Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis

Hongce Zhang[1 2] and **Maxwell Shinn**[3] and **Aarti Gupta**[1]
and **Arie Gurfinkel**[4] and **Nham Le**[4] and **Nina Narodytska**[5]

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 a simpler problem than the analysis of neural activity. However, in practice, reasoning about an RNN's behaviour is 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 counter-example 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, RNNs have been employed in cognitive neuroscience to help us to understand decision-making in humans and other animals [22, 25, 36, 28]. However, whether we use neural networks for a computer vision task or for a cognitive task, we would like to better understand the mechanistic process behind this technology.

One way to understand neural networks is to formally analyse their properties. Indeed, formal verification of neural networks is a rapidly growing research area [16, 44, 32]. 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 [32], or whether a controller avoids unnecessary turning action for an aircraft control problem [16].

Roughly speaking, there are two main approaches in verification: complete and incomplete methods. A complete verification framework guarantees that a method either proves that the property holds or finds a counterexample to this property. So, if it terminates, it gives an exact answer regarding satisfaction of a property to check [16, 20]. An incomplete verification framework provides a method that can either prove a property or it remains unknown whether the property holds. Unlike complete verification methods, incomplete methods cannot always provide definite answers. However, by judiciously using approximations, they can be more computationally efficient.

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 [16, 17, 47, 30], while verification of *recurrent* neural networks has received significantly less attention [3, 42]. One reason for this is 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 [3]. 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 is very challenging for both complete methods, where the number of neurons is large, and incomplete methods, where approximation errors accumulate with network depth.

In this work, we propose a novel approach to analyse RNNs. The main idea is to identify and exploit special properties of recurrent networks that allow us to reason about them efficiently. Here we focus on an important special class of recurrent neural networks which are trained to solve cognitive behavioral tasks [34], 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 [22]. Additionally, linear projections of RELU unit activations correspond qualitatively and quantitatively to neural activity as measured through electrophysiological recordings [22, 25, 36, 28]. Cognitive tasks can be solved with recurrent networks which have a shallow transition relation, a deep unfolding depth, and, more importantly, have only a few modes of operation. The domain specific properties of these networks make them amenable to formal analysis. Our proposed approach employs three building blocks for analysis of these networks: polytope propagation, invariant detection and counterexample-guided abstraction refinement (CEGAR) [7]. 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

[1] Princeton, USA, {hongcez, aartig}@princeton.edu
[2] This work was mostly done during internship at VMware Research.
[3] Yale University, USA, maxwell.shinn@yale.edu
[4] University of Waterloo, Canada, {arie.gurfinkel, nv3le}@uwaterloo.ca
[5] VMware Research, USA, nnarodytska@vmware.com

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 and 3, we will briefly introduce background on RNN and verification of neural networks. In Section 4, we explain why and how RNN is used in cognitive domains. Section 5 explains reachability analysis and Section 6 gives details of our verification methods for RNN, followed by experiments and discussion.

2 Background

We give details of feed-forward NNs and Mixed Integer Linear Programming (MILP) in Appendix A of the full version of the paper [1].

Recurrent Neural Networks. A *Recurrent Neural Network* (RNN) is a neural network that operates over a sequence of inputs [13]. 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:

$$s^{k+1} = f(W_{rec}s^k + W_{in}x^k + b_{rec}), \tag{1}$$

$$o^k = g(W_{out}s^k + b_{out}), \tag{2}$$

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 4. 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. [13]. 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 [15]. *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 [10] and PPL [6], which provide the functionality for the conversion between the two representations.

Safe Inductive Invariant of a State Transition System. An RNN can also be treated as state transition system. For a state transition system over state variables V, input variables Inp, transition relation Tr and initial states $Init$: $\langle V, Inp, Tr, Init\rangle$, and a set of error states Bad, a safe inductive invariant is defined as a formula Inv such that the following formulas are valid:

$$Init(V) \implies Inv(V) \tag{3}$$

$$Inv(V) \wedge Tr(V, Inp, V') \implies Inv(V') \tag{4}$$

$$Inv(V) \wedge Bad(V) \implies \bot \tag{5}$$

The safe inductive invariant Inv, if it exists, guarantees that the error states are unreachable. Additionally, assumptions on Inp may be added to describe invariants under a certain input condition. $Init$ in (3) can be replaced with a state predicate $p(V)$, which represents the safe inductive invariant for transitions starting from the particular set of states that satisfies predicate $p(V)$.

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. [45] 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.

3 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 [16], Marabou [17], MIPVerify [38] provide complete verification algorithms. These frameworks are based on Satisfiablity Modulo Theories (SMT) or/and MILP search engines. The main issue with these methods is scalability. For example, neural networks used for computer vision tasks contain millions of parameters. In practice, formalizations of large networks are challenging to reason about for modern solvers. For example, SMT-based verification frameworks, like Reluplex [16] and Marabou [17], are able to deal with networks containing about a thousand neurons [18]. Another complete verification approach is to perform reachability analysis [40, 39]. Representing the exact reachable space often results in high computation cost. Therefore reachability analysis is usually combined with abstraction techniques [11, 31], resulting in an incomplete verification framework. Compared to the previous work [40] 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 [44], Crown [47], and DeepZ [30]. 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

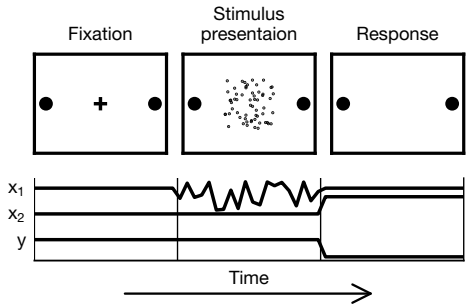


Figure 1. The random dot motion task [27, 29] as shown to subjects (above), and as adapted for a recurrent neural network via two inputs x_1 and x_2 and one expected output y (below).

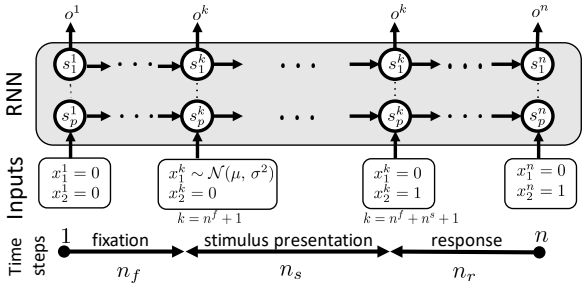


Figure 2. An unfolded RNN for n time steps. At each step the network consumes two inputs and emits an output. The state space is represented by state neurons s_j^k, $j \in [1, p]$, $k \in [1, n]$. Each mode of operation, fixation, stimulus and response, run for n_f, n_s and n_r time steps, respectively.

layers is large, e.g., more than a hundred layers, over-approximation methods tend to produce loose bounds and are not able to check 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 [21]. In another work [14], 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.* [41] 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, which is very different from our reachability analysis.

4 RNNs for cognitive domains

RNNs have become increasingly import in neurosicence, because they can perform adapted versions of the same cognitive tasks as experimental subjects. Trained RNNs exhibit behavioral similarities to subjects [34, 46], and also show patterns of RELU unit activations that correspond to signals from brain recordings [22, 25, 36, 28]. Thus, understanding the mechanism of action 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) [27, 29] is a cognitive task given to humans and other animals in order to study decision-making in the presence of noisy stimuli [12]. 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 [23, 24, 27, 12].

To adapt this task for a recurrent neural network (Figure 1), 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 2 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. Figure 1 shows an example of input signal (x_1^k and x_2^k, $k \in [1, n]$). The first input, x_1^k, corresponds to the stimulus signal. Note that it is constant over the fixation and response phases:

$$\text{if } k \in \text{SM} \text{ then } x_1^k \sim \mathcal{N}(\mu, \sigma^2) \text{ otherwise } x_1^k = 0 \tag{6}$$

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.

$$\text{if } k \in \text{RS} \text{ then } x_2^k = 1 \text{ otherwise } x_2^k = 0 \tag{7}$$

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, $k = 1, \ldots, n$:

$$s^{k+1} := \quad F(s^k, x^k) = W_{rec}\text{RELU}(s^k) + W_{in}x^k + b_{rec} \tag{8}$$

$$o^k := \quad O(s^k) = W_{out}\text{RELU}(s^k) + b_{out}, \tag{9}$$

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:

$$C_{\mathcal{R}}(o) = \begin{cases} 1 & \text{if } o^k \geq 0.5, \forall k \in [n-r, n], \\ -1 & \text{if } o^k \leq -0.5, \forall k \in [n-r, n], \\ 0 & \text{otherwise,} \end{cases} \quad (10)$$

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 transitions 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, namely a hundred layers (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:

$$\min_{k \in \text{SM}} (x_1^k) > p \Rightarrow C_{\mathcal{R}}(o) = 1, \quad (11)$$

$$\max_{k \in \text{SM}} (x_1^k) < p' \Rightarrow C_{\mathcal{R}}(o) = -1. \quad (12)$$

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 [23, 24] suggest that subjects integrate (in the mathematical sense) sensory evidence over time, recent approaches have questioned this perspective [35, 43, 48, 37]. 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.

$$(x_1^j < p') \wedge (\forall k \in \text{SM} \setminus \{j\}, x_1^k > p) \Rightarrow C_{\mathcal{R}}(o) = 1, \quad (13)$$

$$(x_1^j > p) \wedge (\forall k \in \text{SM} \setminus \{j\}, x_1^k < p') \Rightarrow C_{\mathcal{R}}(o) = -1. \quad (14)$$

If subjects do not integrate sensory evidence, alternative hypotheses imply that an instantaneous spike in evidence may trigger a choice [43, 37, 35], 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:

$$\max_{k \in [n-r, n]} (|\ o^k - o^{k-1}\ |) > 1, \quad (15)$$

where r is a parameter specified by a user. Subjects experience changes of mind in the random-dot motion task [26]. These changes were not explicitly discouraged in the behavioral task, and likewise, are not explicitly penalized by objective function.

5 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 [45]. 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 full version [1].

In the second phase, we perform property verification on RNN via reachability analysis. Suppose the property is of the form $C(s^0, x^0, x^1, ...) \implies P(o^n)$. In reachability analysis, we start from state s^0 and compute the set of states $Reach^{\langle i \rangle} : \{s^i \mid s^i = F(s^{i-1}, x^{i-1}), s^{i-1} \in Reach^{\langle i-1 \rangle}\}$ for each layer until reaching layer n, where F is as defined in (8). Then, we compute the output range based on $Reach^{\langle n \rangle}$ and check if it satisfies P. To this end, we need to (1) have a representation of the set of reachable states, and (2) 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 details are given in Section 6.1. As the number of polytopes usually increases along with the number of layers, we discuss two techniques in Section 6.2 and Section 6.3 to keep the representation tractable.

6 Verification of RNNs

In this paper, we use a union of convex polytopes to represent the set of reachable states. This is more precise than other similar representations, such as Zonotope [11]. In particular, our representation captures precisely (i.e., without any over-approximation) the output region of an RNN with ReLU activation functions applied to an input that is a finite union of polytopes.

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};

6.1 Polytope Propagation

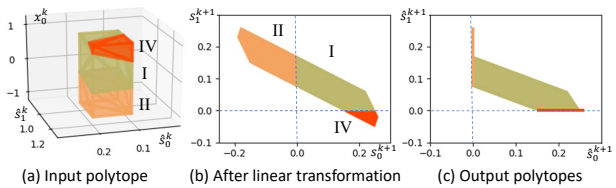
The core of reachability analysis is to efficiently propagate the reachable set throughout the network. Our general method of propagating a polytope over one layer is presented as the function PROPAGATEPOLYTOPEONELAYER in Algorithm 1. It takes a polytope in one layer as an input and maps it to a set 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 [9] 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 6.1. 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 3(a). The linearly mapped polytope (before ReLU) is shown in Figure 3(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 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 3(b) to (c) shows.



(a) Input polytope (b) After linear transformation (c) Output polytopes

Figure 3. Polytope propagation on Example 6.1.

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 6.1, the two ReLUs are (inactive, active). Therefore it has the following ReLU status constraint (here $\otimes$ and $\leq$ are element-wise product and comparison on vectors):

$$(1,-1)^T \otimes \left(W_{rec}(\hat{s}_0^k, \hat{s}_1^k)^T + W_{in}x_0^k + b_{rec}\right) \leq \vec{0}$$

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 6.1 (b), sub-polytope I to IV are subject to different mapping, and the resulting polytopes are shown in Figure 6.1 (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 projected into the appropriate low-dimensional space where we construct a low-dimensional $\mathcal{H}$-polytope. 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 method and the double description method in computing $\mathcal{H}$-representation, as on certain cases one outperforms the other. On the other hand, ELINA [33] and PPL [6] use only double description method in the conversion as QuickHull suffers from degenerate cases.

The above describes finding $\mathcal{H}$- and $\mathcal{V}$-representation after applying linear and ReLU transformation of a given polytope in computing the reachable set of a layer. It uses generic polytope libraries QuickHull and CDD and existing algorithms. Our 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. Recall that a big challenge in polytope propagation is coping with an increasing number of polytopes. A simple solution is to integrate interval arithmetic from Section 2 with polytope propagation. For each polytope P, we estimate the interval bound on the output layers $F(P)$, which gives us a bounding box B around $F(P)$. If B is sufficient to conclude that $F(P)$ is safe, we skip the computation of $F(P)$ and use B instead.

6.2 The CEGAR Approach

Abstraction via over-approximation is another 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 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,

Algorithm 2: CEGAR($F, P_{init}, I, T, N, S, U$): Polytope propagation using the CEGAR method.

Input: F: RNN's input-output relation of one layer, P_{init}: initial polytopes, I: input constraints, T: abstraction threshold, N: number of layers, S, and U: safe and unsafe regions, respectively.
Output: SAFE or UNSAFE.

1 $PolytopeSet \leftarrow P_{init}$; $layer \leftarrow 0$;
2 **while** $PolytopeSet \neq \emptyset$ **do**
3 $\quad OutputPolytopes \leftarrow$ PROPAGATEPOLYTOPE($F, S, I, T, N - layer$);
4 $\quad PolytopeSet \leftarrow \emptyset$;
5 $\quad$ **for each** $p \in OutputPolytopes$ **do**
6 $\quad\quad$ **if** $p \subseteq U$ **then return** UNSAFE;
7 $\quad\quad$ **if** $p \subseteq S$ **then continue**;
8 $\quad\quad$ $layer, P_{precise} \leftarrow$ GETPRECISE(p);
9 $\quad\quad$ **if** $layer = N$ **then return** UNSAFE;
10 $\quad\quad$ $PolytopeSet \leftarrow PolytopeSet \cup P_{precise}$
11 **return** SAFE;

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 [7]. For a polytope P that intersects with the unsafe region where some property fails (in other words, the polytope is ambiguous), 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 success in proving the property (Line 6) or disproving it (there exists at least one output polytope that is completely inside the unsafe region, Line 7). In either case, the CEGAR procedure on this polytope finishes. Otherwise, for the ambiguous polytopes, we again backtrace to the previous layer to refine the abstractions. Our detailed CEGAR procedure is presented in Algorithm 2. 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 [4].

6.3 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 in FNN, in RNN P and Q are comparable 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. 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. It is guaranteed that $P_1 \supseteq Q_0$ and $P_1 \supseteq P_0$. If P_1 contains its image Q_1, then

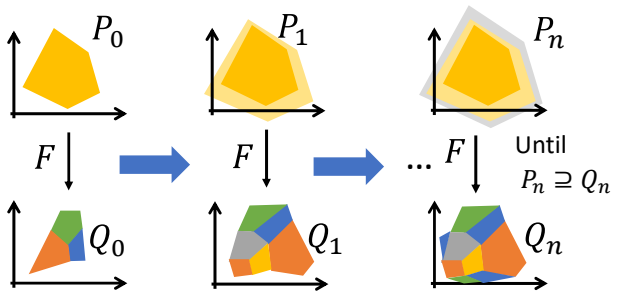


Figure 4. Illustration of constructing safe inductive invariant polytopes.

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 grows out of the safe region, 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 4.

In the above procedure, 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 [5] 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 called *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.

The invariants are relative to the input constraints. As our RNN operates in three modes with different input constraints, the invariants we construct are only for one mode of operation. As the states at the end of fixation period can be pre-computed, we only construct invariants in the stimulus period and response period. In order to estimate whether an invariant in the stimulus period is safe or not, we again use the interval arithmetic to estimate the bounds in the response period for a candidate invariant polytope, and terminate the iteration if such conservative estimation already concludes it is unsafe.

7 Experimental Evaluation

Network specification. The network specification is as described by transitions (8)–(9), 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 4. The parameters of the stimulus noise are as follows: μ is from a given set $C = \cup_{h \in \{0\} \cup [4,10]} \{-2^h/1000, 2^h/1000\}$ and $\sigma = 0.3$. We use TensorFlow via PsychRNN [2] to train RNNs. For the choice function (10), 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 $[c \cdot 2^{-\delta \operatorname{sign}(c)}, c \cdot 2^{\delta \operatorname{sign}(c)}]$ 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. There is no theoretical obstacle to integrate CEGAR and invariant, however, through experiments, we found that this often worsens the overall performance as the invariant construction will be using a more abstract polytope as the starting point, which will more likely result in a failure of reaching a safe inductive invariant.

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 [39]. 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.

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.

SPACER model checker. We have also experimented with SPACER [19], a state-of-the-art model checker included in Z3 [8]. 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.

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 maximum time limit is counted instead.

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.

Table 1. Summary of Results on Property 1 (Time in Seconds)

Regions		PP	CEGAR	Inv.	NNV-ex.	NNV-app.
Simple (19)	#. solved	19	19	19	16	19
	t_{mean}	0.2	0.2	0.2	947.6	21.8
Positive (8)	#. solved	4	7	8	0	3
	t_{mean}	3105.0	2216.6	1133.9	6000	449.4
Negative (3)	#. solved	0	0	0	3	3
	t_{mean}	6000	6000	6000	0.5	0.5
All (30)	#. solved	23	25	27	19	25
	#. fastest	16	0	4	7	3

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. The experiment results are summarized in Table 2 (detailed results can be found in the Appendix C in the full version [1]). 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.

Table 2. Summary of Results on Property 2

Regions		Inv.	NNV-ex.
All (48)	#. solved	23	29
	Ave. time (solved)	804.3	29.58
	Ave. time (all)	3510.4	2392.9
	Uniquely solved	4	4

Discussion and Lessons Learned. For Property 1, the difference of performance pattern 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 its invariants save 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, at the time of the containment check (required by the invariant method), one still needs to convert the star-set representation to projected $\mathcal{H}$- and $\mathcal{V}$-polytopes, as is implemented by NNV. And as the previous work noted, obtaining the convex hull (required by the CEGAR method) on the star set representation becomes more expensive [45].

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] Verification of recurrent neural networks for cognitive tasks via reachability analysis. https://github.com/nnarodytska/VERRNN/blob/master/rnn_veri.pdf, 2019.

[2] PsychRNN. https://github.com/dbehrlich/PsychRNN, last visited: Feb 18, 2020.

[3] M. E. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano, 'Verification of rnn-based neural agent-environment systems', in *AAAI, 2019*, pp. 6006–6013. AAAI Press, (2019).

[4] A. Albarghouthi, A. Gurfinkel, and M. Chechik, 'Craig interpretation', in *SAS 2012*, pp. 300–316, (2012).

[5] R. Bagnara, P. M. Hill, and E. Zaffanella, 'Widening operators for powerset domains', in *VMCAI 2004*, pp. 135–148, (2004).

[6] R. Bagnara, P. M. Hill, and E. Zaffanella, '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).

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, 'Counterexample-guided abstraction refinement', in *CAV*, pp. 154–169, Berlin, Heidelberg, (2000). Springer Berlin Heidelberg.

[8] L. De Moura and N. Bjørner, 'Z3: An efficient SMT solver', in *TACAS*, pp. 337–340, (2008).

[9] M. Fischetti and J. Jo, 'Deep neural networks and mixed integer linear optimization', *Constraints*, **23**(3), 296–309, (2018).

[10] K. Fukuda. An efficient implementation of the double description method. https://github.com/cddlib/cddlib, 2018.

[11] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. T. Vechev, '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).

[12] J. I. Gold and M. N. Shadlen, 'The neural basis of decision making', *Annual Review of Neuroscience*, **30**(1), 535–574, (jul 2007).

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, The MIT Press, 2016.

[14] D. Gopinath, A. Taly, H. Converse, and C. S. Pasareanu, 'Finding invariants in deep neural networks', *CoRR*, **abs/1904.13215**, (2019).

[15] B. Grünbaum, V. Kaibel, V. Klee, and G. Ziegler, *Convex Polytopes*, Graduate Texts in Mathematics, Springer, 2003.

[16] 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 *CAV*, pp. 97–117, (2017).

[17] 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 *CAV*, pp. 443–452, (2019).

[18] Y. Kazak, C. W. Barrett, G. Katz, and M. Schapira, 'Verifying deep-rl-driven systems', in *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019*, pp. 83–89. ACM, (2019).

[19] A. Komuravelli, A. Gurfinkel, and S. Chaki, 'SMT-Based Model Checking for Recursive Programs', in *CAV 2014*, pp. 17–34, (2014).

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

[21] S. Ma, Y. Liu, G. Tao, W. Lee, and X. Zhang, 'NIC: detecting adversarial samples with neural network invariant checking', in *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society, (2019).

[22] V. Mante, D. Sussillo, K. V. Shenoy, and W. T. Newsome, 'Context-dependent computation by recurrent dynamics in prefrontal cortex', *Nature*, **503**(7474), 78–84, (nov 2013).

[23] R. Ratcliff, 'A theory of memory retrieval.', *Psychological Review*, **85**(2), 59–108, (1978).

[24] R. Ratcliff, P. L. Smith, S. D. Brown, and G. McKoon, 'Diffusion decision model: Current issues and history', *Trends in Cognitive Sciences*, **20**(4), 260–281, (apr 2016).

[25] E. D. Remington, D. Narain, E. A. Hosseini, and M. Jazayeri, 'Flexible sensorimotor computations through rapid reconfiguration of cortical dynamics', *Neuron*, **98**(5), 1005–1019.e5, (jun 2018).

[26] A. Resulaj, R. Kiani, D. M. Wolpert, and M. N. Shadlen, 'Changes of mind in decision-making', *Nature*, **461**(7261), 263–266, (aug 2009).

[27] J. D. Roitman and M. N. Shadlen, '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, (November 2002).

[28] A. A. Russo, S. R. Bittner, S. M. Perkins, J. S. Seely, B. M. London, A. H. Lara, A. Miri, N. J. Marshall, A. Kohn, T. M. Jessell, L. F. Abbott, J. P. Cunningham, and M. M. Churchland, 'Motor cortex embeds muscle-like commands in an untangled population response', *Neuron*, **97**(4), 953–966.e8, (feb 2018).

[29] C. D. Salzman, K. H. Britten, and W. T. Newsome, 'Cortical microstimulation influences perceptual judgements of motion direction', *Nature*, **346**(6280), 174–177, (jul 1990).

[30] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev, 'Fast and effective robustness certification', in *NIPS*, pp. 10825–10836, (2018).

[31] G. Singh, T. Gehr, M. Püschel, and M. Vechev, 'An abstract domain for certifying neural networks', *Proc. ACM Program. Lang.*, **3**(POPL), 41:1–41:30, (January 2019).

[32] G. Singh, T. Gehr, M. Püschel, and M. T. Vechev, 'Boosting robustness certification of neural networks', in *ICLR 2019*, (2019).

[33] G. Singh, M. Püschel, and M. Vechev, 'Fast polyhedra abstract domain', in *ACM SIGPLAN Notices*, volume 52, pp. 46–59. ACM, (2017).

[34] H. F. Song, G. R. Yang, and X.-J. Wang, 'Training excitatory-inhibitory recurrent neural networks for cognitive tasks: A simple and flexible framework', *PLOS Computational Biology*, **12**(2), (feb 2016).

[35] G. M. Stine, A. Zylberberg, J. Ditterich, and M. N. Shadlen, 'Differentiating between integration and non-integration strategies in perceptual decision making', (January 2020).

[36] D. Sussillo, M. M. Churchland, M. T. Kaufman, and K. V. Shenoy, 'A neural network that finds a naturalistic solution for the production of muscle activity', *Nature Neuroscience*, **18**(7), 1025–1033, (jun 2015).

[37] D. Thura, J. Beauregard-Racine, C.-W. Fradet, and P. Cisek, 'Decision making by urgency gating: theory and experimental support', *Journal of Neurophysiology*, **108**(11), 2912–2930, (dec 2012).

[38] V. Tjeng, K. Y. Xiao, and R. Tedrake, 'Evaluating robustness of neural networks with mixed integer programming', in *ICLR*, (2019).

[39] H.-D. Tran, F. Cai, M. L. Diego, P. Musau, T. T. Johnson, and X. Koutsoukos, 'Safety verification of cyber-physical systems with reinforcement learning control', *ACM Trans. Embed. Comput. Syst.*, **18**(5s), 105:1–105:22, (October 2019).

[40] H. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson, 'Star-based reachability analysis of deep neural networks', in *FM2019*, LNCS, pp. 670–686. Springer, (2019).

[41] D. Vengertsev and E. Sherman, 'Recurrent neural network properties and their verification with monte carlo techniques', (2020).

[42] Q. Wang, K. Zhang, X. Liu, and C. L. Giles, 'Verification of recurrent neural networks through rule extraction', *CoRR*, **abs/1811.06029**, (2018).

[43] M. L. Waskom and R. Kiani, 'Decision making through integration of sensory evidence at prolonged timescales', *Current Biology*, **28**(23), 3850–3856.e9, (dec 2018).

[44] T. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. S. Boning, and I. S. Dhillon, 'Towards fast computation of certified robustness for relu networks', in *ICML*, pp. 5273–5282, (2018).

[45] K. Y. Xiao, V. Tjeng, N. M. M. Shafiullah, and A. Madry, 'Training for faster adversarial robustness verification via inducing relu stability', in *ICLR 2019*. OpenReview.net, (2019).

[46] G. R. Yang, M. R. Joglekar, H. F. Song, W. T. Newsome, and X.-J. Wang, 'Task representations in neural networks trained to perform many cognitive tasks', *Nature neuroscience*, **22**(2), 297, (2019).

[47] H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel, 'Efficient neural network robustness certification with general activation functions', in *NIPS*, pp. 4944–4953, (2018).

[48] D. M. Zoltowski, K. W. Latimer, J. L. Yates, A. C. Huk, and J. W. Pillow, 'Discrete stepping and nonlinear ramping dynamics underlie spiking responses of LIP neurons during decision-making', *Neuron*, **102**(6), 1249–1258.e10, (jun 2019).

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:

$$a^{\langle k+1 \rangle} = f_j(W^{\langle k \rangle} a^{\langle k \rangle} + b^{\langle k \rangle}), k \in [1, m-1] \quad (16)$$

$$o = f_m(W^{\langle m \rangle} a^{\langle m \rangle} + b^{\langle m \rangle}), \quad (17)$$

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, $k \in [1, m]$.

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

$$\min \sum_i c_i x_i \quad (18)$$

$$\text{subject to } \sum_i a_{ij} x_i \geq b_j, j \in [1, m] \quad (19)$$

$$x_i \in \mathbb{Z}, i \in \mathcal{I}_1,$$

$$x_i \in \mathbb{R}, i \in \mathcal{I}_2,$$

where $\mathcal{I}_1$ is a set of indices of integer variables and $\mathcal{I}_2$ is a set of indices of real variables, $\mathcal{I}_1 \cup \mathcal{I}_2 = [1, n]$.

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 [45]. 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 (18) used in a MILP formulation of a network [9]. For example, if we have a fully-connected layer then the number of variables in (18) equals to the number of neurons in the previous layer. The second issue is the presence of 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 equals to the number of neurons in the network. In [45], Xiao *et al.* initiated research on training easier to verify networks and introduced two techniques, weights sparsification and RELUs stabilization, that are very effective for feed-forward networks.

The main idea of RELUs stabilization is to train a network in such a way that piecewise 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 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 retraining 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.

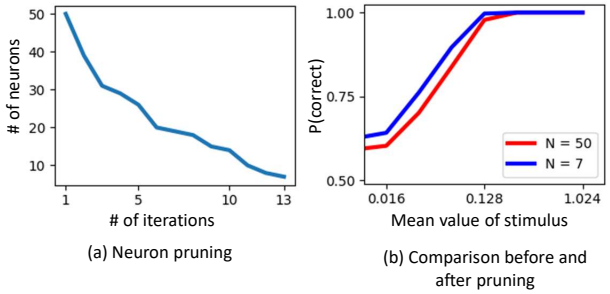


Figure 5. Reducing RNN for easier verification.

We also experimented with weights sparsification idea from [45]. This idea is based on: (a) use the L1 regularization during the training to encourage small weights and (b) set 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 [45]. However, it is not surprising. Note that if we zero one element of a transformation matrix in (8)–(9), 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 Experiments (additional tables)

Here we report the detailed experiment results for Property 1 and 2 (Table 3 and 4). 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 a mix of challenges from both the number of polytopes and the number of facets in a polytope – this points to a direction for future work.

Table 3. 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 4. 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.	?