hat{\varphi}$ is UNSAT **then**
9: **return** True ▷ Invariants sufficiently strong
10: **else**
11: progressMade $\leftarrow$ False
12: **for** $i = 2$ to $n-1$ **do**
13: **if** $ub_i - lb_i \leq \epsilon$ **then**
14: Continue ▷ Already have optimal invariant
15: progressMade $\leftarrow$ True ▷ Still searching for optimal invariant
16: $\alpha'_i \leftarrow (ub_i + lb_i)/2$
17: $I'_i \leftarrow v^m_i \leq \alpha'_i \cdot t$
18: **if** φ'_{I_i} is UNSAT **then**
19: $I_i \leftarrow I'_i, \alpha_i \leftarrow \alpha'_i, ub_i \leftarrow \alpha_i$ ▷ Better invariant found
20: **else**
21: $lb_i \leftarrow \alpha'_i$ ▷ Invariant unchanged, range shrinks
22: **if** !progressMade **then return** False ▷ Best invariants too weak

4.3 Layers with Multiple Memory Units

We now extend our approach to support the most general case: an RNN with layers that contain multiple memory units. The idea is the same as in Alg. 2: we iteratively infer and prove invariants for each layer, leveraging the already-proven invariants from layer k when proving invariants for layer $k+1$. Once all layers have



(a) Original Netowrk

(b) Snapshot network

Fig. 5: An RNN with multiple memory units, in separate layers (on the left), and its snapshot network (on the right).

been handled, we attempt to show that the snapshot query is `UNSAT`; and if that fails, we go back and strengthen previous invariants.

The main difficulty is in inferring invariants for a layer that has multiple memory units. Figure 6 depicts a simple example of an RNN that has a single layer with multiple memory units (although the technique applies also to RNNs with multiple layers with memory units). The key point is that while each memory unit belongs to a single neuron, it affects the assignments of all other neurons in that layer.

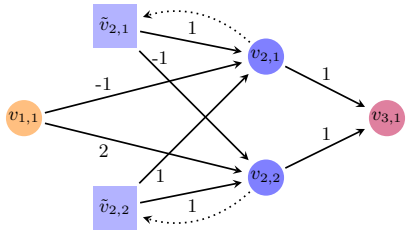


Fig. 6: An RNN with a 2-dimensional recurrent layer. Both memory units affect both neurons of the hidden layer: $v^t_{2,1} = \text{ReLU}(\tilde{v}^t_{2,1} + \tilde{v}^t_{2,2} - v^t_{1,1})$; and $v^t_{2,2} = \text{ReLU}(-\tilde{v}^t_{2,1} + \tilde{v}^t_{2,2} + 2v^t_{1,1})$.

We propose to handle this case using separate linear invariants for upper- and lower-bounding each of the memory units. However, while the invariants have the same linear form as in the single memory unit case, *proving* them requires taking the other invariants of the same layer into account. Consider the example in Figure 6, and suppose we have α^1_l, α^1_u and α^2_l, α^2_u for which we wish to verify that

$$\alpha^1_l \cdot t \leq v^t_{2,1} \leq \alpha^1_u \cdot t \qquad \alpha^2_l \cdot t \leq v^t_{2,2} \leq \alpha^2_u \cdot t \tag{3}$$

In order to prove these bounds we need to dispatch an FFNN verification query that assumes Eq. 3 holds and uses it to prove the inductive step:

$$v^{t+1}_{2,1} = \text{ReLU}(-v^t_{1,1} + v^t_{2,1} + v^t_{2,2}) \leq \alpha^1_u \cdot (t+1) \tag{4}$$

Similar steps must be performed for $v^{t+1}_{2,1}$'s lower bound, and also for $v^{t+1}_{2,2}$'s lower and upper bounds. The key point is that because Eq. 4 involves $v^t_{2,2}$, the bounds proved for this neuron, and hence the values of α^2_l and α^2_u, must be used. In the

single memory unit case, whenever Eq. 4 did not hold we would increase the value of α; and if proving the desired property failed, we would decrease the value of α. Here, the situation is more complex; specifically, it is possible that increasing α_1^u would invalidate previously acceptable assignments for α_l^2 or α_u^2.

For example, consider the network in Figure 6, with $P = \bigwedge_{t=1}^{3} -3 \leq v_{1,1}^t \leq 3$, and $Q = \bigvee_{t=1}^{3} v_{3,1}^t \geq 90$. Our goal is to find values for α_l^1, α_u^1 and α_l^2, α_u^2 that will satisfy Eq. 3. Let us consider $\alpha_l^1 = 0, \alpha_u^1 = 5, \alpha_l^2 = 0$ and $\alpha_u^2 = 0$. Using these values, the induction hypothesis (Eq. 3) and the bounds for $v_{1,1}$, we can indeed prove the upper bound for $v_{2,1}^t$:

$$v_{2,1}^{t+1} = \text{ReLU}(-v_{1,1}^t + v_{2,1}^t + v_{2,2}^t) \leq -v_{1,1}^t + v_{2,1}^t + v_{2,2}^t \leq 3 + 5t + 0 \leq 5(t+1)$$

Unfortunately, the bounds $0 \leq v_{2,2}^t \leq 0$ are inadequate, because $v_{2,2}^t$ can take on positive values. We are thus required to adjust the α values, for example by increasing α_u^2 to 2. However, this change invalidates the upper bound for $v_{2,1}^t$, i.e. $v_{2,1}^t \leq 5t$, as that bound relied on the upper bound for $v_{2,2}^t$; specifically, knowing only that $-3 \leq v_{1,1}^t \leq 3$, $0 \leq v_{2,1}^t \leq 5t$ and $0 \leq v_{2,2}^t \leq 2t$, it is impossible to show that $v_{2,1}^{t+1} \leq 5(t+1)$. Next, trying the assignment $\alpha_l^1 = 0, \alpha_u^1 = 25, \alpha_l^2 = 0$ and $\alpha_u^2 = 6$, the resulting invariants are provable. Unfortunately, these invariants are insufficient for solving the snapshot query, i.e. to show that $v_{3,1}^t < 90$; to see this, recall that $v_{3,1}^t = v_{2,1}^t + v_{2,2}^t$, and thus the strongest bound we can show is $v_{3,1}^t \leq v_{2,1}^t + v_{2,2}^t = 25t + 6t$, which does not imply $v_{3,1}^t < 90$ for $T_{\max} = 3$. Thus, our invariants need to be strengthened — by either increasing α_l^1 or α_l^2, or by decreasing α_u^1 or α_u^2. We observe that adjusting α_u^1 to 21 produces a valid set of invariants that imply the unsatisfiability of the snapshot query.

The example above demonstrates the intricate dependencies between the α values, and the complexity that these dependencies add to the search process. Unlike in the single memory unit case, it is not immediately clear how to find an initial invariant that simultaneously holds for all memory units, or how to strengthen this invariant (e.g., which α constant to try and improve).

Finding an Initial Invariant. We propose to encode the problem of finding initial α values as a *mixed integer linear program* (MILP) [8]. The constraints that the α values must satisfy (e.g., Eq. 4) include weighted sums and piecewise-linear activation functions. Weighted sums can be encoded directly in MILP, and the piecewise-linear constraints can also be encoded in a straightforward way, using big-M encoding [2,29]. There are two main advantages to using MILP in this context:

- assuming only piecewise-linear activation functions, the problem can be precisely encoded in MILP. This means that the MILP solver is guaranteed to return a valid invariant, or soundly report that no such invariant exists.
- MILP instances also include a *cost function* to be minimized. We can leverage this fact to optimize our starting invariant; for example, by setting the cost function to be $\sum \alpha_u - \sum \alpha_l$, the MILP solver will typically suggest a solution in which the upper bound α's are small and the lower bound α's are large.

The main disadvantage to using MILP is that, in order to ensure that the invariants hold for all time steps $1 \leq t \leq T_{\max}$, we must encode all of these steps in the MILP query. For example, going back to Eq. 4, in order to guarantee that

$$v_{2,1}^{t+1} = \text{ReLU}(-v_{1,1}^t + v_{2,1}^t + v_{2,2}^t) \leq \alpha_u^1 \cdot (t+1)$$

we would need to encode within our MILP instance the fact that

$$\bigwedge_{t=1}^{T_{\max}} \left(\text{ReLU}(-v_{1,1}^t + v_{2,1}^t + v_{2,2}^t) \leq \alpha_u^1 \cdot (t+1)\right)$$

which might render the MILP instance difficult to solve for large values of $T_{\max}$.

MILP solvers are mature tools, and can handle very large input instances; and indeed, in our experiments, this was never the bottleneck. Still, should this become a problem, we propose to encode only a subset of the values of $t \in \{1, \ldots, T_{\max}\}$, making the problem easier to solve; and should the α assignment fail to produce an invariant (this will be discovered when φ_I is verified), additional constraints can be added to guide the MILP solver towards a correct solution.

Strengthening the Invariant. If we are unable to prove that snapshot query $\hat{\varphi}$ is `UNSAT` for a given I, then the invariant needs to be strengthened. We propose to achieve this by invoking the MILP solver again, this time adding new linear constraints that force the selection of tighter bounds. For example, if the current invariant is $\alpha_l = 3, \alpha_u = 7$, we propose to add constraints specifying that $\alpha_l \geq 3+\epsilon$ and $\alpha_u \leq 7 - \epsilon$, for some small positive ϵ. This guarantees a strengthening of the invariants, although the improvement may be very small — depending on the value of ϵ being used. It is possible to require the strengthening of all α values, or to stipulate this only for some of the values, based on some heuristics.

Invariants without using MILP. In cases where using a MILP solver is undesirable (for example, if $T_{\max}$ is very large and the MILP instances becomes a bottleneck), we propose an *incremental approach*. Here, we start with an arbitrary assignment of α values, which may or may not constitute an invariant, and iteratively change one α at a time. This change needs to be "in the right direction" — i.e., if our α's do not currently constitute an invariant, we need to weaken the bounds; and if they do constitute an invariant but that invariant is too weak to solve $\hat{\varphi}$, we need to tighten the bounds. The selection of which α to change and by how much to change it can be random, arbitrary, or according to some heuristic. In our experiments we observed that while this approach is computationally cheaper than solving MILP instances, it tends to lead to longer sequences of refining the α's before an appropriate invariant is found. Devising heuristics that will improve the efficiency of this approach remains a topic for future work.

4.4 Time-Dependent Properties

Our technique for verifying inferred invariants and for using them to solve the snapshot query (and hence to prove the property in question) hinges on our ability to reduce each step into an FFNN verification query. In order to facilitate this, we made the simplifying assumption that properties P and Q of the RNN verification query are time-agnostic; i.e. they are of the form $P = \bigwedge_{t=1}^{T_{\max}} \psi_1$ and $Q = \bigvee_{t=1}^{T_{\max}} \psi_2$ for ψ_1 and ψ_2 that are conjunctions of linear constraints. However, this limitation can be relaxed significantly.

Currently, input property P specifies a constant range for the inputs, e.g. $-3 \leq v_{1,1}^t \leq 3$ for all $1 \leq t \leq T_{\max}$. However, we observe that our technique can be applied also for input properties that encode linear time constraints; e.g. $4t \leq v_{1,1}^t \leq 5t$.

These properties can be transferred, as-is, to the FFNN snapshot network, and are compatible with our proposed technique. Likewise, the output property Q can also include constraints that are linear in t; and can also restrict the query for a particular time step $t = t_0$. In fact, even more complex, piecewise-linear constraints can be encoded and are compatible with our technique. However, encoding these constraints might entail adding additional neurons to the RNN. For example, the constraint $(\max(v^t_{1,1}, v^t_{1,2}) \geq 5t)$ is piecewise-linear and can be encoded [6]; the encoding itself is technical, and is omitted to save space. As for constraints that are not piecewise-linear, if these can be soundly approximated using piecewise-linear constraints, then they can be soundly handled using our technique.

5 Evaluation

We created a proof-of-concept implementation of our approach as a Python module, called *RnnVerify*, which reads an RNN in TensorFlow format. The input and output properties, P and Q, and also $T_{\max}$, are supplied in a simple proprietary format, and the tool then automatically: (i) creates the FFNN snapshot network; (ii) infers a candidate invariant using the MILP heuristics from Section 4; (iii) formally verifies that I is an invariant; and (iv) uses I to show that $\hat{\varphi}$, and hence φ, are `UNSAT`. If $\hat{\varphi}$ is `SAT`, our module refines I and repeats the process. We intend to make our code publicly available with the final version of this paper.

For our evaluation, we focused on neural networks for *speaker recognition* — a task for which RNNs are commonly used, because audio signals tend to have temporal properties and varying lengths. We applied our verification technique to prove *adversarial robustness* properties of these networks, as we describe next.

Adversarial Robustness. It has been shown that *many* neural networks are susceptible to *adversarial inputs* [49]. These inputs are generated by slightly perturbing correctly-classified inputs, in a way that causes the misclassification of the perturbed inputs. Formally, given a network N that classifies inputs into labels $l_1, \ldots, l_k$, an input x_0, and a target label $l \neq N(x_0)$, an adversarial input is an input x such that $N(x) = l$ and $\|x - x_0\| \leq \delta$; i.e., input x is very close to x_0, but is misclassified as label l. (There are additional, more complex variants of the problem [30]; we focus on this variant for simplicity.)

Adversarial robustness is a measure of how difficult it is to find an adversarial example — and specifically, what is the minimal amount of noise, i.e. the smallest δ, for which such an example exists. Verification can be used to find adversarial inputs or rule out their existence for a given δ, and consequently can find the smallest δ for which an adversarial input exists [6]. Using verification for proving the adversarial robustness of FFNNs has been studied extensively (e.g., [17,29,31,51]).

Speaker Recognition. A speaker recognition system receives a voice sample and needs to identify the speaker from a set of people. This is a private case of *speaker verification*, where a system needs to determine whether a voice sample belongs to a certain person. RNNs are often applied in implementing such systems [24], rendering them vulnerable to adversarial attacks [34]. Because such vulnerabilities in these systems pose a security concern [23], it is important to verify that their underlying RNNs afford high adversarial robustness.

Benchmarks. We trained 6 speaker recognition RNNs, based on the VCTK dataset [54]. Our networks are of modest, varying sizes: they each contain an input layer of dimension 40, one or two hidden layers with $d \in \{2, 4, 8\}$ memory units, followed by 1 to 3 fully connected, memoryless layers. The networks were trained to distinguish between 20 possible speakers.

Next, we selected 5 random, fixed input points $X = \{x_1, \ldots, x_5\}$, that do not change over time; i.e. $x \in \mathbb{R}^{40}$ and $x_i^1 = x_i^2 = \ldots$ for each $x_i \in X$. Then, for each RNN N and input $x_i \in X$, and for each value $2 \leq T_{\max} \leq 19$, we computed the ground-truth label $l = N(x_i)$, which is the label that received the highest score at time step $T_{\max}$. We also computed the label that received the second-highest score, l_{sh}, at time step $T_{\max}$. Then, for every combination of N, $x_i \in X$, and value of $T_{\max}$, we created the query

$$\langle \underbrace{\bigwedge_{t=1}^{T_{\max}} \|x'^t - x_i^t\|_{L_\infty} \leq 0.01}_{P}, N, \underbrace{l_{sh} \geq l}_{Q} \rangle$$

The allowed perturbation, at most 0.01 in L_∞ norm, was selected arbitrarily. The query is only `SAT` if there exists an input x' that is at distance at most 0.01 from x, but for which label l_{sh} is assigned a higher score than l at time step $T_{\max}$. This formulation resulted in a total of 540 benchmark queries. We then used RnnVerify to solve these queries, using an Intel i5 laptop with 8 cores and 8GB of memory.

We began by comparing our technique to the state of the art, namely unrolling [1]. We selected our smallest RNN — a network with only 2 memory units in one hidden layer, followed by a single fully connected layer. We compared the performance of unrolling to that of our tool, RnnVerify, for increasingly larger $T_{\max}$ values. Both methods returned `UNSAT` results for all of these queries; however, the runtimes clearly demonstrate that our approach is far less sensitive to large $T_{\max}$ values, and can perform orders-of-magnitude better than the state of the art when such values are encountered. Figure 7 summarizes the results.

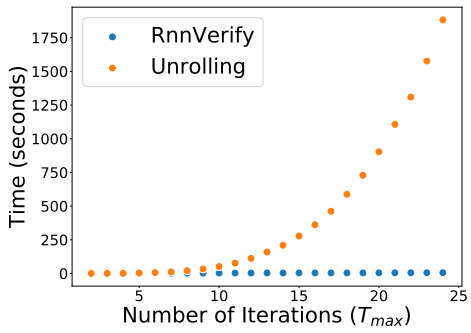


Fig. 7: Comparing the running time (in seconds) of RnnVerify and unrolling, as a function of $T_{\max}$. Both methods returned `UNSAT` on all queries.

Table 1 summarizes the performance of our approach over all 540 benchmark queries. We observe the following points (i) RnnVerify was usually able to decide

within a few seconds whether the inferred linear invariants could prove the desired property or not. The average run time over all 540 experiments was 2.67 seconds; (ii) we successfully proved that for 295 of the tested benchmarks, the RNN was robust around the tested point. For the remaining 245 benchmarks, our results are inconclusive: we do not know whether the network is vulnerable, or whether more sophisticated invariants are needed to prove robustness. Still, in the majority of tested benchmarks, the linear template proved useful; and (iii) we see that our linear invariants generally become less effective for larger values of $T_{\max}$. This is because the linear bounds become more loose as t increases, whereas the neurons' values typically do not increase significantly over time. This highlights the need for more expressive forms of invariants, perhaps to be combined with our current linear template.

Table 1: Running RnnVerify on our 540 benchmarks. Network $N_{x,y,z}$ has a hidden layer with x memory units; a second hidden layer with y memory units, if $y > 0$; and z fully connected layers. Each entry depicts the average runtime, in seconds, over the 5 input points; and also the number $x/5$ of queries where RnnVerify successfully proved adversarial robustness. In the remaining $5-x$ queries, linear invariants were insufficient for proving that the snapshot query is `UNSAT`.

$T_{\max}$	$N_{2,0,1}$	$N_{2,0,2}$	$N_{4,0,2}$	$N_{4,0,3}$	$N_{4,2,3}$	$N_{8,0,2}$
2	1.27 (5/5)	1.38 (5/5)	1.32 (5/5)	1.37 (5/5)	2.78 (5/5)	1.96 (5/5)
3	1.32 (5/5)	2.15 (5/5)	1.34 (5/5)	1.58 (5/5)	5.47 (5/5)	2.63 (5/5)
4	1.51 (5/5)	1.49 (5/5)	1.60 (5/5)	1.56 (5/5)	2.31 (5/5)	1.88 (5/5)
5	1.86 (5/5)	1.51 (5/5)	1.63 (5/5)	72.23 (5/5)	3.02 (5/5)	1.57 (0/5)
6	1.49 (5/5)	1.63 (5/5)	1.55 (5/5)	1.76 (5/5)	2.93 (5/5)	1.76 (0/5)
7	2.58 (5/5)	1.70 (5/5)	1.69 (5/5)	3.39 (5/5)	2.87 (5/5)	2.04 (0/5)
8	1.49 (5/5)	1.79 (0/5)	1.60 (5/5)	1.63 (5/5)	1.93 (5/5)	2.27 (0/5)
9	1.42 (5/5)	1.65 (0/5)	1.60 (5/5)	1.60 (5/5)	2.24 (5/5)	2.10 (0/5)
10	1.47 (5/5)	1.72 (0/5)	4.62 (5/5)	1.71 (5/5)	2.04 (5/5)	1.97 (0/5)
11	1.45 (5/5)	1.74 (0/5)	1.64 (5/5)	1.81 (5/5)	2.44 (5/5)	2.28 (0/5)
12	1.26 (5/5)	1.71 (0/5)	1.36 (5/5)	1.35 (5/5)	3.76 (5/5)	1.70 (0/5)
13	1.14 (5/5)	1.56 (0/5)	1.43 (5/5)	1.30 (5/5)	2.59 (5/5)	1.78 (0/5)
14	1.23 (5/5)	3.06 (0/5)	2.34 (5/5)	1.54 (5/5)	2.30 (5/5)	1.77 (0/5)
15	3.15 (5/5)	1.39 (0/5)	1.23 (1/5)	3.57 (5/5)	2.91 (5/5)	1.99 (0/5)
16	1.25 (5/5)	1.66 (0/5)	1.40 (1/5)	1.34 (5/5)	2.72 (5/5)	2.29 (0/5)
17	1.30 (5/5)	1.49 (0/5)	1.41 (1/5)	1.34 (5/5)	3.09 (5/5)	1.97 (0/5)
18	1.34 (5/5)	1.78 (0/5)	1.42 (1/5)	2.00 (5/5)	3.21 (5/5)	2.22 (0/5)
19	1.38 (5/5)	4.12 (0/5)	3.89 (1/5)	1.56 (5/5)	2.88 (5/5)	5.83 (0/5)

6 Related Work

Due to the rise in neural network prevalence and the discovery of undesirable behaviors in many of them, the verification community has begun putting significant efforts into DNN verification. Recently proposed approaches include the use of SMT solving [27,29,31,36], LP and MILP solving [13,50], symbolic interval propagation [51], abstraction-refinement and abstract interpretation [14,17], and many

others (e.g., [5,12,18,21,32,39,41,48]). Our technique focuses on RNN verification, but uses an FFNN verification engine as a back-end. Consequently, it could be integrated with many of the aforementioned tools, and will benefit from any improvement in scalability of FFNN verification technology.

Whereas FFNN verification has received a great deal of attention, to the best of our knowledge only little research has been carried out on RNN verification. Akintunde et al. [1] were the first to propose such a technique, based on the notion of *unrolling* — the duplication of an RNN and concatenation of the copies, in order to create an equivalent FFNN. Ko et al. [33] proposed a similar framework, aimed at quantifying the robustness of an RNN to adversarial inputs — which can be regarded as an RNN verification technique tailored for a particular kind of properties. The scalability of both approaches is highly sensitive to the number of time steps, $T_{\max}$, specified by the property at hand.

The main advantage of our approach compared to the state of the art is that it is far less sensitive to the number of time steps being considered. Specifically, our construction of the snapshot query $\hat{\varphi}$ is oblivious to $T_{\max}$. This affords great potential for scalability, especially for long sequences of inputs. A drawback of our approach is that it requires invariant inference, which is known to be challenging.

Automated invariant inference is key in program analysis [43], especially for programs with loops, and has been studied extensively since the 70's [10]. A few notable methods for doing so include: (i) abstract interpretation based: here, the program code is automatically analyzed in order to identify atoms from which an invariant formula can be constructed (e.g., [10,22,45]); (ii) counterexample-guided based: these approaches start with a candidate invariant generated, e.g., using program traces, and attempt to verify that it is truly an invariant. If the candidate fails, the counter-example returned by the verification tool is used to refine it (e.g., [3,42]); and (iii) learning based: recent work has suggested using machine learning to automatically suggest loop invariants [46]. It will be interesting to apply these techniques within the context of our framework, in order to more quickly and effectively discover useful invariants.

7 Conclusion

Neural network verification is an open problem that is becoming increasingly important to industry, regulators, and society as a whole. Research to date has focused primarily on FFNNs. We propose a novel approach for the verification of recurrent neural networks — a kind of neural networks that is particularly useful for context-dependent tasks, such as NLP. The cornerstone of our approach is the reduction of RNN verification to FFNN verification through the use of inductive invariants. Using a proof-of-concept implementation, we demonstrated that our approach can tackle many benchmarks orders-of-magnitude more efficiently than the state of the art. These experiments indicate the great potential that our approach holds.

Still, our work so far is but a first step, and we are already working on extending it along three axes. First, our approach depends greatly on our ability to generate useful inductive invariants to the problem at hand. Currently we have focused on invariants that adhere to linear templates; and we plan to experiment with additional approaches, such as those described in Section 6. Second, more work is

required to increase the scalability of our approach, especially for networks that contain multiple hidden layers with memory units. For such networks, generating sufficiently strong invariants is challenging, and we plan to tackle this difficult using *compositional verification* techniques in order to break the RNN into multiple, smaller networks, each with fewer memory units. Finally, the lack of available RNN verification benchmarks is a limiting factor; to mitigate it, we plan to apply our approach to additional, real-world RNN based systems.

Acknowledgements. This project was partially supported by grants from the Semiconductor Research Corporation, the Binational Science Foundation (2017662), the Israel Science Foundation (683/18), and the National Science Foundation (1814369).

References

1. M. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd Conf. on Artificial Intelligence (AAAI)*, pages 6006–6013, 2019.
2. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.
3. D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 9:505–525, 2007.
4. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.
5. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018.
6. N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. https://arxiv.org/abs/1709.10207.
7. K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014. Technical Report. http://arxiv.org/abs/1406.1078.
8. V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.
9. M. Cisse, Y. Adi, N. Neverova, and J. Keshet. Houdini: Fooling Deep Structured Visual and Speech Recognition Models with Adversarial Examples. In *Proc. 30th Advances in Neural Information Processing Systems (NIPS)*, pages 6977–6987, 2017.
10. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. 5th Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978.
11. J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018. Technical Report. http://arxiv.org/abs/1810.04805.
12. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari. Output Range Analysis for Deep Neural Networks. In *Proc. 10th NASA Formal Methods Symposium (NFM)*, pages 121–138, 2018.
13. R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.

14. Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, 2020.
15. J. Elman. Finding Structure in Time. *Cognitive Science*, pages 179–211, 1990.
16. R. Floyd. Assigning Meanings to Programs. In *Program Verification*, pages 65–81. Springer, 1993.
17. T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
18. S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. NNSimplify: Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, 2020.
19. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
20. I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In *Proc. 27th Advances in Neural Information Processing Systems (NIPS)*, pages 2672–2680, 2014.
21. D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
22. S. Gulwani and N. Jojic. Program Verification as Probabilistic Inference. In *Proc. 34th Symposium on Principles of Programming Languages (POPL)*, pages 277–289, 2007.
23. A. Hadid, N. Evans, S. Marcel, and J. Fierrez. Biometrics systems under spoofing attack: an evaluation methodology and lessons learned. *IEEE Signal Processing Magazine*, 32(5):20–30, 2015.
24. G. Heigold, I. Moreno, S. Bengio, and N. Shazeer. End-to-end Text-Dependent Speaker Verification. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*, pages 5115–5119, 2016.
25. G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
26. S. Hochreiter and J. Schmidhuber. Long Short-term Memory. *Neural Computation*, pages 1735–1780, 1997.
27. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
28. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
29. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
30. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles, (FVAV)*, pages 19–26, 2017.
31. G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.

32. Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 2019.
33. C. Ko, Z. Lyu, T. Weng, L. Daniel, N. Wong, and D. Lin. POPQORN: Quantifying Robustness of Recurrent Neural Networks. In *Proc. 36th IEEE Int. Conf. on Machine Learning and Applications (ICML)*, 2019.
34. F. Kreuk, Y. Adi, M. Cisse, and J. Keshet. Fooling End-to-End Speaker Verification with Adversarial Examples. In *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1962–1966, 2018.
35. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proc. 25th Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.
36. L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. https://arxiv.org/abs/1801.05950.
37. G. Lample and D. Chaplot. Playing FPS Games with Deep Reinforcement Learning. In *Proc. 31st Conference on Artificial Intelligence (AAAI)*, pages 2140–2146, 2017.
38. Z. Lipton, D. Kale, C. Elkan, and R. Wetzel. Learning to Diagnose with LSTM Recurrent Neural Networks. In *Proc. 4th Int. Conf. on Learning Representations (ICLR)*, 2016.
39. A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1706.07351.
40. R. Nallapati, B. Xiang, and B. Zhou. Sequence-to-Sequence RNNs for Text Summarization, 2016. Technical Report. http://arxiv.org/abs/1602.06023.
41. N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. Verifying Properties of Binarized Deep Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1709.06662.
42. T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks. Counterexample-Guided Approach to Finding Numerical Invariants. In *Proc. 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 605–615, 2017.
43. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
44. O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv. Decidability of Inferring Inductive Invariants. In *Proc. 43th Symposium on Principles of Programming Languages (POPL)*, pages 217–231, 2016.
45. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV)*, pages 703–719, 2011.
46. X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. Learning Loop Invariants for Program Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 7762–7773, 2018.
47. D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
48. G. Singh, T. Gehr, M. Püschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th Symposium on Principles of Programming Languages (POPL)*, 2019.
49. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.

50. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *Proc. 7th Int. Conf. on Learning Representations (ICLR)*, 2019.
51. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, pages 1599–1614, 2018.
52. P. Werbos. Generalization of Backpropagation with Application to a Recurrent Gas Market Model. *Neural Networks*, pages 339–356, 1988.
53. W. Xiang and T. Johnson. Reachability Analysis and Safety Verification for Neural Network Control Systems, 2018. Technical Report. http://arxiv.org/abs/1805.09944.
54. J. Yamagishi, C. Veaux, and K. MacDonald. CSTR VCTK Corpus: English Multi-speaker Corpus for CSTR Voice Cloning Toolkit, 2019. University of Edinburgh. https://doi.org/10.7488/ds/2645.

Incorrect by Construction: Fine Tuning Neural Networks for Guaranteed Performance on Finite Sets of Examples

Ivan Papusha[1], Rosa Wu[1,2], Joshua Brulé[1], Yanni Kouskoulas[1], Daniel Genin[1], and Aurora Schmidt[1]

[1] Johns Hopkins University Applied Physics Laboratory*
[2] Defense Nuclear Facilities Safety Board**

Abstract. There is great interest in using formal methods to guarantee the reliability of deep neural networks. However, these techniques may also be used to implant carefully selected input-output pairs. We present initial results on a novel technique for using SMT solvers to fine tune the weights of a ReLU neural network to guarantee outcomes on a finite set of particular examples. This procedure can be used to ensure performance on key examples, but it could also be used to insert difficult-to-find incorrect examples that trigger unexpected performance. We demonstrate this approach by fine tuning an MNIST network to incorrectly classify a particular image and discuss the potential for the approach to compromise reliability of freely-shared machine learning models.

Keywords: formal methods · neural networks · satisfiability modulo theory · constraint satisfaction · performance guarantees

1 Introduction

Advances in the construction and training of deep neural networks have transformed many problems in classification, machine learning, and autonomous systems. But the large number of internal degrees of freedom that make these networks so powerful can also prove to be a source of vulnerability—verifying that such complex systems always perform in an expected way is a daunting task. As a result, there is much interest in using automatic formal verification techniques that employ satisfiability modulo theories (SMT) to generate guarantees about the behavior of such networks.

SMT is a recently attractive technology because of practical solver advances and mature implementations. Leveraging a complete decision procedure, solvers can generate a network input that satisfies a given constraint (`sat`), or guarantee that no such input exists (`unsat`). By treating perturbations to the network as

* This work was supported by JHU/APL Internal Research and Development funds.
** The views expressed herein are solely those of the authors, and no official support or endorsement by the Defense Nuclear Facilities Safety Board or the U.S. Government is intended or should be inferred.

variable, we find that we may also use SMT to search for small modifications to the network itself that guarantee performance it did not already have.

In this work, we use Z3 [16] to embed a set of guaranteed input-output examples by taking advantage of the ample degrees of freedom in the biases. Our main contribution is to show that small bias perturbations can internally model intentionally-planted correct or incorrect input-output pairs with moderately reduced performance on the off-target examples. Our approach could be used to fine tune networks to guarantee performance on a critical set of examples, or to poison them with malicious triggers. Furthermore, the technique is constructive—we either exhibit specific bias perturbations satisfying prescribed constraints, or generate a verifiable proof artifact showing that none exist.

Prior work The rise in effective optimization techniques for producing adversarial examples has led to an explosion of interest in how to fool neural networks with inputs that are slight modifications of correctly classified examples. Much effort has been devoted to finding such adversarial examples in various neural networks [9,20]. This has inspired researchers to use SMT to verify or construct neural networks lacking such adversarial examples [1,3,4].

Like Reluplex and related approaches that use SMT to find adversarial examples or guarantee their absence [12,17,2,6,11,5], we restrict our attention to neural networks with piecewise affine activation layers. These include, for example, rectifier linear unit (ReLU) and HardTanh, but not sigmoid or softmax layers. However, instead of searching for perturbations on *inputs*, we search for perturbations on the network *biases*, thereby globally and tractably parameterizing all possible neural networks of a certain class.

Although we use MNIST as a running example, our patching approach inherits the adverse scaling of SMT with neural network size, likely precluding adoption to vision tasks in the near term. Our philosophy is therefore not just to retrain with a modified training set as in [10], but rather to globally and reliably optimize over the space of all neural networks that satisfy a set of constraints. This allows application in broader frameworks for design and verification of high reliability systems with formal guarantees on end-to-end behavior [17, §2].

As part of this paper, we review methods (§2) for translating a neural network into SMT constraints, and follow with detail on using the encoded network to generate adversarial inputs (§3), as well as adjusting the network parameters to implant guaranteed input-output pairs. We demonstrate this approach (§4) by implanting behavior in a small example network for digit classification. We conclude (§5) with a discussion of the potential for this method to scale to larger networks, as well as future work.

2 Neural Network as Constraints

The key insight to our approach is the observation that certain neural networks are well-suited to analysis via SMT, while still being expressive enough to perform calculations of interest, see *e.g.*, [17,2]. We encode the input-output relations of deterministic neural networks as quantifier free combinations of linear arithmetic constraints.

2.1 Piecewise Affine Networks

Consider a network $f : \mathbf{R}^n \to \mathbf{R}^m$, represented by a function $y = f(x;\theta)$, with parameters θ. The input x and output y are n- and m-dimensional real vectors, respectively. In a typical architecture, the neural network is designed as a sequential composition of alternating affine functions and piecewise affine (*e.g.*, ReLU, HardTanh) activations,

$$f = \beta_K \circ \alpha_K \circ \cdots \circ \beta_1 \circ \alpha_1. \tag{1}$$

Each affine function $\alpha_k : \mathbf{R}^{n_k} \to \mathbf{R}^{m_k}$ is parameterized by a dense m_k-by-n_k real weight matrix $W^{(k)}$ and an m_k-dimensional bias vector $b^{(k)}$,

$$\text{[Affine]} \quad \alpha_k(x; W^{(k)}, b^{(k)}) := W^{(k)}x + b^{(k)}. \tag{2}$$

Similarly, the activation functions $\beta_k : \mathbf{R}^{m_k} \to \mathbf{R}^{m_k}$ are piecewise affine. We consider componentwise ReLU and HardTanh activations, although any piecewise affine activation can be likewise treated,

$$\text{[ReLU]} \quad \beta_k(x)_i := \max(x_i, 0), \quad i = 1, \ldots, m_k, \tag{3}$$

$$\text{[HardTanh]} \quad \beta_k(x)_i := \max(\min(x_i, 1), -1), \quad i = 1, \ldots, m_k. \tag{4}$$

The input dimension of the network f is $n = n_1$ in the first layer, and the output dimension is $m = m_K$ in the last. The signal dimension can only change in the affine layers ($m_k \neq n_k$ in general), and remains the same through the activations ($m_k = n_{k+1}$). For convenience, we split the network parameters into weight and bias components $\theta = (\theta_{\text{weight}}, \theta_{\text{bias}})$,

$$\theta_{\text{weight}} = (W^{(1)}, \ldots, W^{(K)}) \in \mathbf{R}^{m_1 \times n_1} \times \cdots \times \mathbf{R}^{m_K \times n_K},$$
$$\theta_{\text{bias}} = (b^{(1)}, \ldots, b^{(K)}) \in \mathbf{R}^{m_1} \times \cdots \times \mathbf{R}^{m_K}.$$

SMT encoding We encode the neural network by introducing intermediate variables $x^{(1)}, \ldots, x^{(K+1)}, y^{(1)}, \ldots, y^{(K)}$ to hold the results of the compositions in (1). Specifically, for an input variable x and output variable y, the input-output relation of the neural network $y = f(x;\theta)$ is equivalent to

$$(x = x^{(1)}) \wedge \left(\bigwedge_{k=1}^{K} x^{(k+1)} = \beta_k(y^{(k)}) \wedge y^{(k)} = \alpha_k(x^{(k)}) \right) \wedge (y = x^{(K+1)}). \tag{5}$$

The affine layers are encoded as-is following (2),

$$\text{[Affine-Encoding]} \quad v = \alpha_k(u) \iff v = W^{(k)}u + b^{(k)}, \tag{6}$$

with variables u, v and parameters $W^{(k)}, b^{(k)}$.

To encode the activation functions, note that equality constraints involving 'min' and 'max' can be written as a logical combination of affine atoms:

$$\eta = \min(\xi, a) \iff [(\xi \geq a) \to (\eta = a)] \wedge [(\xi < a) \to (\eta = \xi)], \tag{7}$$

$$\eta = \max(\xi, b) \iff [(\xi < b) \to (\eta = b)] \wedge [(\xi \geq b) \to (\eta = \xi)]. \tag{8}$$

Accordingly, the piecewise affine activation functions are logical conjunctions over individual components,

$$\text{[ReLU-Encoding]} \quad v = \beta_k(u) \iff \bigwedge_{j=1}^{m_k} (v_j = \max(u_j, 0)), \tag{9}$$

$$\text{[HardTanh-Encoding]} \quad v = \beta_k(u) \iff \bigwedge_{j=1}^{m_k} (v_j = \max(\min(u_j, 1), -1)). \tag{10}$$

Put together, equations (6)–(10) can be substituted successively into equation (5), resulting in an encoding of the neural network (1) into a single formula consisting of conjunctions, disjunctions, and negations of affine atoms. Thus, any neural network constraint of the form $y = f(x; \theta)$, with variables x and y, and parameters θ, corresponds to a conjunction of constraints of the form (5).

2.2 Using Pretrained PyTorch modules

To automate our experiments, we developed a Python package, LANTERN ("safer than a torch"), which converts common neural network modules from the popular PYTORCH library [18] to variables and constraints that can be further manipulated with an SMT solver such as Z3 [16]. We assume that the (trained) network is represented as a `Sequential` module, a PYTORCH container that holds other modules and applies them in sequence. We further assume that the modules within a given `Sequential` instance are either `Linear`, `ReLU`, or `Hardtanh`.

For each module in a `Sequential`, LANTERN generates Z3 variables that correspond to the inputs and outputs of that module, and encodes the behavior of that module as affine constraints (see §2.1). In addition, it creates constraints that equate the output variables of each module with the input variables of the next module in the sequence. This process returns the input and output variables of the entire `Sequential`, as well as all the constraints that represent the internal modules.

The default settings of PYTORCH result in models parameterized by 32-bit floats, which can give computationally difficult SMT formulas. When the floats are losslessly cast to `Real`-sorted variables, formulas involving the neural network can be handled using Z3's linear real arithmetic solver. However, in practice we found that arbitrary precision calculations often dominated decision run times, meaning that computations involving even moderately sized networks benefited from a translation to IEEE floating-point arithmetic. Therefore, our software also supports quantizing networks into floating-point representations.

The `round_model()` function truncates the significand of the floating-point parameters of a trained network to a desired number of bits. This function provides an adjustable trade-off between the neural network's performance and the difficulty of the corresponding SMT problem. A rounded model remains a valid `Sequential` object, and can be run just like the original at inference time, albeit with reduced accuracy. By quantizing the model itself, we preserve a one-to-one correspondence between the SMT problem and the network, even though the rounded model is no longer equivalent to the original.

3 Method for Planting Examples

A common application of the SMT encoding (§2.1) is to find perturbations on an input that would result in a classifier misclassifying otherwise correct examples. The existence of techniques to find small perturbations is well documented [11,6]. We will briefly summarize these findings (§3.1) with an eye toward explaining our novel neural network modification strategy (§3.2).

3.1 Adversarial Input Generation

Consider a trained network f, which *correctly* classifies an input x^0 as y^0, so that specifically $y^0 = f(x^0; \theta)$ for the given input-output pair (x^0, y^0). We would like the network to instead output a specified y^1 for a perturbed input $x^0 + \Delta x$, where $y^0 \neq y^1$ and the perturbation magnitude $\|\Delta x\|$ is small.

In this setting, finding a minimal adversarial input amounts to solving the (nonconvex) optimization problem

$$\begin{array}{l} \text{minimize} \quad \|\Delta x\| \\ \text{subject to } f(x^0 + \Delta x; \theta) = y^1 \end{array} \tag{11}$$

over the variable $\Delta x \in \mathbf{R}^n$. This will give a smallest perturbation Δx on the input that is enough to get the network to misclassify x^0 as y^1. We target the ℓ_∞ norm $\|\cdot\|$, because it can be represented with piecewise affine ('max') functions, although many other norms are possible. The parameters of the network θ remain constant throughout the adversarial input generation process.

Optimal perturbation The objective in (11) can be minimized with bisection by posing a sequence of queries to the SMT solver. Specifically, define the formula

$$F(\alpha) = \exists \Delta x \in \mathbf{R}^m.\ (y^1 = f(x^0 + \Delta x; \theta)) \wedge (\|\Delta x\| \leq \alpha).$$

If, for a given value of $\alpha \in [\alpha_-, \alpha_+]$ (where $F(\alpha_+)$ is `sat` and $F(\alpha_-)$ is `unsat`), the formula $F(\alpha)$ is `sat`, then we know that at the optimum $\|\Delta x^\star\| \leq \alpha$; we should therefore decrease the upper bound to $\alpha_+ := \alpha$, and determine the satisfiability of $F((\alpha_+ + \alpha_-)/2)$, say. Otherwise if $F(\alpha)$ is `unsat`, then a valid input perturbation must have norm no less than α; therefore, to make the network misclassify x^0 as y^1 we should increase the lower bound to $\alpha_- := \alpha$, and try again. This way, a minimal value of $\|\Delta x\|$ can be determined within an error ϵ in $O(\log_2(1/\epsilon))$ bisection steps.

Class membership Encoding the constraint $y^1 = f(x^0 + \Delta x; \theta)$ in (11) deserves special attention in the case of classifiers, because class membership must be encoded with set membership (*e.g.*, lying on the correct side of a decision surface). For example, for $m = 10$ (MNIST digit classification problem), we identify

the output indices with the classes '1', '2', ..., '9', '0'. For example, the network output

$$f(x^0;\theta) = \begin{bmatrix} 0.01 \\ 0.95 \\ \dots \\ 0.02 \end{bmatrix}$$

is interpreted as '2', because the second component has maximal value (softmax layers are disallowed in linear SMT theories). A class equality constraint like $y =$ '7' is in reality a requirement on the seventh component of y to be maximal,

$$(y_7 > y_1) \wedge \cdots \wedge (y_7 > y_6) \wedge (y_7 > y_8) \wedge \cdots \wedge (y_7 > y_{10}). \tag{12}$$

A class membership constraint is thus a conjunction of affine constraints.

Forcing correctness The same adversarial input generation technique can be used if the correctness senses of y^0 and y^1 are switched: when the network *incorrectly* classifies x^0 as y^0, then solving the optimization problem (11) is akin to finding a minimum-size perturbation on the input that will *force* the output to the desired correct value y^1. In this case, the network outputs a correct value with a small input perturbation, even if it originally failed to do so.

3.2 Adversarial Network Modification

The idea of forcing output values introduced in the previous section can similarly be used to *patch* the network parameters to achieve desired performance on specified input-output pairs. The key difference lies in patching the biases only, meanwhile keeping the weights fixed.

Bias patching Consider a supervised task with a training database of input-output pairs $D = \{(x, y)\} \subset \mathbf{R}^n \times \mathbf{R}^m$. We would like to keep the neural network output values the same on a finite set $D^{\text{keep}} \subset \mathbf{R}^n \times \mathbf{R}^m$, and force a change on a finite set $D^{\text{change}} \subset \mathbf{R}^n \times \mathbf{R}^m$ of values. It is not necessary that D^{keep} or D^{change} be subsets of D, but we require that $D^{\text{keep}} \cap D^{\text{change}} = \emptyset$. The procedure for patching the network biases consists of two conceptual steps:

1. [Train] Classically train (*e.g.*, using stochastic gradient descent) a ReLU network $f(x;\theta)$ on the database D, obtaining the parameter vector $\theta = (\theta_{\text{weight}}, \theta_{\text{bias}})$ as a starting point.
2. [Patch] Keeping the weight component θ_{weight} fixed, modify the network from Step 1 by solving the optimization problem

$$\begin{aligned} &\text{minimize} \quad \|\Delta\theta\| \\ &\text{subject to } y = f(x;\theta+\Delta\theta), \quad \text{for all } (x,y) \in D^{\text{keep}}, && (13)\\ &\qquad\qquad y' = f(x';\theta+\Delta\theta), \quad \text{for all } (x',y') \in D^{\text{change}}, && (14)\\ &\qquad\qquad \Delta\theta_{\text{weight}} = 0. && (15) \end{aligned}$$

over the variables $\Delta\theta = (\Delta\theta_{\text{weight}}, \Delta\theta_{\text{bias}})$.

Classical neural network training will not (typically) result in a parameter vector θ that correctly assigns all points in D. However, the SMT patching procedure will force the values in D^{keep} and D^{change}, or otherwise return a proof that a network modification of the prescribed type is impossible.

Linear arithmetic The biases can can be patched because they enter affinely into the neural network constraints (5) (whereas the weights enter multiplicatively). As a result, bias perturbation variables can be added at each α_k network layer while still using a decision procedure based on linear arithmetic, *cf.* (6),

$$v = \alpha_k(u) \iff v = W^{(k)}u + b^{(k)} + \Delta\theta^{(k)}_{\text{bias}}. \tag{16}$$

Staying within a linear decision theory helps performance, although we expect weight modification with nonlinear theories (and multiplicative terms) to be practical in small networks [7].

A key scaling challenge lies in keeping the fewest number of constraints in (13) and (14), since there are as many instances of the fully encoded neural network in the optimization problem as there are examples in $D^{\text{keep}} \cup D^{\text{change}}$. To help this potential difficulty, it is desirable to keep $|D^{\text{keep}}|$ and $|D^{\text{change}}|$ small.

4 Experiments

Following the outlined approach (§3), we encoded small- and medium-sized neural networks to test the generation of adversarial inputs in realistic cases. Additionally, we modified the medium-sized network to give prescribed outputs for prescribed inputs. We performed experiments using the MNIST database of handwritten digits [15].

Because the computational complexity of the SMT decision procedure is heavily dependent on the total number of units in the network under consideration, the 28-by-28 pixel grayscale images of handwritten digits were flattened to vectors of length 784, and dimensionally reduced with Principal Component Analysis (PCA) by selecting the top-30 or top-100 components, depending on the experiment. To improve runtime of the solver, network weights and biases were rounded using the `round_model()` function (§2.2).

4.1 Adversarial Input Generation

The "small" MNIST classifier architecture is shown below.

$$[\text{image}] \overset{784}{\to} \boxed{\text{PCA}} \overset{30}{\to} \underbrace{\Big[[x] \overset{30}{\to} \boxed{\text{Linear}} \overset{10}{\to} \boxed{\text{ReLU}} \overset{10}{\to} \boxed{\text{Linear}} \overset{10}{\to} \boxed{\text{ReLU}} \overset{10}{\to} [y]\Big]}_{y=f(x;\theta)}$$

For the first experiment, the top-30 principal component network was probed to see if there exist adversarial inputs to make the network misclassify specific images. The image representations with reduced dimensionality are treated as

inputs to f, a four-layer PyTorch `Sequential` model composed of alternating `Linear` and `ReLU` modules trained with stochastic gradient descent. The components of the output vector are used to decide the digit class according to (12). This small network achieves 72.0% accuracy on the validation data.

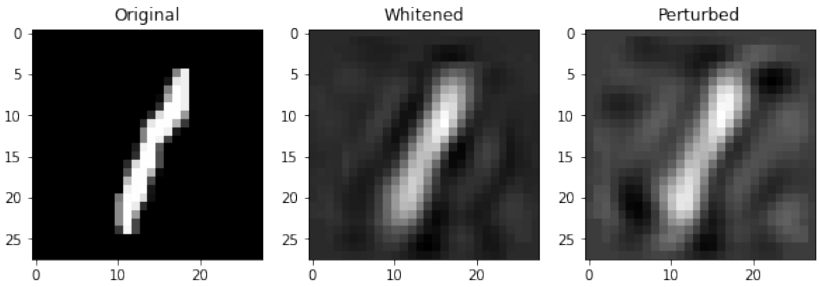


Fig. 1. The network correctly classifies the original MNIST image (left) as '1' by observing the top-30 PCA components (middle). The input-perturbed image is misclassified as '7' (right).

We use Lantern to encode the small network as Z3 constraints. Then we find a reduced-dimensionality representation of an image of a '1' that the network can correctly classify, and force that image to misclassify as '7'. Figure 1 shows the original image, the image after PCA compression, and finally the misclassified version with an adversarial perturbation having magnitude $\|\Delta x\|_\infty = 0.4$.

4.2 Adversarial Network Modification

In this experiment, our goal was to modify the network biases such that several '1' instances would be misclassified as '7', while the other classes continued to be accurately classified. To test scalability and network quantization, we used a slightly larger, top-100 component PCA compressed data set with a similar neural network architecture:

$$[\text{image}] \overset{784}{\to} \boxed{\text{PCA}} \overset{100}{\to} \underbrace{\left[\begin{array}{ccccccccc} & +\Delta\theta_{\text{bias}} & & & & +\Delta\theta_{\text{bias}} & & & \\ & \downarrow & & & & \downarrow & & & \\ [x] \overset{100}{\to} & \boxed{\text{Linear}} & \overset{33}{\to} & \boxed{\text{ReLU}} & \overset{33}{\to} & \boxed{\text{Linear}} & \overset{10}{\to} & \boxed{\text{ReLU}} & \overset{10}{\to} [y] \end{array}\right]}_{y=f(x;\theta+\Delta\theta)}$$

Prior to any bias modifications, this medium-sized network had a 93.0% overall classification accuracy. Table 1 breaks down the accuracy by each digit.

We encoded the linear layers with constraints of the form

$$y^{(k)} = W^{(k)}x^{(k)} + b^{(k)} + \Delta\theta^{(k)}_{\text{bias}} \tag{17}$$

where $W^{(k)}$ and $b^{(k)}$ are the (fixed) network weights and biases for the layer, $x^{(k)}$ and $y^{(k)}$ are the (variable) inputs and outputs of each layer, and $\Delta\theta_{\text{bias}}^{(k)}$ are (variable) bias perturbations.

To construct D^{keep} and D^{change}, we chose one set of digits '0' through '9', which the original network classified correctly, and added them to D^{keep}. This resulted in $|D^{\text{keep}}| = 10$ constraints of type (13). We also found a specific '1' image and set its classification target to '7'. This resulted in $|D^{\text{change}}| = 1$ constraint of type (14). Additionally, we added constraints $\|\Delta\theta_{\text{bias}}\|_\infty \leq 0.25$ to bound maximum parameter perturbations.

Guarantees with inherited performance The results of our experiment are summarized in Table 1, which shows the digit classification performance of the modified network. The accuracy shown for the modified network (second column) is an average of two different D^{keep} and D^{change} sets. These results indicate a considerable decrease in accuracy when classifying '1's, due to the forced prescription in D^{change}, along with moderately smaller accuracy differences in the other classes. Note that for the modified network, the eleven examples in D^{keep} and D^{change} are guaranteed to be at their prescribed, forced values.

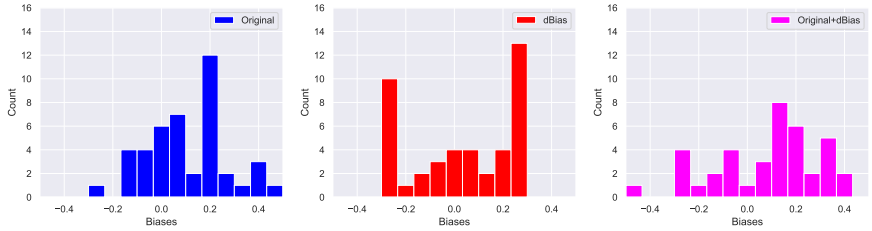
Table 1. Classification accuracy of each digit with the original network weights and biases, modified biases, and modified biases using a quantized model.

Digit	Accuracy (%)		
	Original	Modified	Quantized
'1'	97.6	73.9	68.9
'2'	92.4	89.6	91.1
'3'	91.1	89.3	78.4
'4'	94.0	84.2	86.1
'5'	86.8	77.6	91.4
'6'	95.1	90.5	93.3
'7'	91.6	95.0	93.4
'8'	91.1	85.9	74.8
'9'	91.8	92.5	89.2
'0'	97.9	96.7	98.6
Overall	93.0	87.4	89.2

It takes four hours for Z3 to find a satisfying assignment to $\Delta\theta_{\text{bias}}$ in the top-100 network using linear rational arithmetic.[3] After quantizing the network parameters to 10 bits with `round_model()`, run times went down to 30 minutes per floating point arithmetic decision call (8× speedup). Performance of the quantized model is shown in the last column of Table 1. The per-class accuracy for the quantized model is an average of three different D^{keep} and D^{change} sets.

[3] Tested on an Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz on a 64-bit Windows operating system with 32.0 GB installed RAM.

A visualization of how much the network biases changed is shown in Figure 2. Because the size distributions of the original and perturbed biases are so simi-



(a) original biases (b) $\Delta\theta_{\text{bias}}$ values (c) fine-tuned biases

Fig. 2. Histograms of (a) original network biases, (b) solutions for the bias perturbations for adversarial network modification, and (c) final modified biases.

lar, Figure 2 suggests that detection of this type of network tampering may be difficult. Thus, an across-the-board small change in network biases guaranteed the eleven specific examples to be classified in a user-prescribed way.

5 Conclusion and Future Work

In this work, we have shown how to use SMT to implant behaviors in neural networks that use piecewise affine activations. In doing so, we also detailed a method for automatically encoding PyTorch networks into Z3 constraints. We computed bias perturbations for a relatively small neural network that performs classification on the MNIST data set. We plan to extend this approach to larger neural networks, such as deep convolutional networks for image recognition and deep reinforcement learning networks. In many deep networks, for a particular input, only a small fractions of neurons end up contributing to the output [8,19]. Thus we can (i) select a subset of neurons to modify in a large network, and (ii) improve the efficiency of the decision procedure, by abstracting the majority of the network and locally optimizing the biases of selected neurons.

We plan to test a solver algorithm better optimized to the specific constraint solution problem, thereby increasing the scale of networks that can be modified and whose performance can be guaranteed. Because there are a wealth of networks of modest size, especially those that perform simple autonomous control, we see value to this approach despite SMT's unfavorable computational scaling. Further research on this technique will aid both in understanding the pitfalls of downloading and using freely shared pretrained neural networks, as well as the potential for verifying the provable reliability of neural networks in the loop.

Acknowledgments The authors would like to thank Dr. Kiran Karra and Mr. Chace Ashcraft for discussions on this topic and constructive comments on the approach.

References

1. Bohrer, B., Tan, Y.K., Mitsch, S., Sogokon, A., Platzer, A.: A formal safety net for waypoint-following in ground robots. IEEE Robotics and Automation Letters **4**(3), 2910–2917 (2019)
2. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Kumar, M.P.: A unified view of piecewise linear neural network verification (2017), arXiv:1711.00455 [cs.AI]
3. Carlini, N., Katz, G., Barrett, C., Dill, D.L.: Provably minimally-distorted adversarial examples (2017), arXiv:1709.10207 [cs.LG]
4. Cheng, C.H., Nührenberg, G., Ruess, H.: Maximum resilience of artificial neural networks. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis. pp. 251–268. Springer (2017)
5. Dutta, S., Chen, X., Jha, S., Sankaranarayanan, S., Tiwari, A.: Sherlock - a tool for verification of neural network feedback systems. In: ACM International Conference on Hybrid Systems Computation and Control (HSCC). ACM Press (2019)
6. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Automated Technology for Verification and Analysis, pp. 269–286. Springer (2017)
7. Gao, S., Avigad, J., Clarke, E.M.: δ-Complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning. pp. 286–300. Springer (2012)
8. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. pp. 315–323 (2011)
9. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: Bengio, Y., LeCun, Y. (eds.) International Conference on Learning Representations (ICLR) (2015)
10. Gu, T., Dolan-Gavitt, B., Garg, S.: BadNets: Identifying vulnerabilities in the machine learning model supply chain (2017), arXiv:1708.06733 [cs.CR]
11. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Computer Aided Verification, pp. 3–29. Springer (2017)
12. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 97–117. Springer (2017)
13. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability. Addison–Wesley (2015)
14. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, 2 edn. (2016)
15. LeCun, Y., Cortes, C., Burges, C.J.: MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/ (2010), [Online; accessed 20-April-2020]
16. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
17. Papusha, I., Topcu, U., Carr, S., Lauffer, N.: Affine multiplexing networks: System analysis, learning, and computation (Apr 2018), arXiv:1805.00164 [math.OC]
18. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019)

19. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: International Conference on Learning Representations (ICLR) (2019)
20. Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial examples: Attacks and defenses for deep learning. IEEE Transactions on Neural Networks and Learning Systems **30**(9), 2805–2824 (2019)

Robustness Verification for Ensemble Stumps and Trees

Hongge Chen[2]*, Yihan Wang[3]*, Huan Zhang[1]*, Si Si[4], Yang Li[4], Duane Boning[2], and Cho-Jui Hsieh[1]

[1] University of California, Los Angeles, USA
[2] Massachusetts Institute of Technology, Massachusetts, USA
[3] Tsinghua University, China
[4] Google Research, USA
* Equaly contributed, ranked by alphabetical order

Abstract. We study the robustness verification problem for ensemble decision stumps and trees, including random forest, gradient boosting trees, and Adaboost. Although these models are widely used in practice, there is very limited understanding on how to formally verify the robustness of those models. In this study, we aim to give a comprehensive complexity analysis as well as provide efficient verification algorithms. For ensemble decision stumps, we show that exact robustness verification with L_p norm ball is NP-complete for $p \in (0, \infty)$, while polynomial time algorithms exist for $p = 0$ and $p = \infty$. Approximation algorithms based on dynamic programming are then developed for verifying ensemble stumps for $p \in (0, \infty)$. For ensemble decision trees, it has been proved that exact robustness verification is NP-complete, and the existing verification approach is based on MILP, which does not scale to large-scale problems. We show that ensemble tree verification can be cast as a max-clique problem on a multi-partite graph with bounded boxicity, and by exploiting the boxicity of the graph, we develop an efficient multi-level verification algorithm that can give tight lower bounds on robustness of ensemble decision trees, while allowing iterative improvement and any-time termination.

1 Introduction

Machine learning verification aims to develop methods to bound the behavior of a model within a given input set, and they have become fundamental tools for verifying robustness and safety properties of given models. In this paper, we study the robustness verification problem of ensemble decision stumps and trees, which covers several important machine learning models such as AdaBoost, Random Forests (RFs) and Gradient Boosted Decision Trees GBDTs). These models have been widely used in practice [7, 12, 22] and recent studies have demonstrated that they are vulnerable to adversarial perturbations [10, 8, 6], but there is limited understanding on how to efficiently verify them.

We focus on the robustness verification problem, which can be defined as finding the minimum adversarial perturbation within a given input region (usually an ℓ_p norm ball). [10] showed that computing minimum adversarial perturbation for tree ensemble is NP-complete in general, and they proposed a Mixed-Integer Linear Programming (MILP) based approach to compute the minimum adversarial perturbation. Although exact verification is NP-hard for general tree ensemble, in order to have an efficient verification algorithm for real applications we seek to answer the following questions:

- Do we have polynomial time algorithms for exact verification under some special circumstances?
- For general tree ensemble models with a large number of trees, can we efficiently compute meaningful lower bounds on robustness while scaling to large tree ensembles?

In this paper, we provide the answers to the above-mentioned questions. Our contributions can be summarized below:

- **Robustness Verification for Ensemble Decision Stumps**: For an ensemble of decision stumps (trees with depth 1), we show that there is a fundamental different between the complexity of verifying ℓ_p norm ball with different p. When $p \in (0, \infty)$, we prove that ℓ_p norm verification problem is NP-complete while polynomial time algorithms exist for $p = 0, \infty$. However, we are able to propose an efficient dynamic programming algorithm that can compute a reasonably tight verification bound efficiently for general p.
- **Robustness Verification for Ensemble Decision Trees**: we show that for a single decision tree, robustness verification can be done exactly in linear time. Then we show that for an ensemble of K trees, the verification problem is equivalent to finding the maximum cliques in a K-partite graph, and the graph is in a special form with boxicity equal to the input feature dimension. Therefore, for low-dimensional problems, verification can be done in polynomial time with maximum clique searching algorithms. Finally, for large-scale tree ensembles, we propose a multiscale verification algorithm by exploiting the boxicity of the graph, which can give tight lower bounds on robustness.

2 Background and Related Work

Assume $F : \mathbb{R}^d \to \{1, \ldots, C\}$ is a C-way classification model, given a correctly classified example $\boldsymbol{x}_0$ with $F(\boldsymbol{x}_0) = y_0$, an adversarial perturbation is defined as $\boldsymbol{\delta} \in \mathbb{R}^d$ such that $F(\boldsymbol{x}_0 + \boldsymbol{\delta}) \neq y_0$.

Definition 1 (Robustness Verification Problem). *Given $F, \boldsymbol{x}_0$ and a perturbation radius ϵ, the robustness verification problem aims to determine whether there exists an adversarial example within ϵ ball around $\boldsymbol{x}_0$. In the other word, determine whether the following statement is true:*

$$\exists \boldsymbol{\delta} \text{ s.t. } \|\boldsymbol{\delta}\|_p < \epsilon \text{ and } F(\boldsymbol{x} + \boldsymbol{\delta}) \neq y_0. \tag{1}$$

Exactly solving (1) is usually hard, especially for deep neural networks [11, 20]. **Adversarial attack** algorithms are developed to find an adverarial perturbation $\boldsymbol{\delta}$ that satisfies (1). For example, several widely used attacks have been developed for attacking neural networks [4, 13, 9]. However, adversarial attacks can only find adversarial examples which do not provide a **sound** safety guarantee — even if an attack fails to find an adversarial example, it does not imply no adversarial example exists. Therefore, recent researches have been studied the sound solution to (1) and using them to evaluate safety of a model, leading to the recent developments of robustness verification.

Robustness verification aims to provide a **sound** answer to (1), which means a valid verification algorithm should answer no to (1) only when the existence of adversarial example can be disapproved. A tighter verification algorithm will be able to disaprove (1) for a larger ϵ than looser algorithms. For neural network, it has been shown that solving (1) exactly is NP-complete (for ReLU networks), and thus many recent works have been focusing on developing an efficient and reasonable tight robustness verification algorithm for neural networks [21, 23, 20, 17, 19, 18]. Most of them are following the linear relaxation based approach, where they find linear upper and lower bounds of output neurons with respect to input neurons and then try to answer (1) based on these. However, all of these algorithms are specifically designed for neural networks and cannot be extended to ensemble trees.

Robustness verification for tree ensembles Since ensemble trees are discrete step functions, none of the neural network verification algorithms can be applied. Specialized algorithms is required for verifying tree ensembles. Robustness evaluation and verification is first studied in [10], where they showed that ensemble tree verification is NP-complete when there are multiple trees with depth ≥ 2. An integer programming method was proposed to compute (1) in exponential time. Later on in [2], a single deicion tree is verified for evaluating robustness of an RL policy. A recent work [1] provides a certified defense algorithm for training tree ensembles against ℓ_∞ perturbation, and their algorithm implicitedly use the fact the ℓ_∞ robustness verification for ensemble stumps can be computed efficiently.

3 Robustness Verification for Ensemble Decision Trees and Stumps

3.1 Verification for a single decision tree

We first consider the simplified case with a single decision tree. Assume the decision tree has n nodes and for a given example $\boldsymbol{x}$ with d features, starting from the root, $\boldsymbol{x}$ traverses the decision tree model until reaching a leaf node. Each internal node i determines whether $\boldsymbol{x}$ will be passed to left or right child by checking $\mathbf{I}(x_{t_i} > \eta_i)$, and each leaf node has a value v_i indicating the prediction value of the tree.

If we define B^i as the set of $\boldsymbol{x} \in \mathcal{X}$ that can reach node i, due to the decision tree structure, B^i can be represented as a d-dimensional box:

$$B^i = (l_1^i, r_1^i] \times \cdots \times (l_d^i, r_d^i]. \tag{2}$$

The box can be computed efficiently in linear time by traversing the tree. The detailed algorithm can be found in Appendix A.

We aim to certify whether there exists any misclassified points under perturbation $\|\delta\|_p \leq \epsilon$. We can enumerate boxes for all the leaf nodes and check the minimum distance from $\boldsymbol{x}_0$ to each box. The following proposition shows that the ℓ_p norm distance between a point and a box can be computed in $O(d)$ time, and thus the exact robustness verification problem for a single tree can be solved in $O(dn)$ time.

Proposition 1. *Given a box* $B = (l_1, r_1] \times \cdots \times (l_d, r_d]$ *and a point* $\boldsymbol{x} \in \mathbb{R}^d$*. The closest* ℓ_p *distance from x to B is* $\|z - x\|_p$ *where:*

$$z_i = \begin{cases} x_i, & l_i \le x_i \le u_i \\ l_i, & x_i < l_i \\ u_i, & x_i > u_i. \end{cases} \tag{3}$$

3.2 Ensemble Decision Stumps

We assume there are T decision stumps and the i-th decision stump gives the prediction

$$f^i(x) = \begin{cases} w_l^i & \text{if } x_{t_i} < \eta^i \\ w_r^i & \text{if } x_{t_i} \ge \eta^i. \end{cases}$$

The prediction of decision stump ensemble $F(x) = \sum_i f^i(x)$ can be decomposed into each feature in the following way. For each feature j, assume $j_1, \ldots, j_{T_j}$ are the decision stumps using feature j, we can collect all the thresholds $[\eta^{j_1}, \ldots, \eta^{j_{T_j}}]$. Without loss of generality, assume $\eta^{j_1} \le \cdots \le \eta^{j_{T_j}}$ then the prediction values assigned in each interval can be denoted as

$$g^j(x_j) = v^{j_t} \quad \text{if } \eta^{j_t} < x_j \le \eta^{j_{t+1}} \tag{4}$$

where

$$v^{j_t} = w_l^{j_1} + \cdots + w_l^{j_t} + w_r^{j_{t+1}} + \cdots + w_r^{j_{T_j}}.$$

The overall prediction can be written as summation over the predicted values of each feature:

$$F(x) = \sum_{j=1}^{d} g^j(x_j), \tag{5}$$

and the final prediction is given by $y = sgn(F(x))$.

ℓ_∞ ensemble stump verification Due to the separability of (5), the ℓ_∞ norm perturbation can be done easily in linear time. For each feature j, we just need to check the worst-case perturbation within the range $(x_j - \epsilon, x_j + \epsilon)$ and this can be done by a linear scan through the thresholds $\eta^{j_1}, \ldots, \eta^{j_{T_j}}$. Therefore the verification can be done in polynomial time. This algorithm is implicitly mentioned in [1] for conducting ℓ_∞ certiied defense for tree ensembles.

ℓ_0 ensemble stump verification Assume $F(x)$ is positive and we want to make it the most negative by perturbing δ features (in this case, δ should be an integer). For each feature j, we want to know *the maximum decrease of prediction value by changing this feature,* which can be computed as

$$c^j = \min_t v^{j_t} - g^j(x_j), \tag{6}$$

and we should choose δ features with smallest c^j values to perturb. Let S_δ denotes the set with δ smallest c^j values, we have

$$\min_{\|x-x'\|_0 \le K} F(x') = F(x) + \sum_{i \in S_K} c^j. \tag{7}$$

Therefore verification can be done exactly in $O(T + d)$ time.

ℓ_p ensemble stump verification The difficulty of ℓ_p norm robustness verification is that the perturbations on each feature are correlated, so we can't separate all the features. In the following, we prove that the exact ℓ_p norm verification is NP-complete by showing a reduction from Knapsack to ℓ_p norm ensemble stump verification. This shows that ℓ_p norm verification can belong to a different complexity class compared with the ℓ_∞ norm case. The proof can be found in Appendix B, where we make a connection between ensemble stump verification and Knapsack problem.

Theorem 1. *Exact ℓ_p norm robustness verification (solving eq* (1)*) for an ensemble decision stump is NP-complete when $p \in (0, \infty)$.*

Although it is impossible to solve ℓ_p verification for decision stumps in polynomial time, we show an upper bound of this can be solved in polynomial time by dynamic programming, inspired by the pseudo-polynomial time algorithm for Knapsack.

Let $\eta^{j_1}, \ldots, \eta^{j_{T_j}}$ be the thresholds for feature j and $v^{j_1}, \ldots, v^{j_{T_j}}$ be the corresponding values, our dynamic programming maintains the following value for each ϵ: "given maximal ϵ perturbation to the first j features, what's the minimal prediction of the perturbed x". We denote this value as $D(\epsilon, j)$, then the following recursion holds:

$$D(\epsilon, j+1) = \min_{\delta \in [0, \epsilon]} D(\epsilon - \delta, j) + C(\delta, j+1),$$

where $C(\delta, j+1) := \min_{|x'_j - x_j| < \delta} g^j(x'_j)$ which can be precomputed. Note that δ, ϵ can be real numbers so exactly running this DP requires exponential time. Our approximate algorithm allows ϵ, δ only up to certain precision. If we choose precision ν, then we only consider values $\nu, 2\nu, \ldots, P\nu$ (the smallest P with $P\nu > \epsilon$). To ensure the verification algorithm is sound, the recursion will become

$$D(a\nu, j+1) = \min_{b \in \{1, \ldots, a\}} D((a-b+1)\nu, j) + C(b\nu, j+1), \tag{8}$$

and the final solution should be $D(\lceil \epsilon \rceil, d)$ where $\lceil \epsilon \rceil := T\nu$ means rounding ϵ up to the closest grid. Note that the $+1$ term in the recursion is to ensure that the resulting value is a lower bound of the original solution. The verification algorithm can verify N samples in $O(N(Pd + T))$ time, in which d is dimension and P is the number of discretizations.

3.3 Ensemble Decision Trees: Connection to max clique finding

Now we discuss robustness verification for tree ensembles. Assuming the tree ensemble has K decision trees, we use $S^{(k)}$ to denote the set of leaf nodes of tree k and $m^{(k)}(x)$ to denote the function that maps the input example x to the leaf node of tree k according to

its traversal rule. Given an input example x, the tree ensemble will pass x to each of these K trees independently and x reaches K leaf nodes $i^{(k)} = m^{(k)}(x)$ for all $k = 1, \ldots, K$. Each leaf node will assign a prediction value $v_{i^{(k)}}$. For simplicity we consider the binary classification problem where the original sample is classified as negative and the goal is to find whether there exists an input in the ϵ-ball that will be classified as positive . We will first consider the ℓ_∞ ball verification problem (input region is an ϵ-radius ℓ_∞ ball around x).

We start by defining some notation: let $\mathbb{C} = \{(i^{(1)}, \ldots, i^{(K)}) \mid i^{(k)} \in S^{(k)}, \forall k = 1, \ldots, L\}$ to be all the possible tuples of leaf nodes and let $C(x) = [m^{(1)}(x), \ldots, m^{(K)}(x)]$ be the function that maps x to the corresponding leaf nodes. Therefore, a tuple $C \in \mathbb{C}$ directly determines the model prediction $\sum v_C := \sum_k v_{i^{(k)}}$. Now we define a valid tuple for robustness verification:

Definition 2. *A tuple $C = (i^{(1)}, \ldots, i^{(K)})$ is valid if and only if there exists an $x' \in Ball(x, \epsilon)$ such that $C = C(x')$.*

The robustness verification (1) can then be written as:

$$\textit{Does there exist a valid tuple } C \textit{ such that } \sum v_C > 0?$$

Next, we show how to model the set of valid tuples. We have two observations. First, if a tuple contains any node i with $\inf_{x' \in B^i}\{\|x - x'\|_\infty\} > \epsilon$, then it will be invalid. Second, there exists an x such that $C = C(x)$ if and only if $B^{i^{(1)}} \cap \cdots \cap B^{i^{(K)}} \neq \emptyset$, or equivalently:

$$(l_t^{i^{(1)}}, r_t^{i^{(1)}}] \cap \cdots \cap (l_t^{i^{(K)}}, r_t^{i^{(K)}}] \neq \emptyset, \quad \forall t = 1, \ldots, d.$$

We show that the set of valid tuples can be represented as cliques in a graph $G = (V, E)$, where $V := \{i | B^i \cap \text{Ball}(x, \epsilon) \neq \emptyset\}$ and $E := \{(i, j) | B^i \cap B^j \neq \emptyset\}$. In this graph, nodes are the leaves of all trees and we remove every leaf that has empty intersection with $\text{Ball}(x, \epsilon)$. There is an edge (i, j) between node i and j if and only if their boxes intersect. The graph will then be a K-partite graph since there cannot be any edge between nodes from the same tree, and thus maximum cliques in this graph will have K nodes. We define each part of the K-partite graph as V_k. Here a "part" means a disjoint and independent set in the K-partite graph. The following lemma shows that intersections of boxes have very nice properties:

Lemma 1. *For boxes $B^1, \ldots, B^K$, if $B^i \cap B^j \neq \emptyset$ for all $i, j \in [K]$, let $\bar{B} = B^1 \cap B^2 \cap \cdots \cap B^K$ be their intersection. Then $\bar{B}$ will also be a box and $\bar{B} \neq \emptyset$.*

The proof can be found in the Appendix C. Based on the above lemma, each K-clique (fully connected subgraph with K nodes) in G can be viewed as a set of leaf nodes that has nonempty intersection with each other and also has nonempty intersection with $\text{Ball}(x, \epsilon)$, so the intersection of those K boxes and $\text{Ball}(x, \epsilon)$ will be a nonempty box, which implies each K-clique corresponds to a valid tuple of leaf nodes:

Lemma 2. *A tuple $C = (i^{(1)}, \ldots, i^{(K)})$ is valid if and only if nodes $i^{(1)}, \ldots, i^{(K)}$ form a K-clique (maximum clique) in graph G constructed above.*

Therefore the robustness verification problem can be formulated as

$$\textit{Is there a maximum clique } C \textit{ in } G \textit{ such that } \sum v_C > 0? \tag{9}$$

This reformulation indicates that the tree ensemble verification problem can be solved by an efficient maximum clique enumeration algorithm. Some standard maximum clique searching algorithms can thus be applied here to perform verification:

- **Finding K-cliques in K-partite graphs:** Any algorithm for finding all the maximum cliques in G can be used. The classic B-K backtracking algorithm [3] takes $O(3^{\frac{m}{3}})$ time to find all the maximum cliques where m is the number of nodes in G. Furthermore, since our graph is a K-partite graph, we can apply some specialized algorithms designed for finding all the K-cliques in K-partite graphs [14, 15, 16].
- **Polynomial time algorithms exist for low-dimensional problems:** Another important property for graph G is that each node in G is a d-dimensional box and each edge indicates the intersection of two boxes. This implies our graph G is with "boxicity d" (see [5] for detail). [5] proved that the number of maximum cliques will only be $O((2m)^d)$ and it is able to find the maximum weight clique in $O((2m)^d)$ time. Therefore, for problems with a very small d, the time complexity for verification is actually polynomial.

3.4 An Efficient and Sound Verification Algorithm for Tree Ensemble

Practical tree ensembles usually have tens or hundreds of trees with large feature dimensions, so exact clique findings will take exponential time and will be too slow. We thus develop an efficient multi-level algorithm for computing verification bounds by further exploiting the boxicity of the graph.

Figure 1 illustrates the graph and how our multilevel algorithm runs. There are four trees and each tree has four leaf nodes. A node is colored if it has nonempty intersection with $\text{Ball}(x, \epsilon)$; uncolored nodes are discarded. To answer question (9), we need to compute the maximum $\sum v_C$ among all K-cliques, denoted by v^*. As mentioned before, for robustness verification we only need to compute an upper bound of v^* in order to get a lower bound of minimal adversarial perturbation. In the following, we will first discuss algorithms for computing an upper bound at the top level, and then show how our multi-scale algorithm iteratively refines this bound until reaching the exact solution v^*.

Bounds for a single level. To compute an upper bound of v^*, a naive approach is to assume that the graph is fully connected between independent sets (fully connected K-partite graph) and in this case the maximum sum of node values is the sum of the maximum value of each independent set:

$$\sum\nolimits_{k=1}^{|\tilde{V}|} \max_{i \in V_k} v_i \geq v^*. \tag{10}$$

Here we abuse the notation v_i by assuming that each node i in V_k has been assigned a "pseudo prediction value", which will be used in the multi-level setting. In the simplest case, each independent set represents a single tree, $V_k = S^{(k)}$ and v_i is the prediction of a leaf. One can easily show this is an upper bound of v^* since any K-clique in the graph

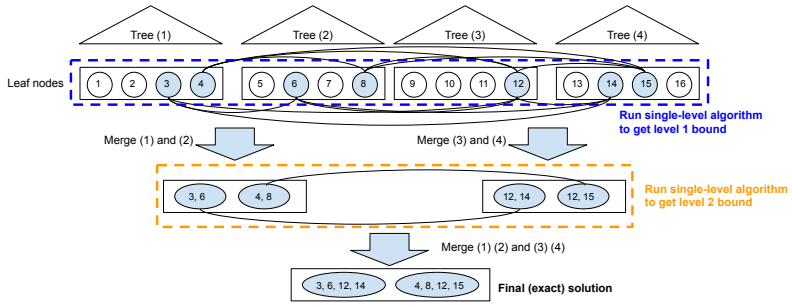


Fig. 1: The proposed multi-level verification algorithm. Lines between leaf node i on tree t_1 and leaf node j on t_2 indicate that their ℓ_∞ feature boxes intersect (i.e., there exists an input such that tree 1 predicts v_i and tree 2 predicts v_j).

is still considered when we add more edges to the graph, and eventually it becomes a fully connected K-partite graph.

Another slightly better approach is to exploit the edge information but only between tree t and $t+1$. If we search over all the length-K paths $[i^{(1)}, \ldots, i^{(K)}]$ from the first to the last part and define the value of a path to be $\sum_k v_{i^{(k)}}$, then the maximum valued path will be a upper bound of v^*. This can be computed in linear time using dynamic programming. We scan nodes from tree 1 to tree K, and for each node we store a value d_i which is the maximum value of paths from tree 1 to this node. At tree k and node i, the d_i value can be computed by

$$d_i = v_i + \max_{j: j \in V_{k-1} \text{ and } (j,i) \in E} d_j. \tag{11}$$

Then we take the max d value in the last tree. It produces an upper bound of v^*, since the maximum valued path found by dynamic programming is not necessarily a K-clique. Again $V_{k-1} = S^{(k-1)}$ in the first level but it will be generalized below.

Merging T independent sets To refine the relatively loose single-level bound, we partition the graph into K/T subgraphs, each with T independent sets. Within each subgraph, we find all the T-cliques and use a new "pseudo node" to represent each T-clique. T-cliques in a subgraph can be enumerated efficiently if we choose T to be a relatively small number (e.g., 2 or 3 in the experiments).

Now we exploit the boxicity property to form a new graph among these T-cliques (illustrated as the second level nodes in Figure 1). By Lemma 1, we know that the intersection of T boxes will still be a box, so each T-clique is still a box and can be represented as a pseudo node in the level-2 graph. Also because each pseudo node is still a box, we can easily form edges between pseudo nodes to indicate the nonempty overlapping between them and this will be a (K/T)-partite boxicity graph since no edge can be formed for the cliques within the same subgraph. Thus we get the level-2 graph. With the level-2 graph, we can again run the single level algorithm to compute a upper bound on v^* to get a lower bound of r^* in (1), but different from the level-1 graph, now we already considered all the within-subgraph edges so the bounds we get will be tighter.

The overall multi-level framework We can run the algorithm level by level until merging all the subgraphs into one, and in the final level the pseudo nodes will correspond to the K-cliques in the original graph, and the maximum value will be exactly v^*. Therefore, our algorithm can be viewed as an anytime algorithm that refines the upper bound level-by-level until reaching the maximum value. Although getting to the final level still requires exponential time, in practice we can stop at any level (denoted as L) and get a reasonable bound. In experiments, we will show that by merging few trees we already get a bound very close to the final solution. Algorithm 1 gives the complete procedure.

Algorithm 1: Multi-level verification framework

input : The set of leaf nodes of each tree, $S^{(1)}, S^{(2)}, \ldots, S^{(K)}$; maximum number of independent sets in a subgraph (denoted as T); maximum number of levels (denoted as L), $L \leq \lceil \log_T(K) \rceil$;

for $k \leftarrow 1, 2, \ldots, K$ **do**
  $U_k^{(0)} \leftarrow \{(A_i, B^{i^{(k)}}) | i^{(k)} \in S^{(k)}, A_i = \{i^{(k)}\}\}$;
  /* U is defined the same as in Algorithm **??**. At level 0, each V_k forms a 1-clique by itself. */
end
for $l \leftarrow 1, 2, \ldots, L$ **do**
  /* Enumerate all cliques in each subgraph at this level. Total $\lceil K/T^l \rceil$ subgraphs. */
  for $k \leftarrow 1, 2, \ldots, \lceil K/T^l \rceil$ **do**
    $U_k^{(l)} \leftarrow U_{(k-1)T+1}^{(l-1)}, U_{(k-1)T+2}^{(l-1)}, \ldots, U_{kT}^{(l-1)}$;
  end
end
for $k \leftarrow 1, 2, \ldots, \lceil K/T^L \rceil$ **do**
  /* Define an independent set V'_k for each $U_k^{(L)}$. In each V'_k, we create ''pseudo nodes'' which combines multiple nodes from lower levels, and assign ''pseudo prediction values'' to them. */
  $V'_k \leftarrow \{A \big| (A, B) \in U_k^{(L)}\}$; /* V'_k is a set of sets; each element in V'_k represents a clique. */
  /* Construct the ''pseudo prediction value'' for each element in V'_k by summing up all prediction values in the corresponding clique. */
  For all $A \in V'_k$, $v_A \leftarrow \sum_{i \in A} v_i$
end
$\bar{v} \leftarrow$ an upper bound of v^* using (10) or (11), given $\tilde{V} = \{V'_1, \cdots, V'_{\lceil K/T^L \rceil}\}$;
/* If $\lceil K/T^L \rceil = 1$, only 1 independent set left and each pseudo node represents a K-clique; (10) or (11) will have a trivial solution where v^* is the maximum v_A in $U_1^{(L)}$. */

Handling multi-class tree ensembles. For a multiclass classification problem, say a C-class classification problem, C groups of tree ensembles (each with K trees) are built for the classification task; for the k-th tree in group c, prediction outcome is denoted as $i^{(k,c)} = m^{(k,c)}(x)$ where $m^{(k,c)}(x)$ is the function that maps the input example x to a leaf node of tree k in group c. The final prediction is given by $\arg\max_c \sum_k v_{i^{(k,c)}}$. Given an input example x with ground-truth class c and an attack target class c', we extract $2K$ trees for class c and class c', and flip the sign of all prediction values for trees in group c', such that initially $\sum_t v_{i^{(t,c)}} + \sum_t v_{i^{(t,c')}} < 0$ for a correctly classified example. Then, we are back to the binary classification case with $2K$ trees, and we can still apply our multi-level framework to obtain a lower bound $\underline{r}_{(c,c')}$ of $r^*_{(c,c')}$ for this target attack pair (c, c'). Robustness of an untargeted attack can be evaluated by taking $\underline{r} = \min_{c' \neq c} \underline{r}_{(c,c')}$.

Dataset		ℓ_1 MILP		Ours ℓ_1 DP approx.			Ours vs. MILP		Ours ℓ_0 verification		
name	ϵ	robust err.	avg. time	precision	robust err.	avg. time	MILP/ours	speedup	avg. robust r^*	robust acc.	avg. time
breast-cancer	0.3	10.94%	.030s	0.01	10.94%	.00025s	1.00	120X	.04	95.62%	.0006
diabetes	0.05	35.06%	.017s	0.0002	35.06%	.0004s	1.00	40X	.0	100%	.0005s
Fashion-MNIST shoes	0.1	10.45%	.105s	0.005	10.55%	.0013s	.99	80.8X	2.09	16.35%	.010s
MNIST 1 vs. 5	0.3	3.30%	0.11s	0.005	3.35%	0.0013s	1.00	71X	3.33	3.50%	.010s
MNIST 2 vs. 6	0.3	9.64%	0.099s	0.005	9.69%	.0012s	.98	82X	1.22	26.43%	.012s

Table 1: **General ℓ_p-norm ensemble stump verification.** This table reports robust test error (robust err.) and average per sample time consumption (avg. time) of each method. For our proposed DP based verification, precision is also reported. For ℓ_0 verification, we also report average robust radius r^*, which means averagely how many features can be perturbed at most when the prediction stays the same.

Handling ℓ_p norm verification for $p < \infty$. In the ℓ_p norm case when $p < \infty$, the elimination step will lead to incorrect answer. Let $Ball_p(x, \epsilon)$ be the ℓ_p-norm ball with radius ϵ around x. For boxes $(B^1, \ldots, B^T)$, even if $B^i \cap B^j \neq \emptyset$ for all i, j and $B^i \cap Ball_p(x, \epsilon) \neq \emptyset$ for all i, it is not guaranteed that $B^1 \cap B^1 \ldots \cap B^T \cap Ball_p(x, \epsilon) \neq \emptyset$. Here we can generalize the framework to ℓ_p cases. In each layer from 1 to L, we split the T trees into groups of K. We find the K-size cliques in each group, which are non-intersected boxes, and form a group of new virtual nodes. After that, we keep the cliques which have nonempty intersection with $Ball_p(x, \epsilon)$ in the group. This group can then be treated as a virtual tree at the next level. This gives us an efficient algorithm for ℓ_p robustness verification.

4 Experimental Results

The results on real datasets demonstrate the proposed verification algorithms can compute a reasonably tight bound while being able to scale to large datasets. The statistics of the data sets are shown in Appendix D.

Robustness Verification for Ensemble Stumps. As discussed in the paper, we show ℓ_p norm verification has polynomial time algorithm only when $p = 0, \infty$. We thus pick $p = 0$ to demonstrate the algorithm works exactly and $p = 1$ to demonstrate our approximate verification algorithm can output reasonably tight bounds. Ensembles that are verified are trained with ℓ_∞ training proposed in [1].

For the ℓ_1 norm robustness verification problem, we have shown it's NP-complete to conduct exact verification. To demonstrate the tightness and efficiency of the proposed Dynamic Programming (DP) based verification, we also run the Mixed Integer Linear Programming [10] to get the exact robust bound which takes exponential time. In Table 1, we can find that the proposed DP algorithm gives almost exactly the same bound with MILP, while being $50 - 100$ times faster. This speedup guarantees its further applications in certified robust training. For the ℓ_0 norm robustness verification problem, we propose a linear time algorithm for conducting exact robustness verification. The results are also reported in Table 1. We can observe that the proposed method can conduct exact verification in less than 0.1 second.

Robustness Verification for Ensemble Trees. We evaluate our approximate ℓ_p verification method for tree ensembles on five real datasets. Ensembles that are verified are trained with ℓ_∞ training proposed in [1], each of which contains 20 trees. Again, we compare the proposed algorithm with MILP-based verification [10] which takes

Dataset		ℓ_1 MILP		Ours ℓ_1 approx.				Ours vs. MILP	
name	ϵ	robust err.	avg. time	K	L	robust err.	avg. time	robust err.	speedup
breast-cancer	0.3	8.03%	.036s	3	2	8.03%	.012s	1.00	3X
diabetes	0.05	33.12%	.027s	3	2	33.12%	.012s	1.00	2.25X
Fashion-MNIST shoes	0.1	10%	.091s	3	2	10%	.011s	1.00	8.23X
MNIST 1 vs. 5	0.3	4.20%	0.088s	3	2	4.20%	.011s	1.00	8X
MNIST 2 vs. 6	0.3	8.60%	.098s	3	2	8.80%	.012s	.98	8.17X

Table 2: **General ℓ_p-norm tree ensemble verification.** This table reports robust test error (robust err.) and average per sample time consumption (avg. time) of each method. For our generalized verification framework, K: size of cliques, and L: layer of the framework are also reported.

Dataset	MILP [10]		LP relaxation		Ours				Ours vs. MILP	
	avg. r^*	avg. time	avg. r_{LP}	avg. time	T	L	avg. r_{our}	avg. time	r_{our}/r^*	speedup
breast-cancer	.210	.012s	.064	.009s	2	1	.208	.001s	.99	12X
diabetes	.049	.061s	.015	.026s	3	2	.042	.018s	.86	3.4X
Fashion-MNIST	.014*	1150*s	.003*	898*s	2	1	.012	11.8s	.86	97X
MNIST	.011*	367*s	.003*	332*s	2	2	.011	5.14s	1.00	71X
MNIST 2 vs. 6	.057	23.0s	.016	11.6s	4	1	.046	.585s	.81	39X

Table 3: Average ℓ_∞ distortion over 500 examples and average verification time per example for three verification methods. Here we evaluate the bounds for **standard (natural) GBDT models**. Results marked with a start ("$\star$") are the averages of 50 examples due to long running time. T is the number of independent sets and L is the number of levels in searching cliques used in our algorithm. A ratio $\underline{r}_{our}/r^*$ close to 1 indicates better lower bound quality.

exponential time to get the exact bound. The results are presented in Table 2, and parameters of the proposed method (K and L) are also reported. We observe that the proposed verification method gets very tight robust error while being much faster than the MILP solver.

Note that as described in the previous section, the ℓ_∞ norm tree verification using our algorithm can be more efficient than a general ℓ_p norm. Here we then show the results on the ℓ_∞ norm verification in Table 3. Note that for this experiment we are trying to verify naturally trained tree ensembles by XGBoost. And instead of computing the robust error for a particular ϵ, we further conduct binary search for each sample to find the minimum ℓ_∞ norm adversarial perturbation, denoted as $\bar{r}_{\text{ours}}$, and compared with the optimal value by MILP, denoted by r^*. Furthermore, to show that directly relaxing MILP to Linear Programming (LP) won't give a tight lower bound, we also report its value $\bar{r}_{\text{LP}}$ in Table 3. The results show that our method can get a reasonably tight lower bound of MILP solution efficiently and much better than the LP relaxation.

5 Conclusion

In this paper, we study the robustness verification problem for ensemble stumps and trees. For both cases, we conduct a careful analysis of the computational complexity and propose efficient approximation algorithms to compute a sound robustness verification bound when the problem is NP-complete. Experimental results on real datasets demonstrate the efficiency and tightness of the proposed methods.

Bibliography

[1] M. Andriushchenko and M. Hein. Provably robust boosted decision stumps and trees against adversarial attacks. In *NeurIPS*, 2019.

[2] O. Bastani, Y. Pu, and A. Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pages 2494–2504, 2018.

[3] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[4] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[5] L. S. Chandran, M. C. Francis, and N. Sivadasan. Geometric representation of graphs in low dimension using axis parallel boxes. *Algorithmica*, 56(2):129, 2010.

[6] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh. Robust decision trees against adversarial examples. In *ICML*, 2019.

[7] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[8] M. Cheng, T. Le, P.-Y. Chen, J. Yi, H. Zhang, and C.-J. Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. In *ICLR*, 2019.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[10] A. Kantchelian, J. Tygar, and A. Joseph. Evasion and hardening of tree ensemble classifiers. In *ICML*, 2016.

[11] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[12] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[13] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[14] M. Mirghorbani and P. Krokhmal. On finding k-cliques in k-partite graphs. *Optimization Letters*, 7(6):1155–1165, 2013.

[15] C. A. Phillips, K. Wang, E. J. Baker, J. A. Bubier, E. J. Chesler, and M. A. Langston. On finding and enumerating maximal and maximum k-partite cliques in k-partite graphs. *Algorithms*, 12(1):23, 2019.

[16] M. Schneider and B. Wulfhorst. Cliques in k-partite graphs and their application in textile engineering. 2002.

[17] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. In *NIPS*, 2018.

[18] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL): 41, 2019.

[19] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. In *NIPS*, 2018.
[20] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel. Towards fast computation of certified robustness for relu networks. In *ICML*, 2018.
[21] E. Wong and J. Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *ICML*, 2018.
[22] H. Zhang, S. Si, and C.-J. Hsieh. GPU-acceleration for large-scale tree boosting. *SysML Conference*, 2018.
[23] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions. In *NIPS*, 2018.

A Algorithm for computing the box for each leaf

Conceptually, the main idea of our single tree verification algorithm is to compute a d-dimensional box for each leaf node such that any example in this box will fall into this leaf. Mathematically, the node i's box is defined as the Cartesian product $B^i = (l_1^i, r_1^i] \times \cdots \times (l_d^i, r_d^i]$ of d intervals on the real line. By definition, the root node has box $[-\infty, \infty] \times \cdots \times [-\infty, \infty]$ and given the box of an internal node i, its children's boxes can be obtained by changing only one interval of the box based on the split condition (t_i, η_i). More specifically, if p, q are node i's left and right child node respectively, then we set their boxes $B^p = (l_1^p, r_1^p] \times \cdots \times (l_d^p, r_d^p]$ and $B^q = (l_1^q, r_1^q] \times \cdots \times (l_d^q, r_d^q]$ by setting

$$(l_t^p, r_t^p] = \begin{cases} (l_t^i, r_t^i] & \text{if } t \neq t_i \\ (l_t^i, \min\{r_t^i, \eta_i\}] & \text{if } t = t_i \end{cases}, \quad (l_t^q, r_t^q] = \begin{cases} (l_t^i, r_t^i] & \text{if } t \neq t_i \\ (\max\{l_t^i, \eta_i\}, r_t^i] & \text{if } t = t_i. \end{cases} \tag{12}$$

After computing the boxes for internal nodes, we can also obtain the boxes for leaf nodes using (12). Therefore computing the boxes for all the leaf nodes of a decision tree can be done by a depth-first search traversal of the tree with time complexity $O(nd)$.

B Proof of Theorem 1

Proof. We show that a 0-1 Knapsack problem can be reduced to an ensemble stump verification problem. A 0-1 Knapsack problem can be defined as follows. Assume there are T items each with weight w_i and value v_i, the (decision version of) 0-1 Knapsack problem aims to determine whether there exists a subset of items S such that $\sum_{i \in S} w_i \leq C$ and with value $\sum_{i \in S} v_i \geq D$.

We construct a decision stump verification problem with T features and T stumps, where each decision stump corresponds to one feature. Assume x is the original example, we define each decision stump to be

$$g^i(s) = -v_i I(s > \eta_i) + \frac{D}{d}, \quad \text{where } \eta_i = x_i + w_i^{(1/p)}, \tag{13}$$

where $I()$ is the indicator function. The goal is to verify ℓ_p robustness with $\epsilon = C^{(1/p)}$. We need to show that this robustness verification problem outputs YES ($\min_{\|x-x'\|_p \leq \epsilon} \sum_i g^i(x'_i) < 0$) if and only if the Knapsack solution is also YES. If the verification found $v^* = \min_{\|x-x'\|_p \leq \epsilon} \sum_i g^i(x'_i) < 0$, let x' be the corresponding solution of verification, then we can choose the following S for 0-1 Knapsack:

$$S = \{i \mid x'_i > \eta_i\} \tag{14}$$

It is guaranteed that

$$\sum_{i \in S} w_i = \sum_{i \in S} |\eta_i - x_i|^p \leq \sum_i |x'_i - x_i|^p \leq \epsilon^p = C \tag{15}$$

and by the definition of g^i we have $\sum_i g^i(x'_i) = D - \sum_{i \in S} v_i \leq 0$, so this subset S will also be feasible for the Knapsack problem. On the other hand, if the 0-1 Knapsack problem has a solution S, for robustness verification problem we can choose x' such that

$$x'_i = \begin{cases} \eta_i & \text{if } i \in S \\ x_i & \text{otherwise} \end{cases}$$

By definition we have $\sum_i g^i(x'_i) = D - \sum_{i \in S} v_i < 0$. Therefore the Knapsack problem, which is NP-complete, can be reduced to ℓ_p norm decision stump verification problem with any $p \in (0, \infty)$ in polynomial time.

C Proof of Lemma 1

Proof. If we have K one dimensional intervals $I_1 = (l_1, r_1], I_2 = (l_2, r_2], \ldots, I_T = (l_K, r_K]$, we want to prove if every pair of them have nonempty overlap $I_1 \cap \cdots \cap I_K \neq \emptyset$. This can be proved by the following. Without loss of generality we assume $l_1 \leq l_2 \leq \cdots \leq l_K$. For each $k < K$, $I_k \cap I_K \neq \emptyset$ implies $l_K < r_k$. Therefore, $(l_T, \min(r_1, r_2, \ldots, r_K)]$ will be a nonempty set that is contained in $I_1, I_2, \ldots, I_K$. Therefore $I_1 \cap I_2 \cap \cdots \cap I_K \neq \emptyset$ and it is another interval.

This can be generalized to d-dimensional boxes. Assume we have boxes $B_1, \ldots, B_K$ such that $B_i \cap B_j \neq \emptyset$ for any i and j. Then for each dimension we can apply the above proof, which implies that $B_1 \cap B_2 \cap \cdots \cap B_K \neq \emptyset$ and the intersection will be another box.

D Data Statistics and Model Parameters

Table 4 presents data statistics and parameters for the models in the main text. The standard test accuracy is the model accuracy on natural, unmodified test sets.

Dataset	training set size	test set size	# of features	# of classes	# of trees	robust ϵ	depth robust	depth natural	standard test acc. robust	standard test acc. natural
breast-cancer	546	137	10	2	4	0.3	8	6	.978	.964
diabetes	614	154	8	2	20	0.2	5	5	.786	.773
Fashion-MNIST	60,000	10,000	784	10	200	0.1	8	8	.903	.903
MNIST	60,000	10,000	784	10	200	0.3	8	8	.980	.980
MNIST 2 vs. 6	11,876	1,990	784	2	1000	0.3	6	4	.997	.998

Table 4: The data statistics and parameters for the models presented in this paper.

Parallelization Techniques for Verifying Neural Networks

Haoze Wu[1], Alex Ozdemir[1], Aleksandar Zeljić[1], Kyle Julian[1], Ahmed Irfan[1], Divya Gopinath[2], Sadjad Fouladi[1], Guy Katz[5], Corina Pasareanu[3,4], and Clark Barrett[1]
[1]Stanford University, USA. [2]NASA Ames, KBR Inc. [3]NASA Ames, Moffett Field, CA.
[4]Carnegie Mellon University, USA. [5]The Hebrew University of Jerusalem, Israel.

***Abstract*—Inspired by recent successes with parallel techniques for solving Boolean satisfiability, we investigate a set of strategies and heuristics for leveraging parallelism to improve the scalability of neural network verification. We present a general description of the partitioning algorithm, implemented within the Marabou framework, and discuss its parameters and heuristic choices. In particular, we explore two novel partitioning strategies, that partition the input space or the phases of the neuron activations, respectively. We introduce a branching heuristic and a direction heuristic that are based on the notion of polarity. We also introduce a highly parallel pre-processing algorithm for simplifying neural network verification problems. An extensive experimental evaluation shows the benefit of these techniques on both existing and new benchmarks. A preliminary experiment with ultra-scaling our algorithm using a large distributed cloud-based platform also shows promising results.**

Under submission to Formal Methods in Computer-Aided Design (FMCAD) 2020

I. Introduction

Recent breakthroughs in machine learning, specifically the rise of *deep neural networks (DNNs)* [1], have expanded the horizon of real-world problems that can be tackled effectively. Increasingly, complex systems are created using machine learning models [2] instead of using conventional engineering approaches. Machine learning models are trained on a set of (labeled) examples, using algorithms that allow the model to capture their properties and generalize them to unseen inputs. In practice, DNNs can significantly outperform hand-crafted systems, especially in fields where precise problem formulation is challenging, such as image classification [3], speech recognition [4] and game playing [5].

Despite their overall success, the black-box nature of DNNs calls into question their trustworthiness and hinders their application in safety-critical domains. These limitations are exacerbated by the fact that DNNs are known to be vulnerable to *adversarial perturbations*, small modifications to the inputs that lead to wrong responses from the network [6], and real-world attacks have already been carried out against safety-critical deployments of DNNs [7, 8]. One promising approach for addressing these concerns is the use of formal methods to certify and/or obtain rigorous guarantees about DNN behavior.

Early work in DNN formal verification [9, 10] focused on translating DNNs and their properties into formats supported by existing verification tools like general-purpose *Satisfiability Modulo Theories* (SMT) solvers (e.g., Z3 [11], CVC4 [12]). However, this approach was limited to small toy networks (roughly tens of nodes).

More recently, a number of DNN-specific approaches and solvers, including Reluplex [13], ReluVal [14], Neurify [15], Planet [16], and Marabou [17], have been proposed and developed. These techniques scale to hundreds or a few thousand nodes. While a significant improvement, this is still several orders of magnitude fewer than the number of nodes present in many real-world applications. Scalability thus continues to be a challenge and the subject of active research.

Inspired by recent successes with parallelizing SAT solvers [18, 19], we propose a set of strategies and heuristics for leveraging parallelism to improve the scalability of neural network verification. The paper makes the following contributions: 1) We present a divide-and-conquer algorithm for neural network verification that is parameterized by different partition strategies and constraint solvers (Sec. III). 2) We describe two partitioning strategies for this algorithm (Sec. III-B): one that works by partitioning the input domain and a second one that performs case splitting based on the activation functions in the neural network. The first strategy was briefly mentioned in the Marabou tool paper [17]; we describe it in detail here. The second strategy is new. 3) We introduce the notion of *polarity* and use it to refine the partitioning (Sec. III-C); 4) We introduce a highly parallelizable pre-processing algorithm that significantly simplifies verification problems (Sec. III-D); 5) We show how polarity can additionally be used to speed up satisfiable queries (Sec. III-E); and 6) We implement the techniques in the Marabou framework and evaluate on existing and new neural network verification benchmarks from the aviation domain. We also perform an *ultra-scalability* experiment using cloud computing (Sec. IV). Our experiments show that the new and improved Marabou can outperform the previous version of Marabou as well as other state-of-the-art verification tools such as Neurify, especially on perception networks with a large number of inputs. We begin with preliminaries, review related work in Sec. V, and conclude in Sec. VI.

II. Preliminaries

In this section, we briefly review neural networks and their formalization, as well as the Reluplex algorithm for verification of neural networks.

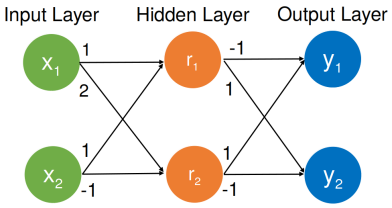


Fig. 1: A small feed-forward DNN $\mathcal{N}$.

A. Formalizing Neural Networks

Deep Neural Networks. A feed-forward *Deep Neural Network* (DNN) consists of a sequence of layers, including an input layer, an output layer, and one or more hidden layers in between. Each non-input layer comprises multiple *neurons*, whose values can be computed from the outputs of the preceding layer. Given an assignment of values to inputs, the output of the DNN can be computed by iteratively computing the values of neurons in each layer. Typically, a neuron's value is determined by computing an affine function of the outputs of the neurons in the previous layer and then applying a non-linear function, known as an *activation function*. A popular activation function is the Rectified Linear Unit (ReLU), defined as $ReLU(x) = max(0, x)$ (see [3, 20, 21]). In this paper, we focus on DNNs with ReLU activation functions; thus the output of each neuron is computed as $ReLU(w_1 \cdot v_1 + \ldots w_n \cdot v_n + b)$, where $v_1 \ldots v_n$ are the values of the previous layer's neurons, $w_1 \ldots w_n$ are the weight parameters, and b is a bias parameter associated with the neuron. A neuron is *active* or in the *active phase*, if its output is positive; otherwise, it is *inactive* or in the *inactive phase*.

Verification of Neural Networks. A neural network verification problem has two components: a neural network N, and a property P. P is often of the form $P_{in} \Rightarrow P_{out}$, where P_{in} is a formula over the inputs of N and P_{out} is a formula over the outputs of N. Typically, P_{in} defines an input region I, and P states that for each point in I, P_{out} holds for the output layer. Given a query like this, a verification tool tries to find a counter-example: an input point i in I, such that when applied to N, P_{out} is false over the resulting outputs. P holds only if such a counter-example does not exist.

The property to be verified may arise from the specific domain where the network is deployed. For instance, for networks that are used as controllers in an unmanned aircraft collision avoidance system (e.g., the ACAS Xu networks [13]), we would expect them to produce sensible advisories according to the location and the speed of the intruder planes in the vicinity. On the other hand, there are also properties that are generally desirable for a neural network. One such property is *local adversarial robustness* [22], which states that a small norm-bounded input perturbation should not cause major spikes in the network's output. More generally, a property may be an arbitrary formula over input values, output values, and values of hidden layers—such problems arise for example in the investigation of the neural networks' explainability [23], where one wants to check whether the activation of a certain ReLU r implies a certain output behavior (e.g., the neural network always predicts a certain class). The verification of neural networks with ReLU functions is decidable and NP-Complete [13]. As with many other verification problems, scalability is a key challenge.

VNN Formulas. We introduce the notion of VNN (Verification of Neural Network) formulas to formalize Neural Network verification queries. Let $\mathcal{X}$ be a set of variables. A *linear constraint* is of the form $\sum_{x_i \in \mathcal{X}} a_i x_i \bowtie b$, where a_i, b are rational constants, and $\bowtie \in \{\leq, \geq, =\}$. A *ReLU constraint* is of the form $\text{ReLU}(x_i) = x_j$, where $x_i, x_j \in \mathcal{X}$.

Definition 1. *A* VNN formula ϕ *is a conjunction of linear constraints and ReLU constraints.*

A feed-forward neural network can be encoded as a VNN formula as follows. Each ReLU r is represented by introducing a pair of input/output variables r_b, r_f and then adding a ReLU constraint $\text{ReLU}(r_b) = r_f$. We refer to r_b as the *backward-facing variable*, and it is used to connect r to the preceding layer. r_f is called the *forward-facing variable* and is used to connect r to the next layer. The weighted sums are encoded as linear constraints.

In general, a property could be any formula P over the variables used to represent $\mathcal{N}$. To check whether P holds on $\mathcal{N}$, we simply conjoin the representation of $\mathcal{N}$ with the negation of P and use a constraint solver to check for satisfiability. P holds iff the constraint is unsatisfiable.

Note that a solver for VNN formulas can solve a property P only if the negation of P is also a VNN formula. We assume this is the case in this paper, but more general properties can be handled by decomposing $\neg P$ into a disjunction of VNN formulas and checking each separately (or, equivalently, using a DPLL(T) approach [24]). This works as long as the atomic constraints are linear. Non-linear constraints (other than ReLU) are beyond the scope of this paper.

B. The Reluplex Procedure

The Reluplex procedure [13] is a sound, complete and terminating algorithm that decides the satisfiability of a VNN formula. The procedure extends the Simplex algorithm—a standard efficient decision procedure for conjunctions of linear constraints—to handle ReLU constraints. At a high level, the algorithm iteratively searches for an assignment that satisfies all the linear constraints, but treats the ReLU constraints lazily in the hope that many of them will be irrelevant for proving the property. There are two ways to fix a violated ReLU constraint r: 1) *repair the assignment* by updating the assignment to forward-facing r_f or backward-facing variable r_b to satisfy r, or 2) *case split* by considering separate cases for each phase of r (adding the appropriate constraint in each case) In both cases, the search continues using the Simplex algorithm, in the first with a soft correction via assignment update and in the second by adding hard constraints to the linear problem. Lazy handling of ReLUs is achieved by the threshold parameter t — the number of times a ReLU is repaired before the algorithm performs a case split. In [13], this parameter was set to 20,

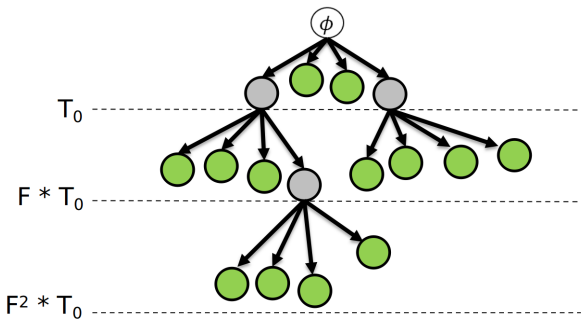


Fig. 2: An execution of the D&C algorithm.

but even more eager splitting is beneficial in some cases. The Reluplex algorithm also uses bound propagation to fix ReLUs to one phase whenever possible.

In this paper, we explore heuristic choices behind the two options to handle violated ReLU constraints. In the case of assignment repair, the question is which variable assignment, r_f or r_b, to modify (often both are possible). We refer to the strategy used to make this decision as the *direction heuristic*, and we discuss direction heuristics, especially in the context of parallel solving in Sec. III-E. For case splitting, the question is which ReLU constraint to choose. We refer to the strategy used for making this decision as the *branching heuristic*. We explore branching heuristics and their application to parallelizing the algorithm in Sec. III-B and Sec. III-C.

III. D&C: Parallelizing the Reluplex Procedure

In this section, we present a parallel algorithm called *Divide-and-Conquer* (or simply D&C) for solving VNN formulas, using the Reluplex procedure and an iterative-deepening strategy. We discuss two partitioning strategies: input interval splitting and ReLU case splitting.

Remark. A divide-and-conquer approach with an input-splitting strategy was described in the Marabou tool paper [17], albeit briefly and informally. We provide here a more general divide-and-conquer framework, which includes new techniques and heuristics, described in detail below.

A. The D&C algorithm

The D&C algorithm partitions an input problem into several sub-problems (that are ideally easier to solve) and tries to solve each sub-problem within a given time budget. If solving a problem exceeds the time budget, that problem is further partitioned and the resulting sub-problems are allocated an increased time budget. Fig. 2 shows solving of problem ϕ as a tree, where the root of the tree denotes the original problem. Sub-problems that exceed their allotted time budget are partitioned, becoming inner nodes, and leaves are sub-problems solved within their time budget. A formula ϕ is satisfiable if some leaf is satisfiable. If the partitioning is *exhaustive*, that is: $\phi \coloneqq \bigvee_{\phi_i \in \mathbf{partition}(\phi,n)} \phi_i$, for any $n > 1$, then ϕ is unsatisfiable iff all the leaves are unsatisfiable.

The pseudo-code of the D&C algorithm is shown in Algorithm 1, which can be seen as a framework parameterized by

Algorithm 1 Divide-and-Conquer

Input: query ϕ, initial partition size N_0, initial timeout T_0, partition size N, timeout factor F
Output: SAT/UNSAT
for $\psi \in \mathbf{partition}(\phi, N_0)$ **do**
  $Q.\text{enqueue}(\langle \psi, T_0 \rangle)$
while $Q.\text{notEmpty}()$ **do**
  $\langle \phi', t \rangle \leftarrow Q.\text{dequeue}()$
  $result \leftarrow \mathbf{solve}(\phi', t)$
  if $result =$ SAT **then**
    return SAT
  else if $result =$ TIMEOUT **then**
    for $\psi \in \mathbf{partition}(\phi', N)$ **do**
      $Q.\text{enqueue}(\langle \psi, t \cdot F \rangle)$
return UNSAT

the partitioning heuristic and the underlying solver. Details of these parameters are abstracted away within the **partition** and **solve** functions respectively and will be discussed in subsequent sections. The D&C algorithm takes as input the VNN formula ϕ and the following parameters: initial number of partitions N_0, initial timeout T_0, number of partitions N, and the timeout factor F. During solving, D&C maintains a queue Q of $\langle$query, timeout$\rangle$ pairs, which is initialized with the partition $N_0 \coloneqq \langle \phi, T_0 \rangle$. While the queue is not empty, the next pair $\langle \phi', t \rangle$ is retrieved from it, and the query ϕ' is solved with time budget t. If ϕ' is satisfiable, then the original query ϕ is satisfiable, and SAT is returned. If ϕ' times out, $\mathbf{partition}(\phi', N)$ creates N sub-problems of ϕ', each of which is enqueued with an increased time budget $t \cdot F$. If the sub-problem ϕ' is unsatisfiable, no special action needs to be taken. If Q becomes empty, the original query is unsatisfiable and the algorithm returns UNSAT. Note that the main loop of the algorithm naturally lends itself to parallelization, since the **solve** calls are mutually independent and query-timeout pairs can be asynchronously enqueued and dequeued.

We state without proof the following result, which is a well-known property of such algorithms.

Theorem 1. *The Divide-and-Conquer(ϕ, N_0, T_0, N, F) algorithm is sound and complete if the following holds: 1) the* **solve** *function is sound and complete; and 2) the* **partition** *function is exhaustive.*

In addition, with modest assumptions on **solve** and **partition**, and with $F > 1$, the algorithm can be shown to be terminating. In particular, it is terminating for the instantiations we consider below. The D&C algorithm can be tailored to the available computing resources (e.g., number of processors) by specifying the number of initial splits N_0. The other three search parameters of D&C specify the dynamic behavior of the algorithm, e.g. if T_0 and F are small, or if N is large, then new sub-queries are created frequently, which entails a more aggressive D&C strategy (and vice versa). Notice that we can completely discard the dynamic aspect of D&C by setting the initial timeout to be ∞.

A potential downside of the algorithm is that each call to **solve** that times out is essentially wasted time, overhead above

and beyond the useful work needed to solve the problem. Fortunately, as the following theorem shows, the number of wasted calls is bounded.

Theorem 2. *When Algorithm 1 runs on an unsatisfiable formula with $N \leq N_0$, the fraction of calls to* **solve** *that time out is less than $\frac{1}{N}$.*

Proof. Consider first the case when $N = N_0$. We can view D&C's UNSAT proof as constructing an N-ary tree, as shown in Fig. 2. The ℓ leaf nodes are calls to **solve** that do not time out. The t non-leaves are calls to **solve** that do time out. Since this is a tree, the total number of nodes n is one more than the number of edges. Since each query that times out has an edge to each of its N sub-queries, the number of edges is Nt. Thus we have $n = Nt + 1$ which can be rearranged to show the fraction of queries that time out: $\frac{t}{n} = \frac{1-1/n}{N} < \frac{1}{N}$. If $N < N_0$, then let $k = N_0 - N$. The number of nodes is then $n = Nt + k + 1$, and the result follows as before. □

B. Partitioning Strategies

A partitioning strategy specifies how to decompose a VNN formula to produce (hopefully easier) sub-problems.

A ReLU is *fixed* when the bounds on the backward-facing or forward-facing variable either imply that the ReLU is active or imply that the ReLU is inactive. Fixing as many ReLUs as possible reduces the complexity of the resulting problem.

With these concepts in mind, we present two strategies: 1) *input-based partitioning* creates case splits over the ranges of input variables, relying on bound propagation to fix ReLUs, whereas 2) *ReLU-based partitioning* creates case splits that fix the phase of ReLUs directly. Both strategies are exhaustive, ensuring soundness and completeness of the D&C algorithm (by Theorem 1). The *branching heuristic* which determines the choice of input variable, respectively ReLU, on which to split, can have a significant impact on performance. The branching heuristic should keep the total runtime of the sub-problems low as well as achieve a good *balance* between them. To illustrate, suppose the sub-problems created by splitting ReLU_1 take 10 and 300 seconds to solve, whereas those created by splitting ReLU_2 take 150 and 160 seconds to solve. Though the total solving time is the same, the more balanced split, on ReLU_2, results in shorter wall-clock time (given two parallel workers).

If most splits led to easier and balanced sub-formulas, then D&C would perform well, even without a carefully-designed branching heuristic. However, we have observed that this is not the case for many possible splits: the time taken to solve one (or both!) of the sub-problems generated by such splits is comparable to that required by the original formula (or even worse). Therefore, an effective branching heuristic is crucial. We describe two such heuristics below.

Input-based Partitioning. This simple partitioning strategy performs case splits over the range of an input variable. As an example, consider a VNN formula $\phi \coloneqq \phi' \wedge (-2 \leq x_1 \leq 1) \wedge (-2 \leq x_2 \leq 2)$, where x_1 and x_2 are the two input variables of a neural network encoded by ϕ'. Suppose we call $\mathbf{partition}(\phi, 2)$ using the input-splitting strategy. The choice is between splitting on the range of x_1 or the range of x_2. If we choose x_1, the result is two sub-formulas, ϕ_1 and ϕ_2, where: $\phi_1 \coloneqq \phi' \wedge (\mathbf{-2 \leq x_1 < -0.5}) \wedge (-2 \leq x_2 \leq 2)$ and $\phi_2 \coloneqq \phi' \wedge (\mathbf{-0.5 \leq x_1 \leq 1}) \wedge (-2 \leq x_2 \leq 2)$. An obvious heuristic is to choose the input with largest range. A more complex heuristic was introduced in [17]. It samples the network repeatedly, which requires considerable overhead. In fact, both of these heuristics perform reasonably well on benchmarks with only a few inputs (the ACAS Xu benchmarks, for example). Unfortunately, regardless of the heuristic used, this strategy suffers from the "curse of dimensionality" — with a large number of inputs it becomes increasingly difficult to fix ReLUs by splitting the range of only one input variable. Thus, the input-partitioning strategy does not scale well on such networks (e.g., perception networks), which often have hundreds or thousands of inputs.

ReLU-based Partitioning. A complementary strategy is to partition the search space by fixing ReLUs directly. Consider a VNN formula $\phi \coloneqq \phi' \wedge (\text{ReLU}(x) = y)$. A call to $\mathbf{partition}(\phi, 2)$ using the ReLU-based strategy results in two sub-formulas ϕ_1 and ϕ_2, where $\phi_1 \coloneqq \phi' \wedge (\text{ReLU}(x) = y) \wedge (\mathbf{x \leq 0}) \wedge (\mathbf{y = 0})$ and $\phi_2 \coloneqq \phi' \wedge (\text{ReLU}(x) = y) \wedge (\mathbf{x > 0}) \wedge (\mathbf{x = y})$. Note that here, ϕ_1 is capturing the inactive and ϕ_2 the active phase of the ReLU. Next, we consider a heuristic for choosing a ReLU to split on.

C. Polarity-based Branching Heuristics

We want to estimate the difficulty of sub-problems created by a partitioning strategy. One key related metric is the number of bounds that can be tightened as the result of a ReLU-split. As a light-weight proxy for this metric, we propose a metric called *polarity*.

Definition 2. *Given the ReLU constraint ReLU$(x) = y$, and the bounds $a \leq x \leq b$, where $a < 0$, and $b > 0$, the polarity of the ReLU is defined as: $p = \frac{a+b}{b-a}$.*

Polarity ranges from -1 to 1 and measures the symmetry of a ReLU's bounds with respect to zero. For example, if we split on a ReLU constraint with polarity close to 1, the bound on the forward-facing variable in the active case, $[0, b]$, will be much wider than in the inactive case, $[a, 0]$. Intuitively, forward bound tightening would therefore result in tighter bounds in the inactive case. This means the inactive case will probably be much easier than the active case, so the partition is unbalanced and therefore undesirable. On the other hand, a ReLU with a polarity close to 0 is more likely to have balanced sub-problems. We also observe that ReLUs in early hidden layers are more likely to produce bound tightening by forward bound propagation (as there are more ReLUs that depend on them).

We thus propose a heuristic that picks the ReLU whose polarity is closest to 0 among the first $k\%$ unfixed ReLUs, where k is a configurable parameter. Note that, in order to compute polarities, we need all input variables to be bounded, which is a reasonable assumption.

Algorithm 2 Iterative Propagation

Input: VNN query ϕ, timeout t
Output: preprocessed query ϕ'.
$progress \leftarrow \top$; $\phi' \leftarrow \phi$
while $progress = \top$ **do**
  $progress \leftarrow \bot$
  for r in $\mathbf{getUnfixedReLUs}(\phi')$ **do**
    $\psi \leftarrow \mathbf{polarityConstraint}(r)$
    $result = \mathbf{solve}(\phi' \wedge \psi, t)$
    if $result =$ UNSAT **then**
      $\psi' \leftarrow \mathbf{flipPhase}(\psi)$
      $\phi' \leftarrow \phi' \wedge \psi'$
      $progress \leftarrow \top$
return ϕ'

D. Fixing ReLU Constraints with Iterative Propagation

As discussed earlier, the performance of D&C depends heavily on the ability to split on ReLUs that result in balanced sub-formulas. However, sometimes a considerable portion of ReLUs in a given neural network cannot be split in this way. To eliminate such ReLUs we propose a preprocessing technique called *iterative propagation*, which aims to discover and fix ReLUs with unbalanced partitions.

Concretely, for each ReLU in the VNN formula, we temporarily fix the ReLU to one of its phases and then attempt to solve the problem with a short timeout. The goal is to detect unbalanced and (hopefully) easy unsatisfiable cases. Pseudocode is presented in Algorithm 2. The algorithm takes as input the formula ϕ and the timeout t, and, if successful, returns the equivalent formula ϕ' which has fewer unfixed ReLUs than ϕ. The outer loop computes the fixed point, while the inner loop iterates through the as-of-yet unfixed ReLUs. For each unfixed ReLU, the $\mathbf{polarityConstraint}$ function returns a constraint for the phase estimated to be easier using the polarity metric. If the solver returns UNSAT, then we can safely fix the ReLU to its other phase using the $\mathbf{flipPhase}$ function. We ignore the case where the solver returns SAT, since in practice this only occurs for formulas that are very easy in the first place.

Iterative propagation complements D&C, because the likelihood of finding balanced partitions is increased by fixing ReLUs that lead to unbalanced partitions. Moreover, iterative propagation is highly parallelizable, as each ReLU-fixing attempt can be solved independently. In Section IV, we report results using iterative propagation as a preprocessing step, though it is possible to integrate the two processes more closely, e.g., by performing iterative propagation after every $\mathbf{partition}$ call.

E. Speeding Up Satisfiable Checks with Polarity-Based Direction Heuristics

In this section, we discuss how the polarity metric introduced in Sec. III-C can be used to solve satisfiable instances quickly. When splitting on a ReLU, the Reluplex algorithm faces the same choice as the D&C algorithm. For unsatisfiable cases, the order in which ReLU case splits are done make little difference on average, but for satisfiable instances, it can be very beneficial if the algorithm is able to hone in on a satisfiable sub-problem. We refer to the strategy for picking which ReLU phase to split on first as the *direction heuristic*.

We propose using the polarity metric to guide the direction heuristic for D&C. If the polarity of a branching ReLU is positive, then we process the active phase first; if the polarity is negative, we do the reverse. Intuitively, formulas with wider bounds are more likely to be satisfiable, and the polarity direction heuristic prefers the phase corresponding to wider bounds for the ReLU's backward-facing variable.

Repairing an assignment when a ReLU is violated can also be guided by polarity (recall the description of the Reluplex procedure from Sec. II), as choosing between forward- or backward-facing variables amounts to choosing which ReLU phase to explore first. Therefore, we use this same direction heuristic to guide the choice of forward- or backward-facing variables when repairing the assignment. For example, suppose constraint $\text{ReLU}(x_b) = x_f$ is part of a VNN formula ϕ. Suppose the range of x_b is $[-2, 1]$, $\mathsf{A}(x_b) = -1$ and $\mathsf{A}(x_f) = 1$, where A is the current variable assignment computed by the Simplex algorithm. To repair this violated ReLU constraint, we can either assign 0 to x_f or assign 1 to x_b. In this case, the ReLU has negative polarity, meaning the negative phase is associated with wider input bounds, so our heuristic chooses to set $\mathsf{A}(x_f) = 0$.

We will see in our experimental results (Sec. IV) that these direction heuristics improve performance on satisfiable instances. Interestingly, they also have a positive performance impact on unsatisfiable instances.

IV. Experimental Evaluation

In this section, we discuss our implementation of the proposed techniques and evaluate its performance on a diverse set of real-world benchmarks – safety properties of control systems and robustness properties of perception models.

A. Implementation

We implemented the techniques discussed above in Marabou [17], which is an open-source neural network verification tool implementing the Reluplex algorithm. The tool also integrates the symbolic bound tightening techniques introduced in [14]. We refer to Marabou running the D&C algorithm as D&C-Marabou. Two partitioning strategies are supported: the original input-based partitioning strategy and our new ReLU-splitting strategy. All D&C configurations use the following parameters: the initial partition size N_0 is the number of available processors; the initial timeout T_0 is 10% of the network size in seconds; the number of online partitions N is 4; and the timeout factor F is 1.5. The k parameter for the direction heuristic (see Sec. III-C) is set to 5. The per-ReLU timeout for iterative propagation is 2 seconds. When the input dimension is low (≤ 10), symbolic bound tightening is turned on, and the threshold parameter t (see Sec. II) is reduced from 20 to 1. The parameters were chosen using a grid search on a small subset of benchmarks.

B. Benchmarks

The benchmark set consists of network-property pairs, with networks from three different application domains: aircraft collision avoidance (ACAS Xu), aircraft localization (TinyTaxiNet), and digit recognition (MNIST). Properties include robustness and domain-specific safety properties.

ACAS Xu. The ACAS Xu family of VNN benchmarks was introduced in [13] and uses prototype neural networks trained to represent an early version of the ACAS Xu decision logic [2]. The ACAS Xu benchmarks are composed of 45 fully-connected feed-forward neural networks, each with 6 hidden layers and 50 ReLU nodes per layer. The networks issue turning advisories to the controller of an unmanned aircraft to avoid near midair collisions. The network has 5 inputs (encoding the relation of the ownship to an intruder) and 5 outputs (denoting advisories: e.g., weak left, strong right). Proving that the network does not produce erroneous advisories is paramount for ensuring safe aviation operation. We consider four realistic properties expected of the 45 networks. These properties, numbered 1–4, are described in [13].

TinyTaxiNet. The TinyTaxiNet family contains perception networks used in vision-based *autonomous taxiing*: the task of predicting the position and orientation of an aircraft on the taxiway, so that a controller can accurately adjust the position of the aircraft [25]. The input to the network is a downsampled grey-scale image of the taxiway captured from a camera on the aircraft. The network produces two outputs: the lateral distance to the runway centerline, and the heading angle error with respect to the centerline. Proving that the networks accurately predict the location of the aircraft even when the camera image suffers from small noise is safety-critical. This property can be captured as local adversarial robustness. If the k^{th} output of the network is expected to be b_k for inputs near $\mathbf{a}$, we can check the unsatisfiability of the following VNN formula:

$$(y_k \geq b_k + \epsilon) \wedge \bigwedge_{i=1}^{N} (a_i - \delta \leq x_i \leq a_i + \delta),$$

where $\mathbf{x}$ denotes the actual network input, N the number of network inputs, and y_k the actual k^{th} output. The network is (δ, ϵ)-locally robust on $\mathbf{a}$, only if the formula is unsatisfiable. The training images are compressed to either 2048 or 128 pixels, with value range [0,1]. We evaluate the local adversarial robustness of two networks. TaxiNet1 has 2048 inputs, 1 convolutional layer, 2 feedforward layers, and 128 ReLUs. TaxiNet2 has 128 inputs, 5 convolutional layers, and a total of 176 ReLUs. For each network, we generate 100 local adversarial robustness queries concerning the first output (distance to the centerline). For each model, we sample 100 uniformly random images from the training data, and sample (δ, ϵ) pairs uniformly from the set $\{\langle 0.004, 3\rangle, \langle 0.004, 9\rangle, \langle 0.008, 3\rangle, \langle 0.008, 9\rangle, \langle 0.016, 9\rangle\}$. Setting $\delta = 0.004$ allows a 1 pixel-value perturbation in pixel brightness along each input dimension, and the units of ϵ are meters.

MNIST. In addition to the two neural network families with safety-critical real-world applications, we evaluate our techniques on three fully-connected feed-forward neural networks (MNIST1, MNIST2, MNIST3) trained on the MNIST dataset [26] to classify hand-written digits. Each network has 784 inputs (representing a grey-scale image) with value range [0,1], and 10 outputs (each representing a digit). MNIST1 has 10 hidden layers and 10 neurons per layer; MNIST2 has 10 hidden layers and 20 neurons per layer; MNIST3 has 20 hidden layers and 20 neurons per layer. We consider *targeted robustness* queries, which asks whether, for an input $\mathbf{x}$ and an incorrect output y', there exists a point in the ℓ^∞ δ-ball around x that is classified as y'. We sample 100 such queries for each network, by choosing random training images and random incorrect labels. We choose δ values evenly from $\{0.004, 0.008, 0.0016, 0.0032\}$.

C. Experimental Evaluation

We present the results of the following experiments: 1) Evaluation of each technique's effect on run-time performance of Marabou on the three benchmark sets. We also compare against Neurify, a state-of-the-art solver on the same benchmarks. 2) An analysis of trade-offs when running iterative propagation pre-processing. 3) Exploration of D&C scalability at a large scale, using cloud computing. More details about the experimental setup and results are shown in the Appendix.

1) Evaluation of the techniques on ACAS Xu, TinyTaxiNet, MNIST : We denote the ReLU-based partitioning strategy as **R**, polarity-based direction heuristics as **D**, and iterative propagation as **P**. We denote as **S** a hybrid strategy that uses input-based partitioning on ACAS Xu networks, and ReLU-based partitioning on perception networks. We run four combinations of our techniques: 1) **R**; 2) **S+D**; 3) **S+P**; 4) **S+D+P**, and compare them with two baseline configurations: 1) the sequential mode of Marabou (denoted as **M**); 2) D&C-Marabou with its default input-based partitioning strategy (denoted as **I**).

We compare with Neurify [15], a state-of-the-art solver, on the same benchmarks. Neurify derives over-approximations of the output bounds using techniques such as symbolic interval analysis and linear relaxation. On ACAS Xu benchmarks, it operates by iteratively partitioning the input region to reduce error in the over-approximated bounds (to prove UNSAT) and by randomly sampling points in the input region (to prove SAT). On other networks, Neurify uses off-the-shelf solvers to handle ReLU-nodes whose bounds are potentially overestimated. Neurify also leverages parallelism, as different input regions or linear programs can be checked in parallel.

We run all Marabou configurations and Neurify on a cluster equipped with Intel Xeon E5-2699 v4 CPUs running CentOS 7.7. 8 cores and 64GB RAM are allocated for each job, except for the **M** configuration, which uses 1 processor and 8GB RAM per job. Each job is given a 1-hour wall-clock timeout.

Results. Table I shows a breakdown of the number of solved instances and the run-time for all Marabou configurations and for Neurify. We group the results by SAT and UNSAT instances. For each row, we highlight the entries corresponding to the configuration that solves the most instances (ties broken by run-time). Here are some key observations:

TABLE I: Evaluation of the Techniques on ACAS Xu, TinyTaxiNet, MNIST

Bench.	**M**		**I**		**R**		**S**		**S+D**		**S+P**		**S+D+P**		**Neurify**	
[# inst.]	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time	#S	Time
ACAS	40	17224	**45**	**4884**	**45**	5009	**45**	**4884**	**45**	5480	**45**	8419	**45**	7244	39	4167
[180]	101	57398	130	48954	125	45036	130	48954	131	51413	130	50828	131	53717	**133**	1438
TinyTaxi.	34	4591	34	1815	34	433	34	433	34	419	34	533	**35**	1172	**35**	**88**
[200]	141	33909	110	24088	147	23079	147	23079	147	22345	**149**	**20583**	**149**	21949	146	7158
MNIST	11	2349	19	13032	22	9680	22	9680	26	11727	20	9956	**29**	19351	27	151
[300]	140	64418	78	27134	181	52776	181	52776	183	59195	184	67625	**185**	68307	153	10640
All	85	24164	98	19731	101	15122	101	14997	105	17626	99	18908	**109**	27767	101	4406
[680]	382	155725	318	100176	453	120891	458	124809	461	132953	463	139036	**465**	143973	432	19236

Number of solved instances (#S) and run-time in seconds of different configurations. For each benchmark set, top and bottom rows show data for satisfiable (SAT) and unsatisfiable (UNSAT) instances respectively. The results for configuration **S** are computed virtually from **R** and I.

– On ACAS Xu benchmarks, both input-based partitioning (**I**) and ReLU-based partitioning (**R**) yield performance gain compared with the sequential solver (**M**), with **I** being more effective. On perception networks, **I** solves significantly fewer instances than **M** while **R** continues to be effective.
– Comparing the performance of **S**, **S+D**, and **S+P** suggests that the polarity-based direction heuristics and iterative propagation each improve the overall performance of D&C-Marabou. Interestingly, the polarity-based heuristic improves the performance on not only SAT but also UNSAT instances, suggesting that by affecting how ReLU constraints are repaired, direction heuristics also favorably impact the order of ReLU-splitting. On the other hand, iterative propagation alone only improves performance on UNSAT instances. **S+D+P** solves the most instances among all the Marabou configurations, indicating that the direction heuristics and iterative propagation are complementary to each other.
– **S+D+P** solves significantly more instances than Neurify overall. While Neurify's strategy on Acas Xu benchmarks allows it to dedicate more time on proving UNSAT by rapidly partitioning the input region (thus yielding much shorter run-times than **S+D+P** on that benchmark set), its performance on SAT instances is subject to (un)lucky guesses. When it comes to perception neural networks that are deeper and have higher input dimensions, symbolic bound propagation, on which Neurify heavily relies, becomes more expensive and less effective. By contrast, Marabou does not rely solely on symbolic interval analysis, but in addition uses interval bound-tightening techniques (see [17] for details).

Fig. 3 shows a cactus plot of the 6 Marabou configurations and Neurify on all benchmarks. In this plot, we also include two virtual portfolio configurations: **Virt.-Marabou** takes the best run-time among all Marabou configurations for each benchmark, and **Virt.-All** includes Neurify in the portfolio. Interestingly, **S+D+P** is outperformed by **S+D** in the beginning but surpasses **S+D** after 500 seconds. This suggests that iterative propagation creates overhead for easy instances, but benefits the search in the long run. We also observe that Neurify can solve a subset of the benchmarks very rapidly, but solves very few benchmarks after 1500 seconds. One possible explanation is that Neurify splits the input region and makes solver calls eagerly. While this allows it to resolve some

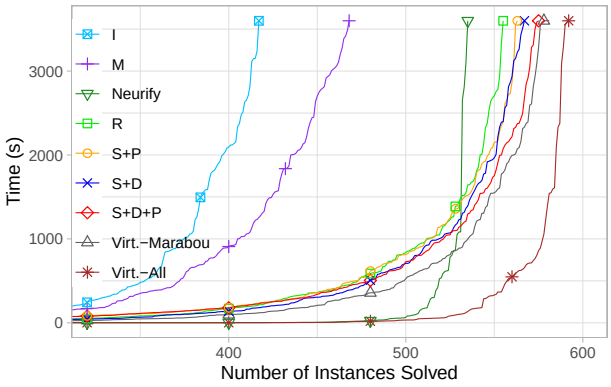


Fig. 3 Cactus plot: all solvers + two virtual best configurations.

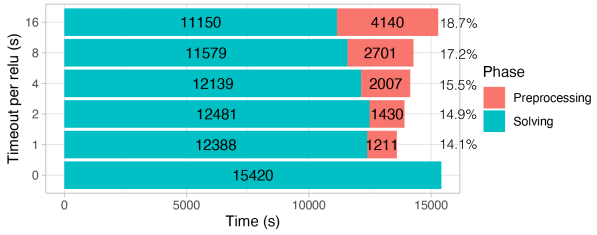


Fig. 4 The effect of varying per-ReLU timeout in preprocessing.

queries quickly, it also results in rapid (exponential) growth of the number of sub-regions and solver calls. By contrast, Marabou splits lazily. While it creates overhead sometimes, it results in more solved instances overall. The **Virt.-All** configuration solves significantly more instances than **Virt.-Marabou**, suggesting that the two procedures are complementary to each other. We note that the bound tightening techniques presented in Neurify can be potentially integrated into Marabou, and the polarity-based heuristics and iterative propagation could also be used to improve Neurify and other VNN tools.

2) Costs of Iterative Propagation: As mentioned in Sec. 2, intuitively, the longer the time budget during iterative propagation, the more ReLUs should get fixed. To investigate this trade-off between the number of fixed ReLUs and the overhead, we choose a smaller set of benchmarks (40 ACAS Xu benchmarks, 40 TinyTaxiNet benchmarks, and 40 MNIST benchmarks), and vary the timeout parameter t of iterative propagation. Each job is run with 32 cores, and a wall-clock timeout of 1 hour, on the same cluster as in Experiment IV-C1.

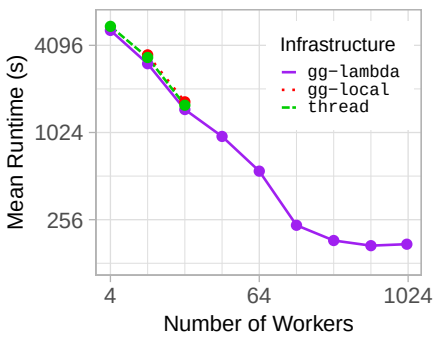


Fig. 5: Ultra-Scalability of D&C.

Results. Fig. 4 shows the preprocessing time + solving time of different configurations on commonly solved instances. The percentage next to each bar represents the average percentage of fixed ReLUs. Though the run-time and unfixed ReLUs continue to decrease as we invest more in iterative propagation, performing iterative propagation no longer provides performance gain when the per-ReLU-timeout exceeds 8 seconds.

3) Ultra-Scalability of D&C: D&C-Marabou runs on a single machine, which intrinsically limits its scalability to the number of hardware threads. To investigate how the D&C algorithm scales with much higher degrees of parallelism, we implemented it on top of `gg` [27], a platform for expressing parallelizable computations and executing them either *locally*, using different processes, or *remotely*, using cloud services. We call this implementation of D&C, *gg-Marabou*.

We measure the performance of D&C and gg-Marabou at varying levels of parallelism to establish that they perform similarly and to evaluate the scalability of the D&C algorithm. Our experiments use three underlying infrastructures: D&C-Marabou (denoted `thread`), gg-Marabou executed locally (`gg-local`), and gg-Marabou executed remotely on AWS Lambda [28] (`gg-lambda`). We vary the parallelism level, p, from 4 to 16 for the local infrastructures and from 4 to 1000 for `gg-lambda`. For `gg-lambda`, we run 3 tests per benchmark, taking the median time to mitigate variation from the network. From the UNSAT ACAS Xu benchmarks which D&C-Marabou can solve in under two hours using 4 cores, we chose 5 of the hardest instances. We set $T_0 = 5\,\text{s}$, $F = 1.5$, $N = 2^{\lfloor (5+\log_2 p)/3 \rfloor}$ and use the input-based partitioning strategy.

Results. Fig. 5 shows how mean runtime (across benchmarks) varies with parallelism level and infrastructure. Our first conclusion from Fig. 5 is that `gg` does **not** introduce significant overhead; at equal parallelism levels, all infrastructures perform similarly. Our second conclusion is that gg-Marabou scales well up to over a hundred workers. This is shown by the constant slope of the runtime/parallelism level line up to over a hundred workers. We note that the slope only flattens when total runtime is small: a few minutes.

V. Related Work

Over the past few years, a number of tools for verifying neural network have emerged and broadly fall into two categories — precise and abstraction-based methods. Precise approaches are complete and usually encode the problem as an SAT/SMT/MILP constraint [13, 17, 29, 30]. Abstraction-based methods are not necessarily complete and abstract the search space using intervals [14, 15] or more complex abstract domains [31]–[33]. However, most of these approaches are sequential, and for details, we refer the reader to the survey by Liu et al. [34]. To the best of our knowledge, only Marabou [17] and Neurify [15] (and its predecessor ReluVal [14]) leverage parallel computing to speed up verification.

As mentioned in Sec. IV, Neurify combines symbolic interval analysis with linear relaxation to compute tighter output bounds and uses off-the-shelf solvers to derive more precise bounds for ReLUs. These interval analysis techniques lend themselves well to parallelization, as independent linear programs can be created and checked in parallel. By contrast, D&C-Marabou creates partitions of the original query and solves them in parallel. Neurify supports a selection of hard-coded benchmarks and properties and often requires modifications to support new properties, while Marabou provides verification support for a wide range of properties.

Our work is inspired by the *Cube-and-Conquer* algorithm [18], which targets very hard SAT problems. Cube-and-Conquer is a divide-and-conquer technique that partitions a Boolean satisfiability problem into sub-problems by conjoining cubes —a cube is a conjunction of propositional literals— to the original problem and then employing a conflict-driven SAT solver [35] to solve each sub-problem in parallel. The propositional literals used in cubes are chosen using look-ahead [36] techniques. Our approach uses similar ideas, but in the VNN domain.

VI. Conclusions and Future Work

In this paper, we presented a set of techniques that leverage parallel computing to improve the scalability of neural network verification. We described an algorithm based on partitioning the verification problem in an iterative manner and explored two strategies that work by partitioning the input space or by splitting on ReLUs, respectively. We introduced a branching heuristic and a direction heuristic, both based on the notion of polarity. We also introduced a highly parallelizable preprocessing algorithm for simplifying neural network verification problems. Our experimental evaluation shows the benefit of these techniques on existing and new benchmarks. A preliminary experiment with ultra-scaling using the gg platform on Amazon Lambda also shows promising results.

Future work includes: i) Investigating more dynamic strategies for choosing hyper-parameters of the D&C framework. ii) Investigating different ways to interleave iterative propagation with D&C. iii) Investigating the scalability of ReLU-based partitioning to high levels of parallelism. iv) Improving the performance of the underlying solver, Marabou, by integrating conflict analysis (as in CDCL SAT solvers and SMT solvers) and more advanced bound propagation techniques such as those used by Neurify. v) Extending the techniques to handle

other piecewise-linear activation functions such as hard tanh and leaky ReLU, to which the notion of polarity applies.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019. [Online]. Available: https://doi.org/10.2514/1.G003724

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1106–1114.

[4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR (Poster)*, 2014.

[7] M. Cissé, Y. Adi, N. Neverova, and J. Keshet, "Houdini: Fooling deep structured prediction models," *CoRR*, vol. abs/1707.05373, 2017.

[8] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *ICLR (Workshop)*. OpenReview.net, 2017.

[9] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 243–257.

[10] ——, "Challenging SMT solvers to verify neural networks," *AI Commun.*, vol. 25, no. 2, pp. 117–135, 2012.

[11] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[12] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 171–177.

[13] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, 2017, pp. 97–117.

[14] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1599–1614.

[15] ——, "Efficient formal safety analysis of neural networks," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, 2018, pp. 6369–6379. [Online]. Available: http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks

[16] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2017, pp. 269–286.

[17] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *International Conference on Computer Aided Verification*, 2019, pp. 443–452.

[18] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 7261. Springer, 2011, pp. 50–65.

[19] M. J. H. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9710. Springer, 2016, pp. 228–245.

[20] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*. Omnipress, 2010, pp. 807–814.

[21] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1, 2013, p. 3.

[22] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *arXiv preprint arXiv:1709.02802*, 2017.

[23] D. Gopinath, H. Converse, C. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2019, pp. 797–809.

[24] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis– putnam–logemann–loveland procedure to dpll(*t*)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[25] K. D. Julian, R. Lee, and M. J. Kochenderfer, "Validation of image-based neural network controllers through adaptive stress testing," *arXiv preprint arXiv:2003.02381*, 2020.

[26] "The MNIST database of handwritten digits Home Page," http://yann.lecun.com/exdb/mnist/.

[27] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, 2019, pp. 475–488. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fouladi

[28] "AWS lambda," https://docs.aws.amazon.com/lambda/index.html.

[29] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," *CoRR*, vol. abs/1705.01320, 2017. [Online]. Available: http://arxiv.org/abs/1705.01320

[30] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[31] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[32] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 802–10 813.

[33] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[34] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," 2019.

[35] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 131–153.

[36] M. Heule and H. van Maaren, "Look-ahead based SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 155–184.

APPENDIX

A. More Details on gg-Marabou

The gg platform is a tool for expressing parallelizable computations and executing them. To use it, the programmer expresses their computation as *task graph*: a dependency graph of tasks, where each task is an executable program (e.g., a binary or shell script) that reads some input files and produces some output files. These output files can encode the result of the task, or an extension to the task graph that must be executed in order to produce that result. In our implementation of the D&C algorithm on top of gg, each task runs the base solver with a timeout. If the solver completes, the task returns the result, otherwise it returns a task graph extension encoding the division of the problem into sub-queries.

The local part of the gg experiment is run on a machine with 24 Xeon E5-2687W v4 CPUs, 132GB RAM, running Ubuntu 20.04.

B. More Details on Evaluation of Techniques

We present here a more detailed report of the runtime performance of different configurations and Neurify, as shown in Table II. We break down the ACAS Xu benchmark family by properties, and the other two benchmark sets by networks.

Fig. 6 shows the log-scaled pairwise comparisons between different configurations.

Bench.	**M**		**I**		**R**		**S+D**		**S+P**		**S+D+P**		**Neurify**	
[# inst.]	S	T	S	T	S	T	S	T	S	T	S	T	S	T
ACAS1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	17	32455	42	37125	37	33141	43	40936	42	36783	43	40107	**45**	**558**
ACAS2	34	17210	**39**	**4863**	**39**	4985	**39**	5456	**39**	8228	**39**	7074	33	4167
45	0	0	4	5461	4	5121	4	4042	4	4070	4	4156	**4**	**88**
ACAS3	**3**	9	**3**	10	**3**	12	**3**	13	**3**	91	**3**	83	**3**	**0**
45	**42**	18254	**42**	4900	**42**	4569	**42**	4826	**42**	6571	**42**	6295	**42**	**742**
ACAS4	**3**	5	**3**	11	**3**	12	**3**	11	**3**	100	**3**	87	**3**	**0**
45	**42**	6689	**42**	1468	**42**	2205	**42**	1609	**42**	3404	**42**	3159	**42**	**49**
ACAS	40	17224	**45**	**4884**	**45**	5009	**45**	5480	**45**	8419	**45**	7244	39	4167
180	101	57398	130	48954	125	45036	131	51413	130	50828	131	53717	**133**	**1438**
TinyTaxiNet1	**11**	1168	**11**	1579	**11**	370	**11**	356	**11**	337	**11**	357	**11**	**85**
100	84	27773	63	17883	89	14052	89	12521	89	14651	**89**	**14683**	81	7148
TinyTaxiNet2	23	3423	23	236	23	63	23	63	23	196	**24**	815	**24**	**3**
100	57	6136	47	6205	58	9027	58	9824	60	5932	60	7266	**65**	**10**
TinyTaxiNet	34	4591	34	1815	34	433	34	419	34	533	**35**	1172	**35**	**88**
200	141	33909	110	24088	147	23079	147	22345	149	20583	**149**	**21949**	146	7158
MNIST1	9	2178	11	3658	12	2190	12	1412	12	3682	**13**	**5715**	8	108
100	73	11880	47	12387	**80**	**12999**	80	15213	80	14090	80	13571	54	2285
MNIST2	2	171	5	3494	6	3246	7	3787	4	1782	9	9140	**13**	**6**
100	37	22069	17	5698	46	14576	46	14833	**48**	**19026**	47	15141	45	3247
MNIST3	0	0	3	5880	4	4244	**7**	6528	4	4492	**7**	**4496**	6	36
100	30	30469	14	9049	55	25201	57	29149	56	34509	**58**	**39595**	54	5108
MNIST	11	2349	19	13032	22	9680	26	11727	20	9956	**29**	**19351**	27	151
300	140	64418	78	27134	181	52776	183	59195	184	67625	**185**	**68307**	153	10640
All	85	24164	98	19731	101	15122	105	17626	99	18908	**109**	**27767**	101	4406
680	382	155725	318	100176	453	120891	461	132953	463	139036	**465**	**143973**	432	19236

TABLE II: Number of solved instances (S) and run time in seconds (T) of different configurations. For each family, top and bottom rows show data for satisfiable (SAT) and unsatisfiable (UNSAT) instances respectively.

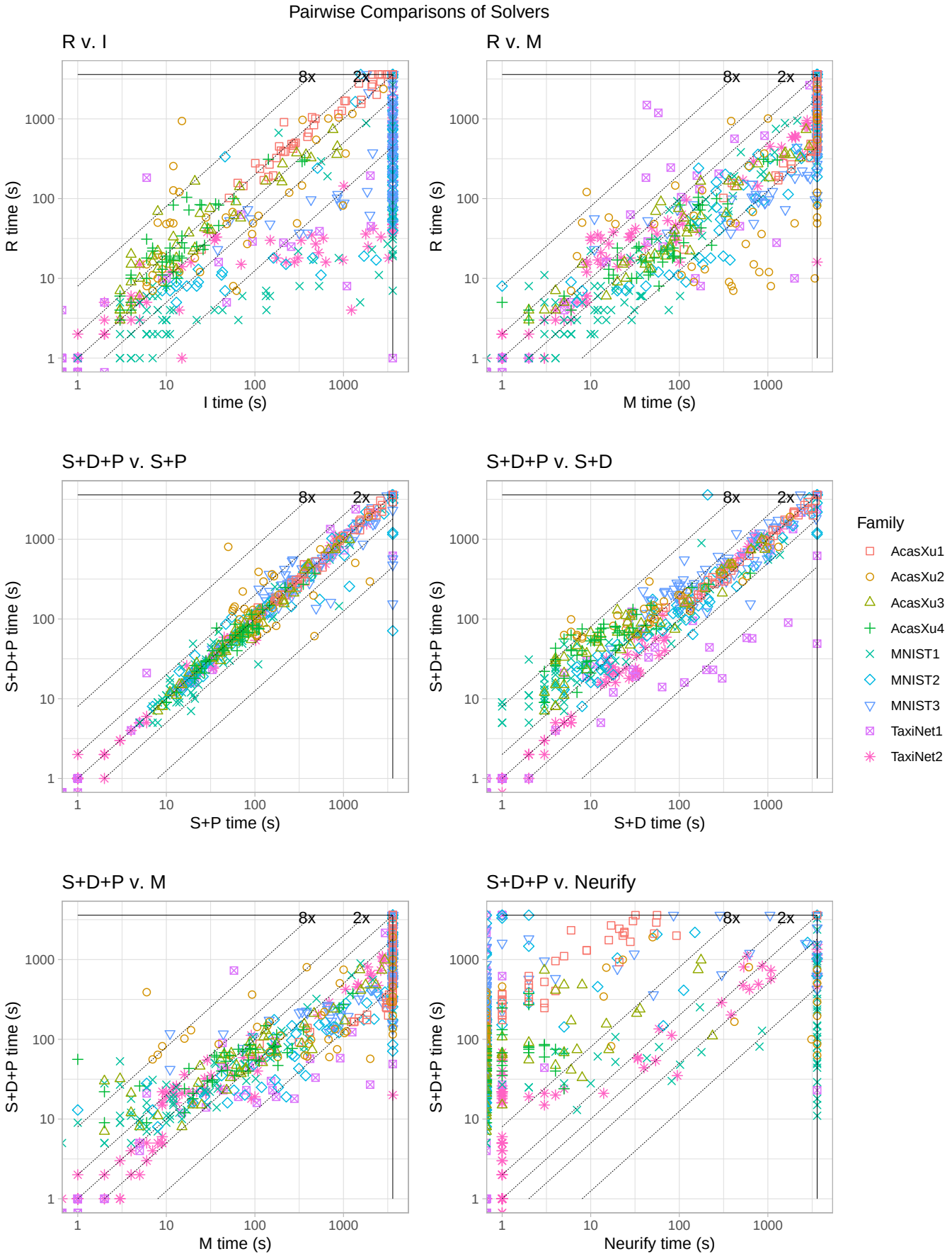


Fig. 6: Pairwise comparison between different configurations on all benchmarks.

Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis

Hongce Zhang[1⋆], Maxwell Shinn[2], Aarti Gupta[1], Arie Gurfinkel[3], Nham Le[3], and Nina Narodytska[4]

[1] Princeton University, USA {hongcez, aartig}@princeton.edu
[2] Yale University, USA maxwell.shinn@yale.edu
[3] University of Waterloo, Canada {arie.gurfinkel, nv3le}@uwaterloo.ca
[4] VMware Research, USA nnarodytska@vmware.com

Abstract. Recurrent Neural Networks (RNNs) are one of the most successful neural network architectures that deal with temporal sequences, e.g., speech and text recognition. Recently, RNNs have been shown to be useful in cognitive neuroscience as a model of decision-making. RNNs can be trained to solve the same behavioral tasks performed by humans and other animals in decision-making experiments, allowing for a direct comparison between networks and experimental subjects. Analysis of RNNs is expected to be simpler than the analysis of neural activity. However, in practice, reasoning about an RNN's behaviour is still a challenging problem. In this work, we take an approach based on formal verification for the analysis of RNNs. We make two main contributions. First, we consider the cognitive domain and formally define a set of useful properties to analyse for a popular experimental task. Second, we employ and adapt well-known verification techniques for reachability analysis to our focus domain, i.e., polytope propagation, invariant detection, and counterexample-guided abstraction refinement. Our experiments show that our techniques can effectively solve classes of benchmark problems that are challenging for state-of-the-art verification tools.

1 Introduction

Deep neural networks are among the most successful artificial intelligence technologies making impact in a variety of practical applications, including computer vision and natural language processing. Recently, recurrent neural networks (RNNs) have been employed in cognitive neuroscience to help understand decision-making in humans and other animals [19,22,33,25]. One way to understand neural networks is to formally analyse their properties. Indeed, formal verification of neural networks is a rapidly growing research area [15,41,29]. The main question that verification tackles is: given a network structure and a set of properties, check whether a neural network fulfills these properties. For example, properties may include whether an image is susceptible to an adversarial perturbation in a computer vision classification task [29], or whether a controller avoids unnecessary turning action for an aircraft control problem [15].

A lot of progress has been made in verifying neural networks over the last few years. The majority of work focuses on analysing *feed-forward* neural networks [15,16,44,27],

⋆ This work was mostly done during an internship at VMware Research.

while verification of *recurrent* neural networks has received significantly less attention [2,39]. This could be due to the fact that conceptually, verification of recurrent networks is no different from verification of feed-forward networks if we assume that the depth of unfolding is bounded. One can convert a recurrent network to a feed-forward network by *unfolding* the network's transition relation, i.e., by repeating it for a fixed number of steps [2]. Unfortunately, in practice, the depth of the unfolding can be large since we might need to repeat the transition relation more than a hundred times, leading to deep networks. Reasoning about such deep networks remains challenging.

In this work, we propose a novel approach to analyse RNNs. The main idea is to identify and exploit special properties of recurrent networks to aid efficient reasoning. Here we focus on an important class of recurrent neural networks which are trained to solve cognitive behavioral tasks [31], analogous to the tasks given to human and animal subjects in decision-making studies. Subjects and trained networks exhibit similar task performance as measured by response time and the probability of a correct response across difficulty levels [19]. Additionally, linear projections of RELU unit activations correspond qualitatively and quantitatively to neural activity as measured through electrophysiological recordings [19,22,33,25]. Cognitive tasks can be solved with recurrent networks which have a shallow transition relation, a deep unfolding depth, and, more importantly, only few modes of operations. The domain specific properties of these networks make them amenable to formal analysis. Our proposed approach employs three building blocks for the analysis of these networks: polytope propagation, invariant detection and counterexample-guided abstraction refinement (CEGAR) [6]. Polytope propagation is feasible (an exact propagation up to some depth and approximate afterward) because the transition relation of RNN is a shallow perception. Invariant detection is possible because we have only a few modes of operation and each mode spans over a prolonged time interval. Finally, CEGAR is effective because computed approximate reachability regions are often sufficient to prove a property, so only a few refinements are needed.

We make the following contributions. First, we analyse the cognitive domain and define properties of the network that are important to verify in this domain. Second, we propose new methods to verify properties of RNNs. Our approach consists of two phases. We adapt the "easy-to-verify networks" paradigm for training recurrent networks. Then, we perform verification using a hybrid polytope propagation, invariant detection and counterexample-guided refinement technique. Third, we perform a comparative analysis of our approach with several modern verification tools. We provide insights on why these networks are hard to reason about for existing tools, like SMT-based solvers. The main challenges are: how to handle an exponential number of polytopes during exact reachability analysis, and how/where to employ approximation while enabling enough precision to prove the required properties. Our experimental results demonstrate that our proposed techniques offer solutions in addressing these challenges.

In Section 2, we explain why and how RNN is used in cognitive domains. Section 3 explains reachability analysis and our work to facilitate such analysis on RNN, followed by experiments and discussion in Section 4.

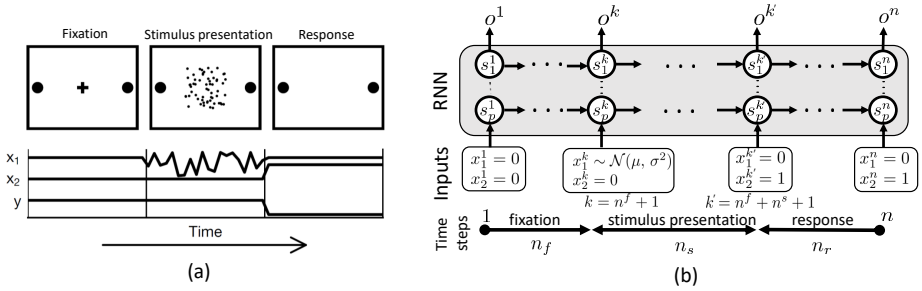


Fig. 1. (a) The random dot motion task [24,26] as shown to subjects (above), and as adapted for a recurrent neural network via two inputs x_1 and x_2 and one expected output y (below). (b) An unfolded RNN for n time steps. At each step the network consumes two inputs and emits an output. The state space is represented by state neurons s_j^k, $j \in [1, p]$, $k \in [1, n]$. Each mode of operation, fixation, stimulus and response, run for n_f, n_s and n_r time steps, respectively.

2 RNNs for cognitive domains

RNNs have become increasingly important in neuroscience.Trained RNNs exhibit behavioral similarities to human and animal subjects [31,43] in adapted versions of the same cognitive tasks, and also show patterns of RELU unit activations that correspond to signals from brain recordings [19,22,33,25]. Thus, understanding the mechanism of RNNs may have implications for linking brain activity to behavior. Here we focus on the random dot motion task in perceptual decision-making.

Cognitive task. The random dot motion task (Figure 1 (a)) [24,26] is a cognitive task given to humans and other animals in order to study decision-making in the presence of noisy stimuli [11]. In this task, dots on a video screen move in random directions, but with a mean direction either to the left or right. After a fixed duration stimulus presentation, subjects must identify and report this mean direction of motion. Task difficulty varies trial to trial through increased or decreased total strength of motion in either direction, known as motion coherence. Subjects interact with the task using eye movements, as captured by a high-speed camera and real-time eye-tracking software. This task consists of three phases:

- *fixation*: the subject initiates a trial by directing their gaze to the fixation cross at the center of the screen,
- *stimulus presentation*: a moving dots stimulus (sensory evidence) is presented to the subject for a fixed duration of time,
- *response*: the subject responds to the stimulus by directing their gaze to the left or right target, indicating the perceived mean direction of motion.

Successful performance on this task is thought to rely on integrating the noisy sensory evidence across time [20,21,24,11].

To adapt this task for a recurrent neural network (Figure 1(a)), the task elements are encapsulated within two input streams x_1 and x_2, and the expected output within a single stream y. The first input x_1 represents the noisy stimulus (sensory evidence) via a Gaussian random variable with fixed variance and non-zero mean, where positive

(negative) values indicate perceived rightward (leftward) direction of motion, and zero indicates the lack of the stimulus on the screen. The second input x_2 changes from 0 to 1 to indicate the beginning of the response period, which in this study was fixed to the end of stimulus presentation. The expected output of the network y represents the horizontal position of an eye movement, which is zero during the fixation and stimulus presentation periods and +1 or -1 for the response period, matching the mean direction of motion in a given trial.

Overview of the network. The RNN is trained to perform the random dot motion task. Figure 1(b) shows a schematic representation of the recurrent network. The network operates over n times steps. We split the interval $[1, n]$ into three sub-intervals spanning each of the three phases of the task: $\text{FX} \cup \text{SM} \cup \text{RS} = [1, n]$. The fixation phase spans over the interval $\text{FX} = [1, n_f]$, the stimulus presentation spans over the interval $\text{SM} = [n_f + 1, n_f + n_s]$ and the response spans over the interval $\text{RS} = [n_f + n_s + 1, n_f + n_s + n_r]$. At each step, the network consumes an input signal x^k and produces an output signal o^k, $k = 1, \ldots, n$.

Network specification. Each input x^k consists of two input values: x_1^k and x_2^k. The first input, x_1^k, corresponds to the stimulus signal. Note that it is constant over the fixation and response phases:

$$\text{if } k \in \text{SM} \text{ then } x_1^k \sim \mathcal{N}(\mu,\, \sigma^2) \text{ otherwise } x_1^k = 0 \tag{1}$$

During the stimulus presentation phase, x_1^k are samples from a Gaussian distribution with a task parameters μ and σ, where $\mu > 0$ represents the case when dots (noisy stimuli) move to the left, and $\mu < 0$ to the right. The second input, x_2^k, $k = 1, \ldots, n$, is an indicator input that signals whether the network is in the response phase or not.

$$\text{if } k \in \text{RS} \text{ then } x_2^k = 1 \text{ otherwise } x_2^k = 0 \tag{2}$$

At each step the network produced an output o^k, $k = 1, \ldots, n$. During training, our loss function encouraged the output to stay 0 in $\text{FX} \cup \text{SM}$ phases, and move toward $-1/1$ during the RS phase.

The transition relation of the network for the cognitive task has the following structure, $k = 1, \ldots, n$:

$$s^{k+1} := \quad F(s^k, x^k) = W_{rec}\text{RELU}(s^k) + W_{in}x^k + b_{rec} \tag{3}$$

$$o^k := \quad O(s^k) = W_{out}\text{RELU}(s^k) + b_{out}, \tag{4}$$

where $\text{RELU}(z) = \max(z, 0)$, W_{rec} and b_{rec} are the recurrent connectivity matrix and bias, respectively, W_{in} is an input connectivity matrix, W_{out} and b_{out} are output connectivity matrix and bias, respectively. We recall that the purpose of the network is to mimic the cognitive experiment. During an experiment, the subject makes a decision, e.g. left or right. Given a trained RNN $\mathcal{R}$, we define a decision function of the network, $C_{\mathcal{R}}(o)$, as follows:

$$C_{\mathcal{R}}(o) = \begin{cases} 1 & \text{if } o^k \geq 0.5, \forall k \in [n - r, n], \\ -1 & \text{if } o^k \leq -0.5, \forall k \in [n - r, n], \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

Given an input x, we say that the network chooses 1 if the last r outputs are above 0.5, where r is a parameter specified by a user. The network chooses -1 if the last r outputs are below -0.5. No decision is made otherwise.

Training the network. In order to train the network with a similar reward structure to human and animal subjects, we wanted the network to make a choice of ± 1 during the response period, even when the stimulus was ambiguous. This reflects the fact that subjects are only rewarded for correct responses, so a random response will be rewarded on 50% of trials but a failure to respond is never rewarded. Under a mean squared error loss, an optimal network would not make a choice if the stimulus was unclear. To encourage the network to guess, we designed a loss function such that a random choice of ± 1 has a lower expected value than a zero output. This was accomplished using a slope proportional to the square root of the error for outputs ranging from -1 to 1, and squared error outside of this region. More concretely, the loss function is defined as

$$\mathcal{L} = \sum_{t \in \text{FX} \cup \text{SM}} (y^t - o^t)^2 + \sum_{t \in \text{RS}} h(y^t, o^t)$$

$$h(y, o) = \begin{cases} 1 + (o \times \text{sign}(y) + 1)^2, & o \times \text{sign}(y) < -1 \\ (|o \times \text{sign}(y) - 1|/2)^{1/2}, & -1 \leq o \times \text{sign}(y) \leq 1 \\ (o \times \text{sign}(y) - 1)^2, & 1 < o \times \text{sign}(y) \end{cases}$$

Properties of the network. We highlight several properties relevant to verification of the above recurrent network. First, as the network encodes a simple behavioral pattern, the transition relation can be described with a small numbers of neurons. Second, the network dynamics mimic the three phases of the original experiment, so it has only three modes of operation. We can disregard the first phase during verification because all inputs are constants. Third, the input signal is well defined in the sense that points are sampled from a known distribution. This contrasts with computer vision tasks, where we cannot formally define all images of cars, for example. Finally, the depth of the recurrent network is large (110 time steps where the first 10 fixation steps can be pre-computed).

Properties to verify. We define a set of properties for networks that solve the random dot motion task.

Property 1 checks whether the network always makes the correct choice when all evidence falls above (below) a given threshold value of $p > 0$ ($p' < 0$), where p and p' are parameters of the property:

$$\min_{k \in \text{SM}} (x_1^k) > p \Rightarrow C_{\mathcal{R}}(o) = 1, \tag{6}$$

$$\max_{k \in \text{SM}} (x_1^k) < p' \Rightarrow C_{\mathcal{R}}(o) = -1. \tag{7}$$

In other words, if the stimulus is sufficiently strong, the network should output the correct response.

A weaker version of this property focuses on testing a hypothesis about the mechanism of this network. While classical theories of decision-making [20,21] suggest that subjects integrate (in the mathematical sense) sensory evidence over time, recent

approaches have questioned this perspective [32,40,45,34]. This weaker version tests whether there exists a stream of sensory evidence which indicates the network is not integrating all available information. It verifies whether the network always makes the correct choice given a sufficiently large mean strength of evidence:

$$\sum_{k \in \text{SM}} \frac{x_1^k}{n_f} > p \Rightarrow C_{\mathcal{R}}(o) = 1; \quad \sum_{k \in \text{SM}} \frac{x_1^k}{n_f} < p' \Rightarrow C_{\mathcal{R}}(o) = -1.$$

Property 2 checks whether an instantaneous large sample at the j-th point can trigger a choice when opposed by all remaining evidence.

$$(x_1^j < p') \wedge (\forall k \in \text{SM} \setminus \{j\}, x_1^k > p) \Rightarrow C_{\mathcal{R}}(o) = 1, \tag{8}$$

$$(x_1^j > p) \wedge (\forall k \in \text{SM} \setminus \{j\}, x_1^k < p') \Rightarrow C_{\mathcal{R}}(o) = -1. \tag{9}$$

If subjects do not integrate sensory evidence, alternative hypotheses imply that an instantaneous spike in evidence may trigger a choice [40,34,32], even if it opposes the mean direction of evidence. This property tests if evidence at a single point can trigger a choice despite consistent sensory evidence in the opposite direction.
In our experiments, we focus on verification of Property 1 and Property 2. We also define the following property for future exploration.
Property 3 checks whether there exists an input that leads to oscillating output state or a change in decision during the last steps:

$$\max_{k \in [n-r, n]} (\mid o^k - o^{k-1} \mid) > 1, \tag{10}$$

where r is a parameter specified by a user. Subjects experience changes of mind in the random-dot motion task [23]. These changes were not explicitly discouraged in the behavioral task, and likewise, are not explicitly penalized by objective function.

3 Overview of the proposed approach

Our approach consists of two phases. In the first phase we train an easier to verify network following ideas from [42]. While we had to adapt this approach to work for recurrent neural networks, we achieved a significant reduction of the network size without losing performance on the main cognitive task. We discuss our result of the first phase in Appendix B.

In the second phase, we perform property verification on RNN via reachability analysis. Suppose the property is of the form $C(s^0, x^0, x^1, ...) \implies P(o^n)$, where C is a predicate on initial state and inputs, and P is a predicate on the network output at step n. In reachability analysis, we start from $Reach^{\langle 0 \rangle} = s^0$ and compute the set of reachable states at step i, $Reach^{\langle i \rangle} : \{s^i \mid s^i = F(s^{i-1}, x^{i-1}), s^{i-1} \in Reach^{\langle i-1 \rangle}\}$ for each layer until reaching layer n, where F is as defined in (3). Then, we compute the output range based on $Reach^{\langle n \rangle}$ and check if it all satisfies P. To this end, we need to (a) have a representation of the set of reachable states, and (b) compute the reachable set for each layer till the final output.

We use a finite union of convex polytopes as the representation of the reachable set on a layer. We compute the reachable set layer by layer, which we call *propagating polytopes*. The computation for propagating polytopes is essentially applying piecewise affine transformation, and the details are given in Appendix C. As the number of polytopes usually increases along with the number of layers, we discuss two techniques here to keep the representation tractable.

3.1 The CEGAR Approach

Abstraction via over-approximation is a common approach to cope with an increasing number of polytopes. One abstraction is to group the polytopes that are close to each other and use their convex hull (computed from their vertices) as the abstraction of the reachable regions. Testing the distances between each pair of polytopes is expensive. Instead, we use a simple heuristic: we group the sub-polytopes that come from the same polytope in the previous layer. This heuristic is based on the fact that these sub-polytopes are connected, and thus would not be too far from each other. This grouping is an over-approximation — it can make unreachable states seem reachable, but not the other way around. In the implementation, we start this abstraction when the number of polytopes exceed a user-controlled threshold. We keep a record of the layer where we start to use abstraction. For each abstracted polytope in this layer, we keep a reference to the corresponding precise polytopes. Once abstraction is started in one layer, it is also applied in all subsequent layers.

Sometimes, our abstraction is too coarse to prove the desired properties. In such cases, we apply the counterexample-guided abstraction refinement (CEGAR) principle [6]. For a polytope P that intersects with the unsafe region where some property fails, we backtrace to a polytope $P_{precise}$ in the previous layer where we are about to apply abstraction. From $P_{precise}$, we again start propagation, first using the exact propagation method, until the threshold is reached again. This refinement may lead to proving the property or obtaining a counterexample that cannot be refined (completely inside the unsafe region). In either case, the CEGAR procedure on this polytope finishes. Otherwise, we again backtrace to refine the abstractions. The CEGAR loop is guaranteed to terminate as it makes at least one refinement per iteration, and there are only finitely many (though exponential) number of exact polytopes.

We note that our abstraction function (the convex hull of transformed polytopes) and our refinement function (replace the convex hull with union of convex polytopes) are an application of the standard finite-power-set domain from abstract interpretation on the convex polytopes [3].

3.2 Learning Invariants

Another technique to avoid an explosion in the number of polytopes is to find polytopes that represent a safe inductive invariant. For a given RNN with a fixed constraint on the inputs for each layer, we can define an *inductive invariant polytope* similar to a safe inductive invariant in a state transition system. We use $Q = F(P)$ to denote the image of P under transformation F. If Q is completely inside P, then P is an inductive invariant polytope. Unlike feed-forward networks, in RNN, P and Q are comparable

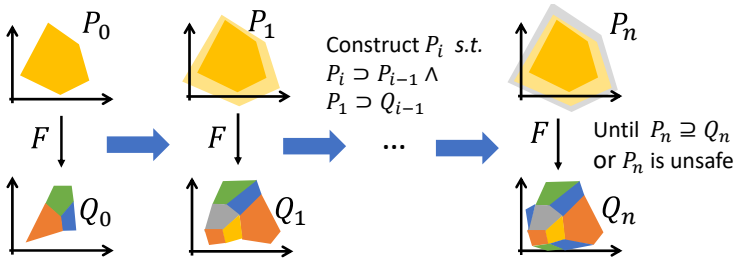


Fig. 2. Illustration of constructing safe inductive invariant polytopes.

as RNN uses the same hidden neurons for all iterations. Additionally, if such P is safe (does not fall out of the safe region), we can conclude P and all states reachable from P are safe. Thus, there is no need to propagate P further.

We construct an inductive invariant polytope from a given polytope as follows. Given polytope P_0, let Q_0 be its image (a union of convex polytopes). P_0 is an inductive invariant if Q_0 is contained in P_0. If this is not the case, let P_1 be the "join" of P_0 and Q_0 (therefore, by construction, $P_1 \supseteq Q_0$ and $P_1 \supseteq P_0$). If P_1 contains its image Q_1, then P_1 is inductive. If not, let P_2 be the "join" of P_1 and Q_1, and we continue to check on P_2. This process continues until either P_i becomes inductive or unsafe. If P_i becomes inductive, we successfully find a safe inductive invariant polytope P_i which contains the given polytope P_0. If P_i becomes unsafe, the construction fails as our relaxation creates too abstract a polytope. We will go with the exact image of P_0 (namely Q_0) and try in the next layer if we can construct a safe inductive invariant polytope from any polytope in Q_0. This procedure is illustrated in Figure 2. We elaborate the "join" operation used above in Appendix D.

4 Experimental Evaluation

Network specification. The network specification is as described by transitions (3)–(4), where $s^k \in \mathbb{R}^7, k \in [1, n]$. We unfold the network for 110 steps, so $n_f = 10$, $n_s = 50$ and $n_r = 50$. Inputs of the network are specified in Section 2. The parameters of the stimulus noise are as follows: μ is from a given set: $C = \cup\{-2^h/1000, 2^h/1000\}$, where $h \in \{0\} \cup [4, 10]$, and $\sigma = 0.3$. We use TensorFlow via PsychRNN [1] to train RNNs. For the choice function (5), we used $r = 10$. Verification is done on a machine with i5-8300H CPU and 32GB of memory.

Property specification. We consider Property 1 and Property 2. For Property 1, we limit the stimulus to be in a range (all positive or all negative) for a bounded input space, and test the strong version of Property 1 within that range. We constructed 30 ranges using coherence values C from the training data. Ranges are in the form $[c{\cdot}2^{-\delta\,\mathrm{sign}(c)}, c{\cdot}2^{\delta\,\mathrm{sign}(c)}]$ where $c \in \cup_{h\in\{0,5,7,9,10\}}\{-2^h/1000, 2^h/1000\}$, and $\delta \in \{0.2, 0.5, 0.7\}$. For Property 2, we enumerate a few positions of the pulses at the i-th input, ($i \in \{1, ..., 4\}$), while keeping the rest in a fixed range in the opposite direction (in similar ranges as Property 1).

We set 100 minutes as the timeout limit for each stimulus range. For simplicity, we name our methods as PP (polytope propagation with interval arithmetic), CEGAR (PP

plus counterexample-guided abstraction refinement) and Inv. (PP plus invariant construction). For CEGAR, the polytope bound to start approximation is set to be as small as 50 to favor a more abstract representation.

NNV framework. NNV is used for direct comparison to the polytope propagation methods we used. It is a MATLAB toolbox for neural network verification and performs reachability analysis using the star set representation [36]. Polytopes can be precisely captured by this representation. As the number of star sets increases, NNV can also over-approximate the reachable region on a layer using an interval hull. We refer to the exact and interval hull approximation methods as NNV-ex. and NNV-app., respectively.

Marabou. Marabou is an SMT-based tool that answers queries about network properties by solving constraint satisfaction problem [16]. Here it serves as a comparison using the SMT or MILP-based methods. For Marabou, the RNN is first unrolled to form a feed-forward network.

SPACER model checker. We have also experimented with SPACER [17], a state-of-the-art model checker included in Z3 [7]. SPACER has been successfully applied for verification of a variety of recurrent models in the domain of software verification, smart-contract analysis, and verification of control systems. Conceptually, SPACER is also based on polytope propagation. However, it propagates an *under-approximation* of bad states *backwards* from a property violation towards the initial condition. Throughout, it generalizes the polytopes based on symbolic reasoning on the transition relation. Unlike other techniques in this paper, SPACER is based on symbolic (as opposed to numeric) computation, using infinite precision arithmetic and symbolic quantifier elimination. While it is able to solve variants of Property 1, the running time is not competitive (over 20 hours). The main bottleneck is the blow up due to infinite precision arithmetic. It would be interesting to explore whether similar techniques or ideas can be lifted to the numerical setting.

The computation in polytope propagation is easily parallelizable to scale with the number of threads since the polytopes on the same layer are independent. Therefore, in our experiments, we focus on comparing single-thread performance and limit all methods to using a single thread.

Metrics. The number of solved instances, within the time and resource limits, is one of the metrics we can use to compare the performance. For solving time, we report the average time-to-termination. In the time-out cases, the time-limit is counted instead.

Table 1. Summary of Results (Time in Seconds)

Property 1								Property 2			
Regions		PP	CEGAR	Inv.	NNV-ex.	NNV-app.	Marabou	Regions		Inv.	NNV-ex.
Simple (19)	#. solved	**19**	**19**	**19**	16	19	18	All (48)	#. solved	23	**29**
	t_{mean}	**0.2**	**0.2**	**0.2**	947.6	21.8	1141.6		Ave. time (solved)	804.3	**29.58**
Positive (8)	#. solved	4	7	**8**	0	3	0		Ave. time (all)	3510.4	**2392.9**
	t_{mean}	3105.0	2216.6	1133.9	6000	**449.4**	6000		Uniquely solved	4	4
Negative (3)	#. solved	0	0	0	**3**	**3**	0				
	t_{mean}	6000	6000	6000	**0.5**	**0.5**	6000				
All (30)	#. solved	23	25	**27**	19	25	18				
	#. fastest	**16**	0	4	7	3	0				

Evaluation results. For Property 1, among the 30 stimulus ranges, 3 violate the property: $[m \cdot 2^{-\delta}, m \cdot 2^{\delta}]$, where $m = 2^0/1000$, $\delta \in \{0.2, 0.5, 0.7\}$. For the remaining cases, the property is checked to be valid. According to the difficulty and the sign of the checked stimulus ranges, we categorized them into three types:

- simple region (I): solvable by interval arithmetic.
- challenging regions: positive (II) and negative (III), both unsolvable by interval arithmetic.

A summary of the experimental results is listed in Table 1. In general, our techniques perform well on the simple regions and positive challenging regions, and NNV-ex. and NNV-app. work well on the negative challenging regions. In positive challenging regions, although NNV-app. on average takes the shortest time to terminate, its abstraction is too coarse to verify 5 of 8 ranges. The invariant method solves the most instances, although on average it does not rank the fastest on the instances it can solve. The hybrid polytope propagation method (PP) achieves the fastest time on most instances.

Property 2 is shown to be harder than Property 1 as the spike in the opposite direction usually adds a significant disturbance on the states and drives the reachable region. For this property, we only experimented with the Inv and NNV-ex. method. NNV-ex. is capable of solving 6 more ranges, with a relatively lower average solving time. However, the two have the same number of uniquely solved instances. Although not tested here, we expect the NNV-app. method will be able to solve more instances than NNV-ex. within the time-limit.

Discussion and Lessons Learned. For Property 1, the performance difference on the positive and negative categories leads us to a further investigation on the underlying causes. It turned out that the challenges in the positive regions are mainly the explosion on the number of polytopes. Among them, the "invariant" method solves the most regions in the time limit, as it saves the efforts of propagating safe polytopes. The challenges in the negative regions are mainly due to an increasing number of facets. The $\mathcal{H}$- and $\mathcal{V}$-polytope representations are known to have issues in scaling with an exponential increase in number of facets; this is also the reason that our methods fail in the negative regions (in the experiments, the number of facets quickly increased to 10^5 within the first 15 timesteps). On the other hand, the star set representation is able to handle better a large number of facets, since it uses a higher dimension space for coefficients and does not keep the representation of facets in the original space. This trade-off delays the blow-up. However, it cannot substitute for polytope representation. For containment check (required by the invariant method), one still needs to convert the star-set representation to projected $\mathcal{H}$- and $\mathcal{V}$-polytopes. And as the previous work noted, obtaining the convex hull (required by the CEGAR method) on the star set representation becomes more expensive [42].

For Property 2, in the case of having a spike in evidence in the opposite direction, there is no clear separation between where the number of polytope dominates vs. where the number of facets dominates. The two challenges are now mixed up. To successfully verify this property, the analysis method must be able to handle both the exponential increase of the number of polytopes and the exponential increase of the number of facets. Exploring solutions to both challenges is left for future work.

References

1. PsychRNN. https://github.com/dbehrlich/PsychRNN (last visited: Feb 18, 2020)
2. Akintunde, M.E., Kevorchian, A., Lomuscio, A., Pirovano, E.: Verification of rnn-based neural agent-environment systems. In: AAAI, 2019. pp. 6006–6013. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33016006, https://doi.org/10.1609/aaai.v33i01.33016006
3. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: SAS 2012. pp. 300–316 (2012). https://doi.org/10.1007/978-3-642-33125-1_21, https://doi.org/10.1007/978-3-642-33125-1_21
4. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. In: VMCAI 2004. pp. 135–148 (2004). https://doi.org/10.1007/978-3-540-24622-0_13, https://doi.org/10.1007/978-3-540-24622-0_13
5. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming **72**(1-2), 3–21 (2008)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
7. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)
8. Fischetti, M., Jo, J.: Deep neural networks and mixed integer linear optimization. Constraints **23**(3), 296–309 (2018)
9. Fukuda, K.: An efficient implementation of the double description method. https://github.com/cddlib/cddlib (2018)
10. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy, SP 2018. pp. 3–18. IEEE Computer Society (2018). https://doi.org/10.1109/SP.2018.00058, https://doi.org/10.1109/SP.2018.00058
11. Gold, J.I., Shadlen, M.N.: The neural basis of decision making. Annual Review of Neuroscience **30**(1), 535–574 (jul 2007). https://doi.org/10.1146/annurev.neuro.29.051605.113038
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. The MIT Press (2016)
13. Gopinath, D., Taly, A., Converse, H., Pasareanu, C.S.: Finding invariants in deep neural networks. CoRR **abs/1904.13215** (2019), http://arxiv.org/abs/1904.13215
14. Grünbaum, B., Kaibel, V., Klee, V., Ziegler, G.: Convex Polytopes. Graduate Texts in Mathematics, Springer (2003), https://books.google.com/books?id=5iV75P9gIUgC
15. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: CAV. pp. 97–117 (2017)
16. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.W.: The Marabou framework for verification and analysis of deep neural networks. In: CAV. pp. 443–452 (2019)
17. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. In: CAV 2014. pp. 17–34 (2014). https://doi.org/10.1007/978-3-319-08867-9_2, https://doi.org/10.1007/978-3-319-08867-9_2
18. Ma, S., Liu, Y., Tao, G., Lee, W., Zhang, X.: NIC: detecting adversarial samples with neural network invariant checking. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019. The Internet Society (2019), https://www.ndss-symposium.org/ndss-paper/nic-detecting-adversarial-samples-with-neural-network-invariant-checking/
19. Mante, V., Sussillo, D., Shenoy, K.V., Newsome, W.T.: Context-dependent computation by recurrent dynamics in prefrontal cortex. Nature **503**(7474), 78–84 (nov 2013). https://doi.org/10.1038/nature12742

20. Ratcliff, R.: A theory of memory retrieval. Psychological Review **85**(2), 59–108 (1978). https://doi.org/10.1037/0033-295x.85.2.59
21. Ratcliff, R., Smith, P.L., Brown, S.D., McKoon, G.: Diffusion decision model: Current issues and history. Trends in Cognitive Sciences **20**(4), 260–281 (apr 2016). https://doi.org/10.1016/j.tics.2016.01.007
22. Remington, E.D., Narain, D., Hosseini, E.A., Jazayeri, M.: Flexible sensorimotor computations through rapid reconfiguration of cortical dynamics. Neuron **98**(5), 1005–1019.e5 (jun 2018). https://doi.org/10.1016/j.neuron.2018.05.020
23. Resulaj, A., Kiani, R., Wolpert, D.M., Shadlen, M.N.: Changes of mind in decision-making. Nature **461**(7261), 263–266 (aug 2009). https://doi.org/10.1038/nature08275
24. Roitman, J.D., Shadlen, M.N.: Response of neurons in the lateral intraparietal area during a combined visual discrimination reaction time task. The Journal of neuroscience : the official journal of the Society for Neuroscience **22**, 9475–9489 (Nov 2002)
25. Russo, A.A., Bittner, S.R., Perkins, S.M., Seely, J.S., London, B.M., Lara, A.H., Miri, A., Marshall, N.J., Kohn, A., Jessell, T.M., Abbott, L.F., Cunningham, J.P., Churchland, M.M.: Motor cortex embeds muscle-like commands in an untangled population response. Neuron **97**(4), 953–966.e8 (feb 2018). https://doi.org/10.1016/j.neuron.2018.01.004
26. Salzman, C.D., Britten, K.H., Newsome, W.T.: Cortical microstimulation influences perceptual judgements of motion direction. Nature **346**(6280), 174–177 (jul 1990). https://doi.org/10.1038/346174a0
27. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: NIPS. pp. 10825–10836 (2018), http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification
28. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 41:1–41:30 (Jan 2019). https://doi.org/10.1145/3290354, http://doi.acm.org/10.1145/3290354
29. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: Boosting robustness certification of neural networks. In: ICLR 2019 (2019), https://openreview.net/forum?id=HJgeEh09KQ
30. Singh, G., Püschel, M., Vechev, M.: Fast polyhedra abstract domain. In: ACM SIGPLAN Notices. vol. 52, pp. 46–59. ACM (2017)
31. Song, H.F., Yang, G.R., Wang, X.J.: Training excitatory-inhibitory recurrent neural networks for cognitive tasks: A simple and flexible framework. PLOS Computational Biology **12**(2) (feb 2016). https://doi.org/10.1371/journal.pcbi.1004792
32. Stine, G.M., Zylberberg, A., Ditterich, J., Shadlen, M.N.: Differentiating between integration and non-integration strategies in perceptual decision making (January 2020). https://doi.org/10.1101/2020.01.24.918169
33. Sussillo, D., Churchland, M.M., Kaufman, M.T., Shenoy, K.V.: A neural network that finds a naturalistic solution for the production of muscle activity. Nature Neuroscience **18**(7), 1025–1033 (jun 2015). https://doi.org/10.1038/nn.4042
34. Thura, D., Beauregard-Racine, J., Fradet, C.W., Cisek, P.: Decision making by urgency gating: theory and experimental support. Journal of Neurophysiology **108**(11), 2912–2930 (dec 2012). https://doi.org/10.1152/jn.01071.2011
35. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: ICLR (2019)
36. Tran, H.D., Cai, F., Diego, M.L., Musau, P., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. ACM Trans. Embed. Comput. Syst. **18**(5s), 105:1–105:22 (Oct 2019). https://doi.org/10.1145/3358230, http://doi.acm.org/10.1145/3358230
37. Tran, H., Lopez, D.M., Musau, P., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Star-based reachability analysis of deep neural networks. In: FM2019. pp. 670–686.

LNCS, Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_39, https://doi.org/10.1007/978-3-030-30942-8_39
38. Vengertsev, D., Sherman, E.: Recurrent neural network properties and their verification with monte carlo techniques (2020)
39. Wang, Q., Zhang, K., Liu, X., Giles, C.L.: Verification of recurrent neural networks through rule extraction. CoRR **abs/1811.06029** (2018), http://arxiv.org/abs/1811.06029
40. Waskom, M.L., Kiani, R.: Decision making through integration of sensory evidence at prolonged timescales. Current Biology **28**(23), 3850–3856.e9 (dec 2018). https://doi.org/10.1016/j.cub.2018.10.021
41. Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: ICML. pp. 5273–5282 (2018)
42. Xiao, K.Y., Tjeng, V., Shafiullah, N.M.M., Madry, A.: Training for faster adversarial robustness verification via inducing relu stability. In: ICLR 2019. OpenReview.net (2019), https://openreview.net/forum?id=BJfIVjAcKm
43. Yang, G.R., Joglekar, M.R., Song, H.F., Newsome, W.T., Wang, X.J.: Task representations in neural networks trained to perform many cognitive tasks. Nature neuroscience **22**(2), 297 (2019)
44. Zhang, H., Weng, T., Chen, P., Hsieh, C., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: NIPS. pp. 4944–4953 (2018)
45. Zoltowski, D.M., Latimer, K.W., Yates, J.L., Huk, A.C., Pillow, J.W.: Discrete stepping and nonlinear ramping dynamics underlie spiking responses of LIP neurons during decision-making. Neuron **102**(6), 1249–1258.e10 (jun 2019). https://doi.org/10.1016/j.neuron.2019.04.031

Appendix

A Background

Feed-forward neural networks. A *feed-forward neural network* (FNN) is a function that takes an input $x \in \mathbb{R}^d$, applies a sequence of transformations and produces an output vector o, $o \in \mathbb{R}^p$. Transformations are usually grouped in blocks called layers. Each layer is a composition of a linear and a non-linear transformation. Each layer produces an intermediate representation $a^{\langle k \rangle} \in \mathbb{R}^{p_k}$, $k \in [1, m]$. The first layer takes the input x and produces an intermediate state $a^{\langle 1 \rangle}$ that is passed to the next layer. The final layer takes $a^{\langle m-1 \rangle}$ as an input and produces an output o. A network can be described as a composition of functions, assuming $a^{\langle 1 \rangle} = x$, as follows:

$$a^{\langle k+1 \rangle} = f_j(W^{\langle k \rangle} a^{\langle k \rangle} + b^{\langle k \rangle}), k \in [1, m-1] \tag{11}$$

$$o = f_m(W^{\langle m \rangle} a^{\langle m \rangle} + b^{\langle m \rangle}), \tag{12}$$

where f_j is a non-linear function, $W^{\langle k \rangle}$ is a linear transformation matrix and $b^{\langle k \rangle}$ is a bias vector at the kth layer, $k \in [1, m]$.

Recurrent neural networks. A *Recurrent Neural Network* (RNN) is a neural network that operates over a sequence of inputs [12]. At each time-step k, a network consumes an input x^k and its hidden state s^k, produces the next hidden state s^{k+1} and an output o^k. A simple version of recurrent network can be described using the following transition function:

$$s^{k+1} = f(W_{rec} s^k + W_{in} x^k + b_{rec}), \tag{13}$$

$$o^k = g(W_{out} s^k + b_{out}), \tag{14}$$

where f and g are non-linear functions, $W_{rec}, W_{in}, W_{out}, b_{rec}$ and b_{out} are parameters to learn. In this work, f and g are RELU operators. We define the exact structure of the network in Section 2. If we assume that the length of input is bounded by n, we can transform an RNN to a feed-forward network by unrolling the transition relation of an RNN for n time steps, e.g. [12]. There are two main structural differences between feed-forward networks and recurrent neural networks that are relevant from the verification standpoint. The first difference is the depth of the networks. As RNNs operate over long input sequences, unrolling the transition relation leads to deep networks with a large number of layers. Therefore, the resulting unfolded network is very challenging to reason about for both complete and incomplete methods. The second difference is that feed-forward networks apply different transformations on each layer, whereas unrolled recurrent networks use the same transformation.

Polytope and its representation. We recall the definition of a closed convex polytope (or polytope for short) and its two representations: $\mathcal{V}$-polytope and $\mathcal{H}$-polytope [14]. *A $\mathcal{V}$-polytope* is a convex hull of a finite set $Y = \{y^1, \ldots, y^n\}$ of points in $\mathbb{R}^d$:

$$\mathcal{V}\text{-}\mathcal{P} = conv(Y) := \{\sum_{i=1}^{n} \lambda_i y^i \mid \lambda_1, \ldots, \lambda_n \geq 0, \sum_{i=1}^{n} \lambda_i = 1\}$$

An $\mathcal{H}$-polytope is the solution of a finite system of linear inequalities:

$$\mathcal{H}\text{-}\mathcal{P} = \mathcal{H}\text{-}\mathcal{P}(A, b) := \{y \in \mathbb{R}^d \mid a_i^T y \leq b_i, i \in [1, m]\}$$

assuming that the set of solutions is bounded, where $A \in \mathbb{R}^{m \times d}$ is a real matrix with rows a_i^T and $b \in \mathbb{R}^m$ is a real vector with entries b_i.

A polytope is a convex closed subset P of $\mathbb{R}^d$ that can be represented as a $\mathcal{V}$-polytope or an $\mathcal{H}$-polytope. We use both representations in our algorithms. There are existing libraries, for example CDD [9] and PPL [5], which provide the functionality for the conversion between the two representations.

Mixed Integer Linear Programming (MILP). We briefly overview MILP technology as we use it for the polytopes propagation. MILP solves linear problems over a set of integer and real valued variables. MILP contains a set of decision variables, a set of linear constraints over these variables and an objective function to be optimized (minimized or maximized) that is linear in decision variables. Without loss of generality we consider a minimization formulation of a MILP. Let $x_1, \ldots, x_n$ be a set of decision variables, a mixed integer linear program can be written as

$$\min \sum_i c_i x_i \tag{15}$$

$$\text{subject to } \sum_i a_{ij} x_i \geq b_j, j \in [1, m] \tag{16}$$

$$x_i \in \mathbb{Z}, i \in \mathcal{I}_1,$$

$$x_i \in \mathbb{R}, i \in \mathcal{I}_2,$$

where $\mathcal{I}_1$ is a set of indices of integer variables and $\mathcal{I}_2$ is a set of indices of real variables, $\mathcal{I}_1 \cup \mathcal{I}_2 = [1, n]$.

Interval Arithmetic or Bound Propagation. Interval arithmetic computes the upper and lower bounds for a layer based on the bounds of the previous layer. It is a fast but relatively conservative bound estimation. Recently, Xiao et al. [42] proposed an improvement: to estimate the bound of a layer, it backtracks as much as possible rather than directly using the bounds of the previous layer. This gives a tighter bound without incurring much computational overhead.

B Train an easier-to-verify RNN

In the first phase of our approach, we train a recurrent network that is simpler to verify for decision procedures, like MILP or SMT solvers [42]. There are two main bottlenecks for decision procedures to reason about neural networks. The first issue is that there is a large number of variables in linear constraints, like a typical linear constraint (15) used in a MILP formulation of a network [8]. For example, if we have a fully-connected layer, the number of variables in (15) is equal to the number of neurons in the previous layer. The second issue is the presence of $\textsc{Relu} = \max(x, 0)$ operators in a MILP or SMT formulation as these are piecewise linear functions. Each piecewise linear transformation introduces a binary branching factor for the solver. Note that the number of $\max$ operations is equal to the number of neurons in the network. In [42], Xiao *et al.* initiated re-

search on training easier to verify networks and introduced two techniques, weight sparsification and RELUs stabilization, which are very effective for feed-forward networks.

The main idea of RELUs stabilization is to train a network in such a way that piece-wise linear functions, RELUs, are linearized. We say that a neuron s_i^k is *stable* if for all possible inputs of the network $sign(lb(s_i^k)) = sign(ub(s_i^k))$, where $lb(s_i^k)$ is the smallest value that a neuron can take and $ub(s_i^k)$ is the largest value that a neuron can take for all possible inputs. We encourage stabilization of RELUs during the training by using bounds propagation techniques and adding an extra term to the loss function. Note that if lower and upper bounds of s_i^k have the same sign then we know that RELU degenerates to a linear function from a piece-wise linear function. We applied stabilization of neurons in our training procedure. Interestingly, we found that there are state neurons s_j^k such that $ub(s_j^k) < 0$, $k \in [1, n]$. In other words, there is s_j^k that is stable for all k. This means that $\text{RELU}(s_j^k) = 0$ at all time steps signaling that a state space can be reduced. Hence, we reduce the number of state neurons and retrained the smaller network from scratch. Surprisingly, our re-training procedure was extremely effective in reducing the network size. For example, we have successfully reduced the size of the state vector from 50 to 7.

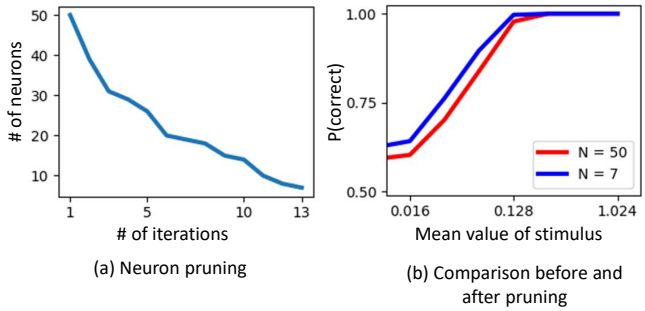


Fig. 3. Reducing RNN for easier verification.

We also experimented with weight sparsification from [42]. This idea consists of (a) using the L1 regularization during the training to encourage small weights and (b) setting small weights, i.e. weights that are smaller than a given threshold, to zero as a post-processing step. However, we noted that the quality of the network is very sensitive to weights sparsification. We were *not* able to achieve significant reductions in the number of non-zero parameters compare to results reported in [42]. However, it is not surprising. Note that if we zero one element of a transformation matrix in (3)–(4), we zero this weight at each timestamp as the same matrix is used at each step. Hence, an RNN's structure seems to be less favorable to weights sparsification compared to feed-forward networks.

C Polytope propagation

The core of reachability analysis is to efficiently propagate the reachable set throughout the network. Our general method of propagating a polytope over one layer is presented as the function PROPAGATEPOLYTOPEONELAYER in Algorithm 1. It takes a polytope in

Algorithm 1: PROPAGATEPOLYTOPEONELAYER(W_{rec}, W_{in}, b_{rec}, H, I): Propagate one polytope for one layer.

Input: $W_{rec}, W_{in}, b_{rec},$: weights and bias of a layer, H: $\mathcal{H}$-representation of the polytope to propagate, I: input constraints.
Output: P_{out}: set of polytopes.

$P_{out} \leftarrow \varnothing$;
$F \leftarrow$ ENCODEMILP(W_{rec}, W_{in}, b_{rec});
$C \leftarrow H \wedge I \wedge F$;
$A \leftarrow$ GETFEASIBLEASSIGNMENTS(C);
for each $a \in A$ **do**
    $RS \leftarrow$ RELUSTATUSCONSTR(a);
    $P_{sub} \leftarrow H \wedge I \wedge RS$;
    $V_{sub} \leftarrow$ GETVERTICES(P_{sub});
    $T \leftarrow$ GETTRANSFORMMATRIX($W_{rec}, W_{in}, b_{rec}, a$);
    $V'_{sub} \leftarrow$ MATRIXMUL(T, V_{sub});
    $H'_{sub} \leftarrow$ GETHRESP(V'_{sub});
    $P_{out} \leftarrow P_{out} \cup \{$POLYTOPE($V'_{sub}, H'_{sub}$)$\}$
return P_{out};

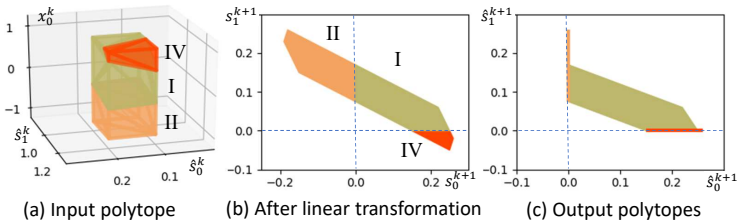
one layer as an input and maps it to a set of convex polytopes in the next layer. We first encode the given polytope and input/output relation of a layer as an MILP problem and use a solver to find sub-polytopes, such that each can be *linearly mapped* to obtain a convex polytope in the next layer (essentially a linearization of ReLU). In this process, we make use of both $\mathcal{H}$- and $\mathcal{V}$-representations. We omit showing the function PROPAGATEPOLYTOPE for the whole network, which simply loops through polytopes and layers using Algorithm 1.

The MILP encoding. We follow an MILP encoding [8] that uses binary indicator variables and slack variables for ReLU activation functions. We use the IBM ILOG CPLEX solver to solve the MILP problem. CPLEX computes all feasible assignments of the indicator variables. This is represented by the function GETFEASIBLEASSIGNMENTS in Algorithm 1.

Example C1. Consider an RNN with two neurons ($n = 2$) and one input, and the following weight matrices:

$$W_{rec} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & -0.4 \end{bmatrix}, W_{in} = \begin{bmatrix} 0.2 \\ -0.1 \end{bmatrix}, b_{rec} = \begin{bmatrix} -0.2 \\ 0.5 \end{bmatrix}$$

Suppose for a layer k, a polytope to propagate is given as: $H = \{0.1 \leq \hat{s}_0^k \leq 0.2, 1.0 \leq \hat{s}_1^k \leq 1.2\}, I = \{-1 \leq x_0^k \leq 1\}$ (where $\hat{s}^k$ represents $\max(s^k, 0)$). This is a polytope in 3-D space as shown in Figure 4(a). The linearly mapped polytope (before ReLU) is shown in Figure 4(b). By solving an MILP problem, we can get the set of feasible indicator variable assignments $\{(0,0),(1,0),(0,1)\}$, where the three tuples correspond



(a) Input polytope (b) After linear transformation (c) Output polytopes

Fig. 4. Polytope propagation on Example C1.

to the three pieces marked as I, II and IV, and value 1 of the indicator variable implies the corresponding ReLU is inactive (output stays 0), while value 0 indicates the ReLU is active (output is equal to input). Each of the three pieces is itself a polytope (therefore a sub-polytope) and will be mapped differently by the ReLUs, as the mapping from Figure 4(b) to (c) shows.

Get the representation of each sub-polytope. As each sub-polytope corresponds to a different ReLU activation status, its $\mathcal{H}$-representation can be constructed by adding the ReLU status constraints, which are generated by the function RELUSTATUSCONSTR. It constrains the sign of ReLU inputs according to a given indicator variable assignment. For instance, for sub-polytope II in Example C1, the two ReLUs are $(\text{inactive}, \text{active})$. Therefore it has the following ReLU status constraint (here $\otimes$ and $\leq$ are element-wise product and comparison on vectors):

$$(1, -1)^T \otimes \left(W_{rec}(\hat{s}_0^k, \hat{s}_1^k)^T + W_{in}x_0^k + b_{rec}\right) \leq \mathbf{0}$$

Apply transformation for each sub-polytope. For each sub-polytope obtained in the previous step, we construct a transformation matrix that captures its unique ReLU mapping and the same linear mapping related to the weights and bias. This matrix operates on the $\mathcal{V}$-representation (which can be obtained from $\mathcal{H}$-representation using conversion functions in libraries like PPL or CDD, as represented by GETVERTICES in Algorithm 1). After applying the transformation on vertices, we reconstruct the $\mathcal{H}$-representation of the new polytope using existing libraries (function GETHRESP). In Example C1 (b), sub-polytope I to IV are subject to different mapping, and the resulting polytopes are shown in Figure C1 (c).

An additional note on the implementation: when constructing the $\mathcal{H}$-representation we take special care for the degenerate polytopes (low-dimensional polytopes in a high-dimensional space). They are first projected into an appropriate low-dimensional space where we construct the $\mathcal{H}$-representation. We use singular value decomposition (SVD) for dimensionality reduction and its results provide the projection matrix to and from the low-dimensional space. This allows us to interchangeably use the QuickHull and the double description methods in computing $\mathcal{H}$-representation, as on certain cases one outperforms the other. On the other hand, ELINA [30] and PPL [5] use only double description method in the conversion as QuickHull suffers from degenerate cases.

The above describes finding $\mathcal{H}$- and $\mathcal{V}$-representation of a given polytope after applying linear and ReLU transformation. Our implementation uses generic polytope libraries (QuickHull and CDD) and existing algorithms. The results are sensitive to numerical errors that are incurred by underlying tools. We believe that these errors are insignificant, e.g. CPLEX precision error is 10^{-8}. This is a common trade-off between numeric and symbolic methods.

Bound estimation with polytope propagation. Propagating polytopes provides the most precise description of the reachable state space. However, it might be sufficient to just propagate an interval hull. For each polytope P, we estimate the interval bound on the output layers, which gives us a bounding box B around the output using the algorithm proposed by Xiao *et al.* [42]. If B is sufficient to conclude that network's output is safe, we skip the polytope propagation and use B instead.

D Joining polytopes

In Section 3.2, we use a "join" operation that produces a convex polytope containing the given polytopes. There are different choices on its implementation. Theoretically, to guarantee the termination of the above iterative procedure, the standard widening operator from the finite powerset of convex polytopes [4] can be used. However, it could be too abstract as each of its application either results in an increase of the dimension of the polytope or in a decrease of the number of constraints. On the other hand, the tight join achieved using the convex hull of polytopes could incur a higher computation cost, and it might take more iterations (or never) get to a fixed point. Here, we propose a light-weight join operator using *constraint relaxation*. For an $\mathcal{H}$-polytope $P : \{y \in \mathbb{R}^d \mid a_i^T y \leq b_i, i \in [1, m]\}$, joined with a $\mathcal{V}$-polytope $Q : conv(\{y^1, ..., y^n\})$, we relax the constraints $a_i^T y \leq b_i, i \in [1, m]$ for each vertex of Q. If for a vertex y^k, the left-hand-side of the i-th inequality constraint $a_i^T y$ is greater than the right-hand-side b_i, we increase b_i to match with $a_i^T y$. Geometrically, this is equivalent to translating the hyperplanes that form P to include Q. Additionally, to ensure the resulted polytope is still closed, we intersect it with the smallest box that contains all vertices of P and Q.

E Related work

A complete verification framework guarantees that a method either proves that the property holds or finds a counterexample to this property. For example, frameworks like Reluplex [15], Marabou [16], MIPVerify [35] provide complete verification algorithms. These frameworks are based on Satisfiablity Modulo Theories (SMT) or/and MILP search engines. Another complete verification approach is to perform reachability analysis [37,36]. Representing the exact reachable space often results in high computation cost. Therefore reachability analysis is usually combined with abstraction techniques [10,28], resulting in an incomplete verification framework. Compared to the previous work [37] that uses the star set representation for exact analysis, our usage of the generic $\mathcal{H}$- and $\mathcal{V}$-polytope representations allows us to leverage existing polytope libraries, and these representations are more friendly for convex hull computation and invariant construction.

An incomplete verification framework provides a method that either guarantees that a given property holds, or it remains unknown whether the property holds. Examples of such frameworks are FastLin [41], Crown [44], and DeepZ [27]. The main idea is to perform safe approximate reasoning about the behaviour of a network. If the approximate reasoning is sufficient to prove a property, an incomplete method succeeds; otherwise it fails. When the number of layers is large, e.g., more than a hundred layers, over-approximation methods tend to produce loose bounds which could negatively impact the ability of checking certain properties.

Another relevant line of work is invariant detection in feed-forward networks. The term *invariant* was previously used in the verification efforts of feed-forward networks to refer to a region in the input space that implies the same output property [18]. In another work [13], the authors propose to search for invariance properties that form decision patterns of neurons activations. In our work, the term invariant refers to an

inductive invariant (well-studied in state transition systems), i.e., it refers to a region in the input space of a layer whose image (output region) does not fall outside the region, under the same mode of operation.

Finally, Vengertsev '*et al.* [38] define a set of state and temporal safety properties for RNNs, and use a Monte Carlo approach to verify the defined properties. Their approach is based on probabilistic verification, inherently different from the reachability analysis.

F Experiments (additional tables)

Here we report the detailed experiment results for Property 1 and 2 (Table 2 and 3). For Property 1, in regions with Category I, the solving time is usually less than a second for our methods, whereas NNV-exact and NNV-app. will suffer from increasing numbers of star sets. After the rounding, CEGAR and inv. may report the same number in time as PP, but in general PP has the lowest overhead in region I. Although NNV-app. always terminates under the time limit, in some cases, it is not precise enough to prove validity of the property. Marabou (used as a comparison with SMT solvers) terminates only for some regions in Category I.

For Property 2, among the 48 stimulus regions, 3 are checked to guarantee the correct prediction even in presence of the spike on the opposite direction, 30 are shown to be vulnerable to the spike and the remaining 15 regions are still in unknown status (unlike Property 1, where there is no remaining stimulus regions left unchecked). The difficulty of Property 2 comes from the mix of challenges of both the large number of polytopes and the large number of facets in a polytope – this points to a direction for future work.

Table 2. Results of Experiments on Property 1 (time in seconds)

δ	μ	lower bound	upper bound	Category	PP	CEGAR	inv.	NNV-ex.	NNV-app.	Marabou	Fastest
	0.001	0.000871	0.001149	I	0.17	0.17	0.18	0.18	0.18	120.38	PP
	0.032	0.027858	0.036758	I	0.17	0.17	0.17	0.19	0.19	108.57	PP
0.2	0.128	0.111430	0.147033	I	0.17	0.17	0.17	T.O.	145.48	T.O.	PP
	0.512	0.445722	0.588134	I	0.17	0.17	0.17	T.O.	61.31	2489.49	PP
	1.024	0.891444	1.176267	I	0.17	0.18	0.17	T.O.	202.50	22.14	PP
	0.001	0.000708	0.001415	I	0.17	0.17	0.17	0.18	0.18	116.31	PP
	0.032	0.022627	0.045255	II	42.80	99.70	23.28	T.O.	(372.70)*	T.O.	inv
0.5	0.128	0.090510	0.181019	II	150.50	390.10	489.27	T.O.	(707.00)*	T.O.	PP
	0.512	0.362039	0.724077	II	T.O.	2026.70	942.63	T.O.	(104.10)*	T.O.	inv
	1.024	0.724077	1.448155	II	T.O.	T.O.	824.06	T.O.	273.60	T.O.	NNV-app.
	0.001	0.000616	0.001626	I	0.17	0.18	0.17	0.19	0.19	144.80	PP
	0.032	0.019698	0.051984	II	106.80	144.30	33.22	T.O.	(627.80)*	T.O.	inv
0.7	0.128	0.078793	0.207937	II	540.00	537.20	468.29	T.O.	(807.90)*	T.O.	inv
	0.512	0.315173	0.831746	II	T.O.	T.O.	5556.55	T.O.	290.50	T.O.	NNV-app.
	1.024	0.630346	1.663493	II	T.O.	2534.50	733.53	T.O.	411.80	T.O.	NNV-app.
	-0.001	-0.001149	-0.000871	I	0.16	0.16	0.16	0.14	0.14	140.90	NNV-ex.
	-0.032	-0.036758	-0.027858	I	0.16	0.16	0.16	0.24	0.24	216.09	PP
0.2	-0.128	-0.147033	-0.111430	I	0.16	0.16	0.16	0.38	0.38	199.49	PP
	-0.512	-0.588134	-0.445722	I	0.17	0.17	0.17	0.22	0.22	199.96	PP
	-1.024	-1.176267	-0.891444	I	0.16	0.16	0.16	0.23	0.23	5031.62	PP
	-0.001	-0.001415	-0.000708	I	0.16	0.16	0.16	0.11	0.11	139.00	NNV-ex.
	-0.032	-0.045255	-0.022627	I	0.16	0.16	0.16	0.26	0.26	352.46	PP
0.5	-0.128	-0.181019	-0.090510	I	0.16	0.16	0.16	0.68	0.68	518.39	PP
	-0.512	-0.724077	-0.362039	I	0.33	0.33	0.33	0.32	0.32	270.34	NNV-ex.
	-1.024	-1.448155	-0.724077	III	T.O.	T.O.	T.O.	0.32	0.32	T.O.	NNV-ex.
	-0.001	-0.001626	-0.000616	I	0.35	0.36	0.35	0.35	0.35	151.18	NNV-ex.
	-0.032	-0.051984	-0.019698	I	0.28	0.28	0.28	0.35	0.35	548.00	PP
0.7	-0.128	-0.207937	-0.078793	I	0.30	0.31	0.30	0.99	0.99	4920.96	PP
	-0.512	-0.831746	-0.315173	III	T.O.	T.O.	T.O.	0.55	0.55	T.O.	NNV-ex.
	-1.024	-1.663493	-0.630346	III	T.O.	T.O.	T.O.	0.53	0.53	T.O.	NNV-ex.

* NNV-app. terminates with unknown (abstraction is too coarse).

Table 3. Results of Experiments on Property 2 (time in seconds)

δ	μ	lower bound	upper bound	pulse position	Invariant	NNV-exact	Safe
0.2	0.001	0.000871	0.001149	1	18.75	0.70	✗
				2	3.5	0.43	✗
				3	4.5	0.35	✗
				4	5.3	0.37	✗
	0.032	0.027858	0.036758	1	3145.8	3.57	✗
				2	4546.6	1.25	✗
				3	23.8	0.41	✗
				4	24.8	0.35	✗
	0.128	0.111430	0.147033	1	724.4	T.O.	✓
				2	T.O.	T.O.	?
				3	1802.2	T.O.	✓
				4	12.9	T.O.	✓
0.7	0.032	0.019698	0.051984	1	T.O.	T.O.	?
				2	883.0	T.O.	✗
				3	T.O.	T.O.	?
				4	T.O.	T.O.	?
	0.128	0.078793	0.207937	1	T.O.	T.O.	?
				2	T.O.	T.O.	?
				3	T.O.	T.O.	?
				4	T.O.	T.O.	?
	0.512	0.315173	0.831746	1	T.O.	T.O.	?
				2	T.O.	T.O.	?
				3	T.O.	T.O.	?
				4	T.O.	T.O.	?
0.2	-0.001	-0.001149	-0.000871	1	511	46.12	✗
				2	304.6	43.74	✗
				3	504.6	41.37	✗
				4	470.6	39.82	✗
	-0.032	-0.036758	-0.027858	1	133.1	45.93	✗
				2	390.2	43.14	✗
				3	223.6	40.85	✗
				4	3291.5	38.78	✗
	-0.128	-0.147033	-0.111430	1	T.O.	46.23	✗
				2	T.O.	39.87	✗
				3	T.O.	35.52	✗
				4	21.9	32.54	✗
0.7	-0.032	-0.051984	-0.019698	1	207.3	45.92	✗
				2	1245.5	43.29	✗
				3	T.O.	40.11	✗
				4	T.O.	38.14	✗
	-0.128	-0.207937	-0.078793	1	T.O.	47.21	✗
				2	T.O.	37.89	✗
				3	T.O.	32.46	✗
				4	T.O.	26.16	✗
	-0.512	-0.831746	-0.315173	1	T.O.	45.21	✗
				2	T.O.	T.O.	?
				3	T.O.	T.O.	?
				4	T.O.	T.O.	?