Fast BATLLNN: Fast Box Analysis of Two-Level Lattice Neural Networks

James Ferlez
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
Irvine, CA, USA
jferlez@uci.edu

Haitham Khedr
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
Irvine, CA, USA
hkhedr@uci.edu

Yasser Shoukry
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
Irvine, CA, USA
yshoukry@uci.edu

ABSTRACT

In this paper, we present the tool Fast Box Analysis of Two-Level Lattice Neural Networks (Fast BATLLNN) as a fast verifier of box-like output constraints for Two-Level Lattice (TLL) Neural Networks (NNs). In particular, Fast BATLLNN can verify whether the output of a given TLL NN always lies within a specified hyper-rectangle whenever its input is constrained to a specified convex polytope (not necessarily a hyper-rectangle). Fast BATLLNN uses the unique semantics of the TLL architecture and the decoupled nature of box-like output constraints to dramatically improve verification performance relative to known polynomial-time verification algorithms for TLLs with generic polytopic output constraints. In this paper, we evaluate the performance and scalability of Fast BATLLNN, both in its own right and compared to state-of-the-art NN verifiers applied to TLL NNs. Fast BATLLNN compares very favorably to even the fastest NN verifiers, completing our synthetic TLL test bench more than 400x faster than its nearest competitor.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**.

KEYWORDS

Neural Networks, Neural Network Verification, Rectified Linear Units

ACM Reference Format:
James Ferlez, Haitham Khedr, and Yasser Shoukry. 2022. Fast BATLLNN: Fast Box Analysis of Two-Level Lattice Neural Networks. In *25th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '22), May 4–6, 2022, Milan, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3501710.3519533

1 INTRODUCTION

Neural Networks (NNs) increasingly play vital roles within safety-critical cyber-physical systems (CPSs), where they either make safety-critical decisions directly (e.g. low-level controllers) or influence high-level supervisory decision making (e.g. vision networks). Ensuring the safety of such systems thus demands algorithms for *formally verifying* the safety of their NN components. However, as CPS safety is characterized by *closed-loop* behavior, it is not enough to simply verify the input/output behavior of a NN component *once*. Such a verifier must also be as fast as possible, since it generally must be invoked many times to do closed-loop verification [29, 33].

In this paper, we propose Fast BATLLNN[1] as an input/output verifier for Rectified Linear Unit (ReLU) NNs with a special emphasis on execution time. In particular, Fast BATLLNN takes a relatively uncommon approach among verifiers in that it *explicitly trades off generality for execution time*: whereas most NN verifiers are designed to work for arbitrary deep NNs and arbitrary half-space output properties (or intersections thereof) [2], Fast BATLLNN instead forgoes this generality in network and properties to reduce verification time. That is, Fast BATLLNN is only able to verify *a very specific subset of deep NNs*: those characterized by a particular architecture, the Two-Level Lattice (TLL) NN architecture introduced in [14]; see Figure 1 and Section 2.3. Similarly, Fast BATLLNN is restricted to verifying *only "box"-like output constraints* (formally, hyper-rectangles). However, our experiments show that *Fast BATLLNN is 400-1900x faster than state-of-the-art general NN verifiers* when verifying the same TLL NNs and properties. Thus, Fast BATLLNN exemplifies that reduced generality can lead to dramatically faster verification, and in particular, it justifies the use of TLL NNs in *design for verifiability*.

In this sense, Fast BATLLNN is primarily inspired by the recent result [15], which showed that verifying a Two-Level Lattice (TLL) NN is an "easier" problem than verifying a general deep

[1] https://github.com/jferlez/FastBATLLNN

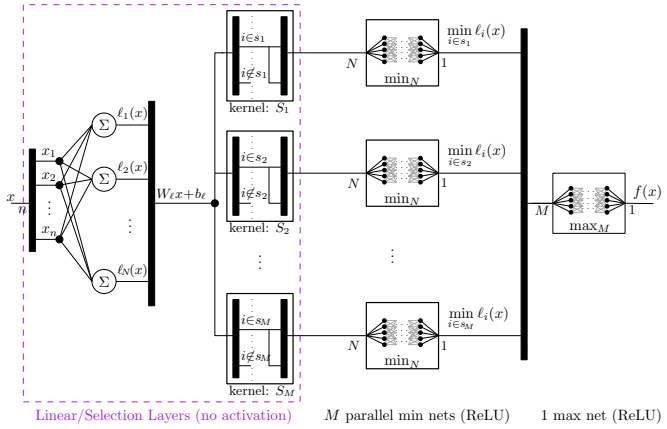

Figure 1: A TLL NN from $\mathbb{R}^n \to \mathbb{R}$ [14]; see also Section 2.3.

HSCC '22, May 4–6, 2022, Milan, Italy

ACM ISBN 978-1-4503-9196-2/22/05.
https://doi.org/10.1145/3501710.3519533

NN. Specifically, [15] exhibits a *polynomial time* algorithm to verify a TLL with respect to an arbitrary half-space output property (i.e. polynomial-time in the number of neurons). Indeed, the *semantic* structure of the TLL architecture is precisely what makes polynomial-time verification possible: in a TLL NN, the neuronal parameters provide direct (polynomial-time) access to each of the affine functions that appear in its response, viewed as a Continuous Piecewise Affine (CPWA) function[2] [15]. Since the same cannot be said of the neuronal parameters in a general deep NN, this indicates that considering only TLL NNs can facilitate a much faster verifier.

Thus, the major contribution of Fast BATLLNN is to further leverage the semantics of the TLL architecture under the additional assumption of verifying box-type (or hyper-rectangle) output properties. In particular, a TLL NN implements (component-wise) min and max lattice operations to compute each of its real-valued output components (as illustrated in Figure 1; see also Section 2.3). This fact can be used to dramatically simplify the verification of box-like output properties, which are component-wise *real-valued intervals* – and hence mutually decoupled. Importantly, the algorithm in [15] cannot leverage the TLL lattice operations in the same way, since it considers general half-space properties, which naturally *couple* the various output components of the TLL NN. As a result, we show that Fast BATLLNN has a big-O complexity whose crucial exponent is *half the size of the analogous exponent in* [15]. The performance consequences of this improvement are reflected in our experiments.

Before we proceed further, it is appropriate to make a few remarks about the restrictions inherent to Fast BATLLNN. Between the two restrictions of significance – the restriction to TLL NNs and the restriction to box-like output properties – the former is apparently more onerous: box-like properties can be used to adaptively assess more complicated properties whenever box-like properties are themselves inadequate. However even the restriction to TLL NNs is less imposing than at first it may seem. On the one hand, it is known that TLL NNs are capable of representing any CPWA [14, 26]; i.e., any function that continuously switches between a finite set of affine functions. Since deep NNs themselves realize CPWA functions, the TLL NN architecture is able to instantiate any function that a generic deep NN can. We do not consider the problem of converting a deep NN to the TLL architecture (nor the possible loss in parametric efficiency that may result), but the extremely fast verification times achievable with Fast BATLLNN suggest that the trade off is very likely worth the cost. On the other hand, a spate of recent results suggest that the TLL architecture is favorable for closed-loop controller design in the first place [9, 14, 16]—potentially obviating the need for such a conversion at all.

Related work: To the best of our knowledge, [15] is the only verification algorithm tailored to a restricted NN architecture.

The literature on more general NN verifiers is far richer. These NN verifiers can generally be grouped into three categories: (i) SMT-based methods, which encode the problem into a Satisfiability Modulo Theory problem [12, 21, 22]; (ii) MILP-based solvers, which directly encode the verification problem as a Mixed Integer Linear Program [3, 6–8, 18, 24, 27]; (iii) Reachability based methods, which perform layer-by-layer reachability analysis to compute the reachable set [5, 13, 19, 20, 29, 31, 35, 36]; and (iv) convex relaxations methods [10, 23, 30, 34]. Methods in categories (i) - (iii) tend to suffer from poor scalability, especially relative to convex relaxation methods. In this paper, we perform comparisons with state-of-the-art examples from category (iv) [23, 32] and category (iii) [5], as they perform well overall in the standard verifier competition [2].

[2]Recall that any ReLU NN also implements a CPWA: i.e., a function that continuously switches between finitely many affine functions.

2 PRELIMINARIES

2.1 Notation

We will denote the real numbers by $\mathbb{R}$. For an $(n \times m)$ matrix (or vector), A, we will use the notation $[\![A]\!]_{[i,j]}$ to denote the element in the i^{th} row and j^{th} column of A. Analogously, the notation $[\![A]\!]_{[i,:]}$ will denote the i^{th} row of A, and $[\![A]\!]_{[:,j]}$ will denote the j^{th} column of A; when A is a vector instead of a matrix, both notations will return a scalar corresponding to the corresponding element in the vector. We will use angle brackets $\langle \cdot \rangle$ to delineate the arguments to a function that *returns a function*. We use one special form of this notation: for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ and $i \in \{1, \dots, m\}$ define

$$\pi_i\langle f \rangle : x \mapsto [\![f(x)]\!]_{[i,:]}. \tag{1}$$

2.2 Neural Networks

We will exclusively consider Rectified Linear Unit Neural Networks (ReLU NNs). A K-layer ReLU NN is specified by K *layer* functions, of which we allow two kinds: linear and nonlinear. A layer of either type is defined in terms of a parameter list $\theta \triangleq (W, b)$ where W is a $(\overline{d} \times \underline{d})$ matrix and b is a $(\overline{d} \times 1)$ vector. Specifically, the *linear* and *nonlinear* layers specified by θ are denoted by L_θ and $L^\sharp_\theta$, respectively, and they are defined as:

$$L_\theta : \mathbb{R}^{\underline{d}} \to \mathbb{R}^{\overline{d}}, \qquad L_\theta : z \mapsto Wz + b \tag{2}$$

$$L^\sharp_\theta : \mathbb{R}^{\underline{d}} \to \mathbb{R}^{\overline{d}}, \qquad L^\sharp_\theta : z \mapsto \max\{L_\theta(z), 0\}. \tag{3}$$

where the max function is taken element-wise. Thus, a K-layer ReLU NN function is specified by functionally composing K such layer functions whose parameters $\theta^{|i}, i = 1, \dots, K$ have dimensions that satisfy $\underline{d}^{|i} = \overline{d}^{|i-1} : i = 2, \dots, K$; we will consistently use the superscript notation $^{|k}$ to identify a parameter with layer k. Whether a layer function is linear or not will be further specified by a set of linear layers, $\texttt{lin} \subseteq \{1, \dots, K\}$. For example, a typical K-layer NN has $\texttt{lin} = \{K\}$, which together with a list of K layer parameters defines the NN:

$$\mathcal{NN} = L_{\theta^{|K}} \circ L^\sharp_{\theta^{|K-1}} \circ \cdots \circ L^\sharp_{\theta^{|1}}. \tag{4}$$

To make the dependence on parameters explicit, we will index a ReLU function $\mathcal{NN}$ by a *list of NN parameters* $\Theta \triangleq (\texttt{lin}, \theta^{|1}, \dots, \theta^{|K})$; in this respect, we will often use $\mathcal{NN}\langle\Theta\rangle : \mathbb{R}^{\underline{d}^{|1}} \to \mathbb{R}^{\overline{d}^{|K}}$.

2.3 Two-Level-Lattice (TLL) Neural Networks

In this paper, we are exclusively concerned with Two-Level Lattice (TLL) ReLU NNs. In this subsection, we formally define NNs with the TLL architecture using the succinct method exhibited in [15]; the material in this subsection is derived from [14, 15].

The most efficient way to characterize a TLL NN is by way of three generic NN composition operators. Hence, we have the

following three definitions, which serve as auxiliary results in order to eventually define a TLL NN in Definition 4.

Definition 1 (Sequential (Functional) Composition). *Let $\mathcal{NN}\langle\Theta_i\rangle : \mathbb{R}^{\underline{d}_i^{|1}} \to \mathbb{R}^{\overline{d}_i^{|K_i}}$, $i = 1, 2$ be two NNs with parameter lists $\Theta_i \triangleq (\texttt{lin}_i, \theta_i^{|1}, \ldots, \theta_i^{|K_i})$, $i = 1, 2$ such that $\overline{d}_1^{|K_1} = \underline{d}_2^{|1}$. Then the **sequential (or functional) composition** of $\mathcal{NN}\langle\Theta_1\rangle$ and $\mathcal{NN}\langle\Theta_2\rangle$, i.e. $\mathcal{NN}\langle\Theta_1\rangle \circ \mathcal{NN}\langle\Theta_2\rangle$, is a NN that is represented by the parameter list $\Theta_1 \circ \Theta_2 \triangleq (\texttt{lin}_1 \cup (\texttt{lin}_2 + K_1), \theta_1^{|1}, \ldots, \theta_1^{|K_1}, \theta_2^{|1}, \ldots, \theta_2^{|K_2})$, where $\texttt{lin}_2 + K_1$ is an element-wise sum.*

Definition 2. *Let $\mathcal{NN}\langle\Theta_i\rangle : \mathbb{R}^{\underline{d}_i^{|1}} \to \mathbb{R}^{\overline{d}_i^{|K}}$, $i = 1, 2$ be two K-layer NNs with parameter lists $\Theta_i = (\texttt{lin}, (W_i^{|1}, b_i^{|1}), \ldots, (W_i^{|K}, b_i^{|K}))$, $i = 1, 2$ such that $\underline{d}_1^{|1} = \underline{d}_2^{|1}$; also note the common set of linear layers, $\texttt{lin}$. Then the **parallel composition** of $\mathcal{NN}\langle\Theta_1\rangle$ and $\mathcal{NN}\langle\Theta_2\rangle$ is a NN given by the parameter list*

$$\Theta_1 \parallel \Theta_2 \triangleq \left(\texttt{lin}, \left(\begin{bmatrix} W_1^{|1} \\ W_2^{|1} \end{bmatrix}, \begin{bmatrix} b_1^{|1} \\ b_2^{|1} \end{bmatrix}\right), \left(\begin{bmatrix} W_1^{|2} & \mathbf{0} \\ \mathbf{0} & W_2^{|2} \end{bmatrix}, \begin{bmatrix} b_1^{|2} \\ b_2^{|2} \end{bmatrix}\right), \ldots, \left(\begin{bmatrix} W_1^{|K} & \mathbf{0} \\ \mathbf{0} & W_2^{|K} \end{bmatrix}, \begin{bmatrix} b_1^{|K} \\ b_2^{|K} \end{bmatrix}\right)\right) \quad (5)$$

where $\mathbf{0}$ is a sub-matrix of zeros of the appropriate size. That is $\Theta_1 \| \Theta_2$ accepts an input of the same size as (both) Θ_1 and Θ_2, but has as many outputs as Θ_1 and Θ_2 combined.

Definition 3 (n-element min/max NNs). *An n-**element** min **network** is denoted by the parameter list $\Theta_{\min_n}$. $\mathcal{NN}\langle\Theta_{\min_n}\rangle : \mathbb{R}^n \to \mathbb{R}$ such that $\mathcal{NN}\langle\Theta_{\min_n}\rangle(x)$ is the minimum from among the components of x (i.e. minimum according to the usual order relation $<$ on $\mathbb{R}$). An n-**element** max **network** is denoted by $\Theta_{\max_n}$, and functions analogously. These networks are described in [14].*

With Definition 1 through Definition 3 in hand, it is now possible for us to define TLL NNs in the same way as [15]. We likewise proceed to first define a *scalar (or real-valued)* TLL NN; the structure of such a scalar TLL NN is illustrated in Figure 1. Then we extend this notion to a *multi-output (or vector-valued)* TLL NN.

Definition 4 (Scalar TLL NN [15]). *A NN that maps $\mathbb{R}^n \to \mathbb{R}$ is said to be **TLL NN of size** (N, M) if its parameter list $\Xi_{N,M}$ can be characterized entirely by integers N and M as follows.*

$$\Xi_{N,M} \triangleq \Theta_{\max_M} \circ \left((\Theta_{\min_N} \circ \Theta_{S_1}) \parallel \ldots \parallel (\Theta_{\min_N} \circ \Theta_{S_M})\right) \circ \Theta_\ell \quad (6)$$

where

- $\Theta_\ell \triangleq (\{1\}, \theta_\ell)$ *for* $\theta_\ell \triangleq (W_\ell, b_\ell)$;
- *each Θ_{S_j} has the form $\Theta_{S_j} = \left(\{1\}, (S_j, \mathbf{0})\right)$ where $\mathbf{0}$ is the column vector of N zeros, and where*
  - $S_j = \left[[\![I_N]\!]_{[\iota_1,:]}^T \ \ldots \ [\![I_N]\!]_{[\iota_N,:]}^T \right]^T$ *for a length-N sequence $\{\iota_k\}$ where $\iota_k \in \{1, \ldots, N\}$ and I_N is the $(N \times N)$ identity matrix.*

*The affine functions implemented by the mapping $\ell_i \triangleq \pi_i\langle L_{\theta_\ell}\rangle$ for $i = 1, \ldots, N$ will be referred to as the **local linear functions** of $\Xi_{N,M}$; we assume for simplicity that these affine functions are unique. The matrices $\{S_j | j = 1, \ldots, M\}$ will be referred to as the **selector matrices** of $\Xi_{N,M}$. Each set $s_j \triangleq \{k \in \{1, \ldots, N\} | \exists \iota \in \{1, \ldots, N\}. [\![S_j]\!]_{[\iota,k]} = 1\}$ is said to be the **selector set of** S_j.*

Definition 5 (Multi-output TLL NN [15]). *A NN that maps $\mathbb{R}^n \to \mathbb{R}^m$ is said to be a **multi-output TLL NN of size** (N, M) if its parameter list $\Xi_{N,M}^{(m)}$ can be written as*

$$\Xi_{N,M}^{(m)} = \Xi_{N,M}^1 \parallel \cdots \parallel \Xi_{N,M}^m \quad (7)$$

*for m equally-sized scalar TLL NNs, $\Xi_{N,M}^1, \ldots, \Xi_{N,M}^m$; these scalar TLLs will be referred to as the **(output) components of** $\Xi_{N,M}^{(m)}$.*

2.4 Hyperplanes and Hyperplane Arrangements

Here we review notation for hyperplanes and hyperplane arrangements; these results will be important in the developemnt of Fast BATLLNN. [25] is the main reference for this section.

Definition 6 (Hyperplanes and Half-spaces). *Let $l : \mathbb{R}^n \to \mathbb{R}$ be an affine map. Then define:*

$$H_l^q \triangleq \begin{cases} \{x | l(x) < 0\} & q = -1 \\ \{x | l(x) > 0\} & q = +1 \\ \{x | l(x) = 0\} & q = 0. \end{cases} \quad (8)$$

*We say that H_l^0 is the **hyperplane defined by** l **in dimension** n, and H_l^{-1} and H_l^{+1} are the **negative and positive half-spaces defined by** l, respectively.*

Definition 7 (Hyperplane Arrangement). *Let $\mathcal{L}$ be a set of affine functions where each $l \in \mathcal{L} : \mathbb{R}^n \to \mathbb{R}$. Then $\{H_l^0 | l \in \mathcal{L}\}$ is an **arrangement of hyperplanes in dimension** n.*

Definition 8 (Region of a Hyperplane Arrangement). *Let $\mathcal{H}$ be an arrangement of N hyperplanes in dimension n defined by a set of affine functions, $\mathcal{L}$. Then a non-empty subset $R \subseteq \mathbb{R}^n$ is said to be a **region of** $\mathcal{H}$ if there is an indexing function $\mathfrak{s} : \mathcal{L} \to \{-1, 0, +1\}$ such that $R = \bigcap_{l \in \mathcal{L}} H_l^{\mathfrak{s}(l)}$; R is said to be n-**dimensional or full-dimensional** if it is non-empty and described by an indexing function $\mathfrak{s}(l) \in \{-1, +1\}$ for all $l \in \mathcal{L}$.*

Theorem 1 ([25]). *Let $\mathcal{H}$ be an arrangement of N hyperplanes in dimension n, and let $\mathcal{R}_\mathcal{H}$ be the set of its full-dimensional regions. Then $|\mathcal{R}_\mathcal{H}|$ is at most $\sum_{k=0}^n \binom{N}{k}$.*

Remark 1. *Note that for a fixed dimension, n, the bound $\sum_{k=0}^n \binom{N}{k}$ grows like $O(N^n/n!)$, i.e. polynomially in N.*

3 PROBLEM FORMULATION

The essence of Fast BATLLNN is its focus on verifying TLL NNs with respect to box-like output constraints. Formally, Fast BATLLNN considers only verification problems of the following form (stated using notation from Section 2).

Problem 1. *Let $\mathcal{NN}\langle\Xi_{N,M}^{(m)}\rangle : \mathbb{R}^n \to \mathbb{R}^m$ be a multi-output TLL NN. Also, let:*

- *$P_X \subset \mathbb{R}^n$ be a closed, convex polytope specified by the intersection of N_X half-spaces, i.e. $P_X \triangleq \cap_{i=1}^{\mathsf{N}_X} \{x : l_{X,i}(x) \leq 0\}$ where each $l_{X,i} : \mathbb{R}^n \to \mathbb{R}$ is affine; and*
- *$P_Y \subset \mathbb{R}^m$ be closed hyper-rectangle, i.e. $P_Y \triangleq \prod_{k=1}^m [a_k, b_k]$ with $-\infty \leq a_k \leq b_k \leq \infty$ for each k.*

Then the verification problem is to decide whether the following formula is true:

$$\forall x \in P_X \subset \mathbb{R}^n. \left(\mathcal{NN}\langle\Xi_{N,M}^{(m)}\rangle(x) \in P_Y \subset \mathbb{R}^m\right). \quad (9)$$

If (9) *is true, the problem is* ***SAT****; otherwise, it is* ***UNSAT****.*

Note that the properties (and their interpretations) in Problem 1 *are dual to the usual convention*; it is more typical in the literature to associate "unsafe" outputs with a closed, convex polytope, and then the *existence* of such unsafe outputs is denoted by **UNSAT** (see [28] for example). However, we chose this formulation for Problem 1 because it is the one adopted by [15], and because it is more suited to NN reachability computations, one of the motivating applications of Fast BATLLNN. Indeed, to verify a property like (9), the typical dual formulation of Problem 1 would require $2 \cdot m$ verifier calls, assuming unbounded polytopes are verifiable (and then the verification would only be with respect to the interior of P_Y). Of course this choice comes with a trade-off: Fast BATLLNN, which directly solves Problem 1, requires adaptation to verify the dual property of Problem 1; we return to this briefly at the end of this section, but it is ultimately left for future work.

In the case of Fast BATLLNN, there is another important reason to consider the stated formulation of Problem 1: both the output property P_Y and the NN $\Xi^{(m)}_{N,M}$ have an essentially *component-wise nature* (see also Definition 5). As a result, a component-wise treatment of Problem 1 greatly facilitates the development and operation of Fast BATLLNN. To this end, we will find it convenient in the sequel to consider the following two verification problems; each is specified for a *scalar TLL NN and a single real-valued output property.* Moreover, we cast them in terms of the negation of the analogous formula derived from Problem 1; the reasons for this will become clear in Section 4.

Problem 1A (Scalar Upper Bound). *Let* $\mathcal{NN}\langle \Xi_{N,M} \rangle : \mathbb{R}^n \to \mathbb{R}$ *be a* ***scalar TLL NN****, and let* $P_X \triangleq \cap_{i=1}^{\mathsf{N}_X} \{x : l_{X,i}(x) \leq 0\}$ *be a closed convex polytope as in Problem 1.*

Then the ***scalar upper bound verification problem for*** $b \in \mathbb{R}$ *is to decide whether the following formula is true:*

$$\exists x \in P_X \subset \mathbb{R}^n . \big(\mathcal{NN}\langle \Xi_{N,M} \rangle(x) > b\big). \tag{10}$$

If (10) *is true, the problem is* ***UNSAT****; otherwise, it is* ***SAT****.*

Problem 1B (Scalar lower Bound). *Let* $\mathcal{NN}\langle \Xi_{N,M} \rangle : \mathbb{R}^n \to \mathbb{R}$ *and* P_X *be as in Problem 1A.*

Then the ***scalar lower bound verification problem for*** $a \in \mathbb{R}$ *is to decide whether the following formula is true:*

$$\exists x \in P_X \subset \mathbb{R}^n . \big(\mathcal{NN}\langle \Xi_{N,M} \rangle(x) < a\big). \tag{11}$$

If (11) *is true, the problem is* ***UNSAT****; otherwise, it is* ***SAT****.*

Thus, note that the formulation of Problem 1 is such that it can be verified by evaluating a boolean formula that contains only instances of Problem 1A and Problem 1B. That is, the following formula has the same truth value as (9):

$$\bigwedge_{k=1}^{m} \Bigg(\neg\Big(\exists x \in P_X \subset \mathbb{R}^n . \big(\mathcal{NN}\langle \Xi^k_{N,M} \rangle(x) < a_k\big)\Big) \quad \wedge \quad \neg\Big(\exists x \in P_X \subset \mathbb{R}^n . \big(\mathcal{NN}\langle \Xi^k_{N,M} \rangle(x) > b_k\big)\Big)\Bigg). \tag{12}$$

We reiterate, however, that the same is not true of the *dual* property to Problem 1. Consequently, Fast BATLLNN requires modification to verify such properties; this is a more or less straightforward procedure, but we defer this to future work, as noted above.

4 FAST BATLLNN: THEORY

In this section, we develop the theoretical underpinnings of Fast BATLLNN. As noted in Section 3, the essential insight of our algorithm is captured by our solutions to problems Problem 1A and Problem 1B. Thus, this section is organized primarily around solving sub-problems of these forms; at the end of this section, we will show how to combine these results into a verification algorithm for Problem 1, and then we will analyze the overall computational complexity of Fast BATLLNN.

4.1 Verifying Problem 1A

Problem 1A, as stated above, regards the TLL NN to be verified merely as a map from inputs to outputs; this is the behavior that we wish to verify, after all. However, this point of view obscures the considerable semantic structure intrinsic to the neurons in a TLL NN. In particular, recall that $\mathcal{NN}\langle \Xi_{N,M} \rangle$ implements the following function, which was derived from the Two-Level Lattice representation of CPWAs – see Section 2.3 and [14, 26]:

$$\mathcal{NN}\langle \Xi_{N,M} \rangle(x) = \max_{1 \leq j \leq M} \min_{i \in s_j} \ell_i(x). \tag{13}$$

In (13), the sets s_j are the *selector sets* of $\mathcal{NN}\langle \Xi_{N,M} \rangle$ and the ℓ_i are the *local linear functions* of $\mathcal{NN}\langle \Xi_{N,M} \rangle$; both terminologies are formally defined in Definition 4. Upon substituting (13) into (10), we obtain the following, far more helpful representation of the property expressed in Problem 1A:

$$\exists x \in P_X \subset \mathbb{R}^n . \big(\max_{1 \leq j \leq M} \min_{i \in s_j} \ell_i(x) > b\big). \tag{14}$$

Literally, (14) compares the output property of interest, $b \in \mathbb{R}$, with a combination of *real-valued* max/min operations applied to scalar affine functions. Crucially, that comparison is made using the usual order relation on $\mathbb{R}$, $\geq$, which is exactly the same order relation upon which the max and min operations are based.

Thus, (14) can be simplified as follows. First note that the result of the max operation in (14) can exceed b on P_X if and only if:

$$\exists x \in P_X . \exists j \in \{1, \ldots, M\} . \Big(\min_{i \in s_j} \ell_i(x) > b\Big). \tag{15}$$

In turn, the result of any one of the min operations in (14) can exceed b on P_X, and hence make (15) true, if and only if

$$\exists x \in P_X . \forall i \in s_j . \Big(\ell_i(x) > b\Big). \tag{16}$$

In particular, (16) is actually an intersection of *half spaces*, some open and some closed: the open half spaces come from local linear functions that violate the property; and the closed half-spaces come from the input property, P_X (see Problem 1). Moreover, there are at most M such intersections of relevance to Problem 1A: one for each of the $j = 1, \ldots, M$ such min operations present in (14). Finally, note that linear feasibility problems consisting entirely of *non-strict* inequality constraints are easy to solve: this suggests that we should first amend the $> b$ inequality with $\geq b$ before proceeding.

Formally, these ideas are captured in the following proposition.

Proposition 1. *Consider an instance of Problem 1A. Then that instance is* ***UNSAT*** *if and only if the set:*

$$\bigcup_{j=1,\ldots,M} \left(\bigcap_{i \in s_j} \{x : \ell_i(x) > b\} \cap \bigcap_{i=1}^{\mathsf{N}_X} \{x : l_{X,i}(x) \leq 0\} \right) \neq \emptyset. \tag{17}$$

Or equivalently, if for at least one of the $j = 1, \ldots, M$, the linear feasibility problem specified by the constraints

$$F_j \triangleq \begin{cases} \ell_{i_1}(x) \geq b \\ \vdots \\ \ell_{i_{|s_j|}}(x) \geq b \end{cases} \bigwedge \begin{cases} l_{X,1}(x) \leq 0 \\ \vdots \\ l_{X,\mathsf{N}_X}(x) \leq 0 \end{cases} \text{for } \{i_1, \ldots, i_{|s_j|}\} = s_j \tag{18}$$

*is feasible, **and** one of the following conditions is true:*

- *it has non-empty interior; or*
- *there is a feasible point that lies only on some subset of the $\{l_{X,i} : i = 1, \ldots, \mathsf{N}_X\}$.*

Proof. The first claim follows immediately from the manipulations described in (15) and (16). The second claim merely exhausts the possibilities for how the constraints $\ell_i(x) > b$ can participate in a feasible set for the linear program given by F_j. □

Remark 2. *The conclusion of Proposition 1 also has the following important interpretation: the $\exists x.(\cdots > b)$ property can be seen to "distribute across" the* max/min *operations in* (14)*, and upon doing so, it converts the lattice operation* max *into **set union** and the lattice operation* min *into **set intersection**. Furthermore, since a TLL NN is constructed from two levels of lattice operations applied to affine functions, the innermost lattice operation of* min *is converted into a set intersection of half-spaces — i.e., a linear feasibility problem.*

Of course Proposition 1 also suggests a natural and obvious algorithm to verify an instance of Problem 1A. The pseudocode for this algorithm appears as the function `verifyScalarUB` in Algorithm 1, and its correctness follows directly from Proposition 1. In particular, `verifyScalarUB` simply evaluates the feasibility of each set of constraints $F_j, j = 1, \ldots, M$ in turn, until either a feasible problem is found or the list is exhausted. Then for each such feasible F_j, `verifyScalarUB` attempts to find an interior point of the feasible set to reconcile it with the desired inequalities in (17); failing that, it searches for a vertex of the feasible set where no output property constraints are active. In practice, these operations can be combined by operating on the feasible point returned by the original feasibility program: an LP can be used to maximize the value of each active constraint in order to explore adjacent vertices. Note further that `verifyScalarUB` may not need to execute all M possible linear programs for properties that are **UNSAT**: it can terminate early on the first "satisfied" linear program found.

```
input : b ∈ ℝ, a upper bound to verify
        Ξ_{N,M}, parameters of a TLL NN to verify
        L_X = {l_{X,1}, ..., l_{X,N_X}}, affine functions
          specifying an input constraint polytope, P_X
output: Boolean (True = SAT; False = UNSAT)
function verifyScalarUB(b, Ξ_{N,M}, L_X)
    for j = 1, ..., M do
        constraints ← [(ℓ_i(x) ≥ b) for i ∈ s_j]
        constraints.append([l_{X,1}(x) ≤ 0, ..., l_{X,N_X}(x) ≤ 0])
        (sol, status) ← SolveLinFeas(constraints)
        if status == Feasible then
            if all([ℓ_i( sol ) > b for i = 1 ... N])
            or FindInt(constraints) == True then
                return False
            end
        end
    end
    return True
end
```

Algorithm 1: `verifyScalarUB`; i.e., solve Problem 1A

4.2 Verifying Problem 1B

Naturally, we start our consideration of Problem 1B in very much the same way as Problem 1A. However, given that Problem 1B and Problem 1A are in some sense *dual*, the result is not nearly as convenient. In particular, substituting (13) into (11), and attempting carry out the same sequence of manipulations that led to Proposition 1 results in the following formula:

$$\bigcap_{j=1,\ldots,M} \left(\bigcup_{i \in s_j} \{x : \ell_i(x) < a\} \cap \bigcap_{i=1}^{\mathsf{N}_X} \{x : l_{X,i}(x) \leq 0\} \right) \neq \emptyset, \tag{19}$$

which has the same truth value as formula (11). Unfortunately, (19) is not as useful as the result in Proposition 1: under the "dual" output constraint $< a$, set intersection and union are switched relative to Proposition 1. Consequently, (19) is not a direct formulation in terms of intersections of half-spaces — i.e., linear feasibility problems.

Nevertheless, rearranging (19) into the union-of-half-space-intersections form of Proposition 1 is possible and profitable. Using basic set operations, it is possible to rewrite (19) in a union-of-intersections form as follows (the set intersection $P_X = \cap_{i=1}^{\mathsf{N}_X} \{x : l_{X,i}(x) \leq 0\}$ is moved outside the outer union for convenience):

$$P_X \cap \bigcup_{\substack{(i_1,\ldots,i_M) \\ \in s_1 \times \cdots \times s_M}} \bigcap_{k=1,\ldots,M} \{x : \ell_{i_k}(x) < a\} \neq \emptyset. \tag{20}$$

By construction, (20) again has the same truth value as (11), but it is now in the desired form. In particular, it is verifiable by evaluating a finite number of half-space intersections much like Proposition 1.

Unfortunately, as a result of this rearrangement, the total number of mutual half-space intersections – or intersection "terms" – has grown from M to $\prod_{j=1}^{M} |s_j|$, where $|s_j|$ is the cardinality of the selector set, s_j. In particular, this number can easily exceed M: for example, if each of the s_j has exactly two elements, then there are 2^M total mutual intersection terms. Thus, verifying (20) in its current form would appear to require (in the worst case) exponentially more linear feasibility programs than the verifier we proposed for Problem 1A. This situation is not only non-ideal in terms of run-time: it would also seem to contradict [15], which describes an algorithm with polynomial-time complexity in N and M — and that algorithm is after all applicable to *more general* output properties.

Fortunately, however, there is one aspect not emphasized in this analysis so far: these intersection terms consist of *half-spaces*, and moreover, each of the half-spaces therein is specified by a hyperplane chosen from among a single, common group of N hyperplanes. This will ultimately allow us to identify each non-empty intersection term in (20) with a *full, n-dimensional region* from this hyperplane arrangement, and by Theorem 1 in Section 2.4, there are at most $O(N^n/n!)$ such regions. Effectively, then, the *geometry* of

this hyperplane arrangement (with N hyperplanes in dimension n) prevents exponential growth in the number intersection terms relevant to the truth of (20): indeed, the polynomial growth, $O(N^n/n!)$, means that many of those intersection terms cannot correspond to valid regions in the arrangement[3].

In particular, consider the following set of affine functions, which in turn defines an arrangement of hyperplanes in $\mathbb{R}^n$:

$$l_i^a \triangleq \ell_i - a. \tag{21}$$

Let $\mathcal{H}_a \triangleq \cup_{i=1}^N \{H^0_{l_i^a}\}$ denote the corresponding arrangement. Now, consider any index $(i_1, \dots, i_M) \in s_1 \times \cdots \times s_M$ specifying an intersection term in (20), and suppose without loss of generality that $i_1, \dots, i_K$ are the only unique indices therein. Then using the notation H^{-1} introduced in Definition 6, it is straightforward to write:

$$\bigcap_{k=1,\dots,M} \{x : \ell_{i_k}(x) < a\} = \bigcap_{k=1}^{K} H^{-1}_{l^a_{i_k}}, \tag{22}$$

and where $l^a_{i_k}$ is as defined in (21). As a consequence, we conclude:

$$\bigcap_{k=1,\dots,M} \{x : \ell_{i_k}(x) < a\} \neq \emptyset$$

$$\Leftrightarrow \exists(\mathfrak{s}' : \{l^a_{i_k} : i = 1, \dots, K\} \to \{-1\}).\Big(\bigcap_{i=1}^{K} H^{\mathfrak{s}'(l^a_{i_k})}_{l^a_{i_k}} \neq \emptyset\Big). \tag{23}$$

Although it seems unnecessary to introduce the function $\mathfrak{s}'$, this notation directly connects (22) to full-dimensional regions of the arrangement $\mathcal{H}_a$. Indeed, it states that the intersection term of interest is non-empty if and only if there is a full-dimensional region in the hyperplane arrangement whose index function $\mathfrak{s} : \{l_i^a : i = 1, \dots N\} \to \{-1, +1\}$ agrees with one of the partial indexing functions $\mathfrak{s}'$ described in (23). *More simply, said intersection term is non-empty if and only if it contains a full-dimensional region from the arrangement $\mathcal{H}_a$; such a region can be said to "witness" the non-emptiness of the intersection term.* This is illustrated in Figure 2.

Formally, we have the following proposition.

Proposition 2. *Consider an instance of Problem 1B. Then that instance is **UNSAT** if and only if the set:*

$$P_X \cap \bigcup_{\substack{(i_1,\dots,i_M)\\ \in s_1\times\cdots\times s_M}} \bigcap_{k=1,\dots,M} \{x : \ell_{i_k}(x) < a\} \neq \emptyset. \tag{24}$$

And this is the case if and only if there exists an index $(i_1, \dots, i_M) \in s_1 \times \cdots \times s_M$ with distinct elements denoted by $\hat{i}_1, \dots, \hat{i}_K \in \{1, \dots, M\}$ such that the following holds:

- *there exists a full-dimensional region R in $\mathcal{H}_a$, specified by $\mathfrak{s}_R : \{l_i^a : i = 1, \dots, N\} \to \{-1, +1\}$, such that:*

$$\mathfrak{s}_R(l^a_{\hat{i}_k}) = -1 \text{ for all } k = 1, \dots, K \tag{25}$$

and

$$R \cap P_X \neq \emptyset. \tag{26}$$

*If such a region R exists, then it is said to **witness** the non-emptiness of the corresponding intersection term with the index $(i_1, \dots, i_M)$.*

Proof. The proof follows from the above manipulations. □

[3]Of course these results apply when n is fixed; see also [15], and the comments therein pertaining to NN verification encodings of 3-SAT problems [21].

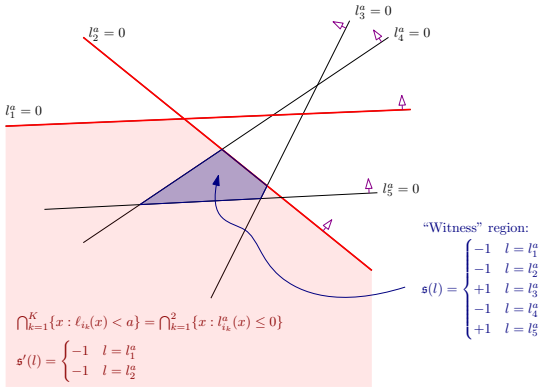

Figure 2: Identifying a non-empty intersection term from (20) using a "witness" region from the arrangement $\mathcal{H}_a$ (positive half-spaces are indicated with blue arrows). Intersection term and defining half-spaces shown in red; "witness" region from $\mathcal{H}_a$ shown in blue. Input constraints, P_X, omitted for clarity.

Proposition 2 establishes a crucial identification between full-dimensional regions in a hyperplane arrangement and the non-empty intersection terms in (20), a verification formula equivalent to the satisfiability of Problem 1B. However, it is still framed in terms of individual indices of the form $(i_1, \dots, i_M)$, which are too numerous to enumerate for reasons noted above. Thus, converting Proposition 2 into a practical and fast algorithm to solve Problem 1B entails one final step: efficiently evaluating a *full-dimensional* region in $\mathcal{H}_a$ to determine if it matches *any* index of the form form $(i_1, \dots, i_M) \in s_1 \times \cdots \times s_M$. This will finally lead to Fast BATLLNN's algorithm to verify an instance Problem 1B by enumerating the regions of $\mathcal{H}_a$ instead of enumerating all of the indices in $s_1 \times \cdots \times s_M$.

Predictably, Fast BATLLNN essentially takes a greedy approach to evaluating the indices in $s_1 \times \cdots \times s_M$. In particular, consider a full-dimensional region, $R \subset \mathbb{R}^n$, of the hyperplane arrangement $\mathcal{H}_a$, and suppose that R is specified by the index function (see Definition 8):

$$\mathfrak{s}_R : \{l_i^a | i = 1, \dots N\} \to \{-1, +1\}. \tag{27}$$

According to Proposition 2, $\mathfrak{s}_R$ will be a witness to a violation of Problem 1B if *each* one of the selector sets, s_j $j = 1, \dots, M$, contains at least one local linear function that can identified with one of R's negative hyperplanes (those assigned -1 by $\mathfrak{s}_R$). Thus, to establish whether $\mathfrak{s}_R$ corresponds to a non-empty intersection term, we can proceed *greedily* over the selector sets s_j: i.e., iterate over each selector set, s_j, checking all of the negative hyperplanes of R for membership therein. This iteration proceeds as long as *some* negative hyperplane of R is found to be an element of the current s_j. If all selector sets can be matched in this way, then the region R is a witness to a violation of Problem 1B as per Proposition 2. If, however, a selector set s_j is found to contain *no* negative hyperplanes of R, then the iteration terminates, since the region R cannot be a witness to a violation of Problem 1B. This greedy algorithm clearly eschews enumeration of $s_1 \times \cdots \times s_M$ in favor of enumerating the individual s_j's in sequence; as a result, it effectively finds the smallest intersection term to which the region R can be a witness.

```
input : a ∈ ℝ, a lower bound to verify
        Ξ_{N,M}, parameters of a TLL NN to verify
        ℒ_X = {l_{X,1}, ..., l_{X,N_X}}, affine functions
          specifying an input constraint polytope, P_X
output: Boolean (True = SAT; False = UNSAT)
1  function verifyScalarLB(a, Ξ_{N,M}, ℒ_X)
2      h_a ← [ℓ_i − a  for i = 1 ... N ]
3      for reg in Regions(h_a) do
4          if reg ∩P_X == ∅ then
5              continue reg
6          end
7          negHypers ← NegativeHyperplanes(reg)
8          for j = 1 ... M do
9              for i in negHypers do
10                 /* reg is on the negative side of ℓ_i − a   */
11                 if i ∈ s_j then
12                     continue j
13                 end
14             end
15             /* This region witnesses a violation; return.
                  (For this j, all negHypers tested without
                  triggering the continue on line 12.)       */
16             return False
17         end
18     end
19     return True
20 end
```

Algorithm 2: `verifyScalarLB`; i.e., solve Problem 1B

The pseudocode for this algorithm, with an outer loop iterating over regions of $\mathcal{H}_a$, appears as `verifyScalarLB` in Algorithm 2.

4.3 On the Complexity of Fast BATLLNN

Given the remarks prefacing equation (12), it suffices to consider the complexity of Proposition 1 and Problem 1B individually. To simplify the notation in this section, we denote the complexity of running a linear program with N constraints in n variables by $\mathsf{LP}(N, n)$. **Note also: we consider complexities for a fixed n.**

4.3.1 Complexity of Problem 1A. Analyzing the complexity of `verifyScalarUB` in Algorithm 1 is straightforward. There are M total min (or intersection) terms, and each of these requires: one LP to check for $\geq b$ feasibility (line 5 of Algorithm 1); followed by at most N LPs to find an interior point (line 8 of Algorithm 1). Thus, the complexity of `verifyScalarUB` is bounded by the following:

$$O(M \cdot N \cdot \mathsf{LP}(N + \mathsf{N}_X, n).) \tag{28}$$

4.3.2 Complexity of Problem 1B. Analyzing the runtime complexity of `verifyScalarLB` in Algorithm 2 is also more or less straightforward, given an algorithm that enumerates the regions of a hyperplane arrangement. Fast BATLLNN uses an algorithm very similar to the reverse search algorithm described in [4] and improved slightly in [17]. For a hyperplane arrangement consisting of N hyperplanes in dimension n, that reverse search algorithm has a per-region complexity bounded by:

$$O(N \cdot \mathsf{LP}(N, n)). \tag{29}$$

By Theorem 1 in Section 2, there are at most $O(N^n/n!)$ such regions.

Indeed, the per-region complexity of the loops in `verifyScalarLB` is easily seen to be bounded by $M \cdot N^2$ operations per region. Thus, it remains to evaluate the complexity of checking the intersection `reg` $\cap P_X$ == $\emptyset$ (see line 4 of Algorithm 2); however, this only appears as a separate operation for pedagogical simplicity. Fast BATLLNN actually follows the technique in [15] to achieve the same assertion: the hyperplanes describing P_X are added to the arrangement $\mathcal{H}_a$, and any region for which one of those hyperplanes satisfies $\mathfrak{s}(l_{X,i}) = +1$ is ignored. This can be done with the additional per-region complexity associated with the size of the larger arrangement, but without enumerating more than $O(N^n/n!)$ regions. Thus, the complexity of Algorithm 2 is bounded by:

$$\begin{aligned} &O\big(\, M \cdot N^2 \cdot (N + \mathsf{N}_X) \cdot \mathsf{LP}(N + \mathsf{N}_X, n)) \cdot (N^n/n!)\,\big) \\ &\qquad = O(M \cdot N^{n+3} \cdot \mathsf{LP}(N + \mathsf{N}_X, n)/n!). \end{aligned} \tag{30}$$

4.3.3 Complexity of Fast BATLLNN Compared to [15]. We begin by adapting the TLL verification complexity reported in [15, Theorem 3] to the scalar TLLs and single output properties of Problem 1A and Problem 1B. In the notation of this paper, it is as follows:

$$O(n \cdot M \cdot N^{2n+3} \cdot \mathsf{LP}(N^2 + \mathsf{N}_X, n)/n!). \tag{31}$$

It is immediately clear that Fast BATLLNN has a significant complexity advantage for either type of property. Even the more expensive `verifyScalarLB` has a runtime complexity of $\approx O(N^n)$ compared to $\approx O(N^{2n})$ for [15, Theorem 3], which also uses larger LPs.

5 IMPLEMENTATION

5.1 General Implementation

The core algorithms of Fast BATLLNN, Algorithm 1 and Algorithm 2, are amenable to considerable parallelism. Thus, in order to make Fast BATLLNN as fast as possible, its implementation is focused on *parallelism and concurrency* as much as possible.

With this in mind, Fast BATLLNN is implemented using a high-performance concurrency abstraction library for Python called `charm4py` [1]. `charm4py` uses an actor model to facilitate concurrent programming, and it provides a number of helpful features to achieve good performance with relatively little programming effort. For example, it employs a cooperative scheduler to eliminate race-conditions, and it transparently offers the standard Python pass-by-reference semantics for function calls on the same Processing Element (PE). Moreover, it can be compiled to run on top of Message Passing Interface (MPI), which allows a single code base to scale from an individual multi-core computer to a multi-computer cluster. Fast BATLLNN was written with the intention of being deployed this way: it offers flexibility in how its core algorithms are assigned PEs, so as to take better take advantage of both compute and memory resources that are spread across multiple computers.

5.2 Implementation Details for Algorithm 2

Between the two core algorithms of Fast BATLLNN, Algorithm 2 is the more challenging to parallelize. Indeed, Algorithm 1 has a trivial parallel implementation, since it consists primarily of a

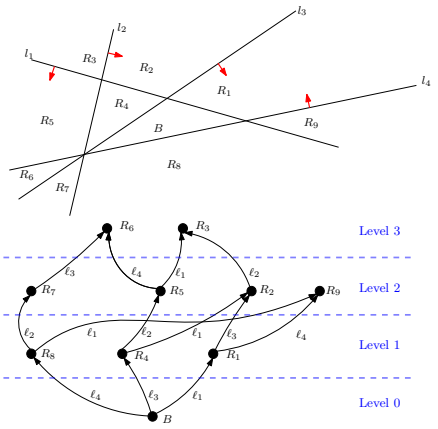

Figure 3: (Top) A hyperplane arrangement $\{H^0_{l_1}, \ldots, H^0_{l_4}\}$ with positive half spaces denoted by red arrows and regions $B, R_1, \ldots, R_9$. (Bottom) The corresponding adjacency poset.

for loop over a known index set. In Algorithm 2, it is the for loop over *regions of a hyperplane* that makes parallelization non-trivial. Hence, this section describes how Fast BATLLNN parallelizes the region enumeration of a hyperplane arrangement for Algorithm 2.

To describe the architecture of Fast BATLLNN's implementation of hyperplane region enumeration, we first briefly introduce the well-known Reverse Search (RS) algorithm for the same task [17], and the algorithm on which Fast BATLLNN's implementation is loosely based. As its name suggests, it is a *search* algorithm: it starts from a known base region of the arrangement and searches for regions adjacent to it, and then regions adjacent to those, and so on. In particular, though, RS implements a "minimum index" rule to effectively identify each region with a unique path to the base region; this ensures that regions are not visited multiple times [4, 17]. RS has the benefit that it is memory efficient, since it tracks the current state of the search using only the memory required to store the current adjacency indices; i.e., the information required to back-track is effectively computed rather than being stored [17, pp. 10]. However, this memory efficiency clearly comes at the expense of having to compute the index for each region. Moreover, the minimum index computation can also be seen as a synchronization cost for a parallel implementation, since it allows multiple workers to proceed without traversing the same region twice.

A natural way to avoid the index computation is to allow multiple concurrent search workers, but have them enter their independent search results into a common, synchronized hash table instead. Assuming an amortized $O(1)$ hash complexity, this solution eliminates the index computation without affecting the overall complexity; on the other hand, it comes with a steep memory penalty, since it requires storing all $O(N^n)$ regions in the worst case. However, there is a way to efficiently enable and coordinate multiple search processes, while avoiding this excessive memory requirement.

To this end, Fast BATLLNN leverages a special property of the region adjacency structure in a hyperplane arrangement. In particular, the regions of a hyperplane arrangement can be organized into a *leveled* adjacency poset [11]. That is, relative to any initial base region, all of the regions in the arrangement can be grouped according to the number of hyperplanes that were "crossed" in the process discovering them; the same idea is also implicit in [4, 17]. This leveled property of the adjacency poset is illustrated in Figure 3: the top pane shows a hyperplane arrangement with its regions labeled; the bottom pane depicts the region adjacency poset for this arrangement, with levels indicated relative to a base region, B. For example, a search starting from B will find region R_1 by crossing *only* l_1 and region R_2 by crossing l_1 and l_3.

Thus, Fast BATLLNN still approaches the region enumeration problem as search, but instead it proceeds level-wise. All of the regions in the *current* level can be easily divided among the available processing elements, which then search in parallel for their immediately adjacent regions; the result of this search is a list of regions comprising the entire *next* level in the adjacency poset, which then becomes the current level and the process repeats. From an implementation standpoint, searching the region adjacency structure level-wise offers a useful way of reducing Fast BATLLNN's memory footprint. In particular, once a level is fully explored, the regions it contains *will never be seen again*. Thus, Fast BATLLNN need only maintain a hash of regions from one level at a time: the hash tables from previous levels can be safely discarded. *In this way, Fast BATLLNN achieves a parallel region search but without resorting to hashing the entire list of discovered regions.*

Finally, we note that a search-type algorithm for region enumeration has a further advantage for solving a problem like Problem 1B. In particular, a search algorithm reveals each new region with a relatively low computational cost — see (29); this is in contrast to some other enumeration algorithms, which must *run to completion before even* ***one region*** *is available*. This is a considerable advantage for Algorithm 2, which can terminate on the first violating region found (if the problem is **UNSAT**): a violating region may be found early in the search, and thus at relatively low computational cost.

6 EXPERIMENTS

We conducted a series of experiments to evaluate the performance and scalability of Fast BATLLNN as a TLL verifier, both in its own right and relative to general NN verifiers applied to TLL NNs. In particular, we conducted the following three experiments:

Exp. 1) Scalability of Fast BATLLNN as a function of TLL input dimension, n; the number of local linear functions, N, and the number of selector sets, M, remained fixed.

Exp. 2) Scalability of Fast BATLLNN as a function of the number of local linear functions, N, with $N = M$; the input dimension, n, remained fixed.

Exp. 3) Comparison with general NN verifiers: nnenum [5], PeregriNN [23] and α, β-Crown [32].

All experiments were run in a VMWare Workstation Pro virtual machine (VM) on a Linux host with 48 hyperthreaded cores and 256 GB of RAM. This VM had 64 GB of RAM but a core count specific to each experiment. A timeout of 300 seconds was used in all cases.

6.1 Experimental Setup: Networks and Properties

6.1.1 TLL NNs Verified. Given that Problem 1 can be decomposed into instances of Problem 1A and Problem 1B, all of these experiments were conducted on scalar-output TLL NNs using real-valued properties of the form in either Problem 1A or Problem 1B.

In Experiments 1 and 2, TLL NNs of the desired n, N and M were generated randomly according to the following procedure, which was designed to ensure that they are unlikely to be degenerate on (roughly) the input set $[-1, 1]^n$. The procedure is as follows:

(1) Randomly generate elements of W_ℓ and b_ℓ according to normal distributions of mean zero and standard deviations of $1/10$ and 1, respectively.
(2) Randomly generate selector sets, s_j, by generating random integers between 0 and $2^{N+1} - 1$, and continue generating them by this mechanism until M are obtained such that no two selector sets satisfy $s_j \subseteq s_{j'}$ (a form of degeneracy).
(3) For each corresponding selector matrix, S_j, solve M instances of the following least squares problem:

$$\min_{x_j \in \mathbb{R}^n} \|S_j W_\ell x_j - S_j b_\ell\|_2, \tag{32}$$

to obtain the M vectors, $x_1, \ldots, x_M$.
(4) Then scale each row of W_ℓ (from (1) above) by the corresponding row of the vector:

$$\left[\max_{j=1\ldots M} \left|[\![x_j]\!]_{[1,1]}\right| \ \ldots \ \max_{j=1\ldots M} \left|[\![x_j]\!]_{[N,1]}\right| \right]^{\mathrm{T}}. \tag{33}$$

This has the (qualitative) effect of forcing the intersections of the local linear functions to be concentrated near the origin.

In Experiment 3, we obtained and used the scalar TLL NNs that were tested in [15]. These networks all have $n = 2$ and $N = M$; there are thirty examples for each of the sizes $N = M = 8, 16, 24, 32, 40, 48, 56$ and 64 (each size has a common neuron count, ranging from 256 neurons for $N = 8$ to 16384 neurons for $N = 64$). We used these networks in particular so as to enable some basis of comparison with the experimental results in [15]. This is relevant, since that tool is not publicly available, and hence omitted from our comparison. *Note: we considered these networks with different, albeit similar, properties to those used in [15]; see Sections 6.1.2-6.1.3 below.*

6.1.2 Input Constraints, P_X. In all experiments, we considered verification problems with $P_X = [-2, 2]^n$. For the TLLs we generated, there is no great loss of generality in considering this fixed size for P_X, since we generated them to be "interesting" in this vicinity; see Section 6.1.1. However, using a hyper-rectangle P_X was *necessary* for Experiment 3, since some of the general NN verifiers accept only hyper-rectangle input constraints. Thus, we made the universal choice $P_X = [-2, 2]^n$ for consistency between experiments.

Note, however, that [15] verified general polytopic input constraints on the networks we borrowed for Experiment 3. Nevertheless, we expect the results for Fast BATLLNN in Experiment 3 to be somewhat comparable to the results in [15], since all of those polytopic constraints are *contained* in the box, $P_X = [-2, 2]^2$.

6.1.3 Output Properties Verified. For a scalar TLL, only two parameters are required to specify an output property: a real-valued scalar and the direction of the inequality. In all cases, the direction of the inequality was determined by the outcome of Bernoulli random variable. And in all cases except one (noted in Experiment 2), the random real-valued property was generated by the following procedure. First, the TLL network was evaluated at 10,000 samples collected from the set P_X; any property between the min and max of these output samples is guaranteed to be **UNSAT**. Then, to get a mixture of **SAT**/**UNSAT** properties, we select a random property from this interval symmetrically extended to twice its original size.

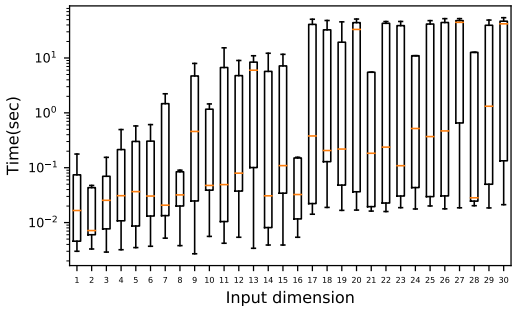

Figure 4: Experiment 1. Scaling the Input Dimension

6.2 Experiment 1: Input Dimension Scalability

In this experiment, we evaluated the scalability of Fast BATLLNN as a function of input dimension of the TLL to be verified. To that end, we generated a suite of TLL NNs with input sizes varying from $n = 1$ to $n = 30$, using the procedure described in Section 6.1.1. We generated one instance for each size, and all TLLs had $N = M = 64$ constant. We then verified each of these TLLs with respect to its own set of 20 randomly generated properties (described in Sections 6.1.2-6.1.3). In this experiment, Fast BATLLNN was run in a 32 core VM.

Figure 4 summarizes the results of this experiment with a box-and-whisker plot of verification times: each box-and-whisker[4] summarizes the verification times for the twenty networks of the corresponding input dimension; no properties/networks resulted in a timeout. The data in the figure shows a clear trend of increasing median, as expected for progressively harder problems (recall the runtime complexities indicated in Section 4.3). By contrast, note that the minimum and maximum runtimes grow very slowly with dimension: given the complexity analysis of Section 4.3, we speculate that these results are likely due to the characteristics of the generated TLLs. That is, the generation procedure appears to "saturate" in the sense that it eventually produces networks which require, on average, a constant number of loop iterations to verify.

6.3 Experiment 2: Network Size Scalability

In this experiment, we evaluated the scalability of Fast BATLLNN as a function of the number of local linear functions, N, in the TLL to be verified. To that end, we generated a suite of of TLL NNs local linear functions ranging in number from $N = 16$ to $N = 512$, again using the procedure described in Section 6.1.1. We generated one instance for each value of N, and all TLLs had $M = N$ and $n = 15$. We then verified each TLL with respect to its own set of 20 randomly generated properties. The properties for sizes $N = 16$ through $N = 256$ were generated as described in Sections 6.1.2-6.1.3; however, our TensorFlow implementation used too much memory to generate samples for the $N = 512$ TLLs, so the properties for these networks were generated using the bounds for the $N = 256$ TLLs. In this experiment, Fast BATLLNN was run in a 32 core VM.

[4] As usual, the boxes denote the first and third quartiles; the orange horizontal line denotes the median; and the whiskers show the maximum and minimum.

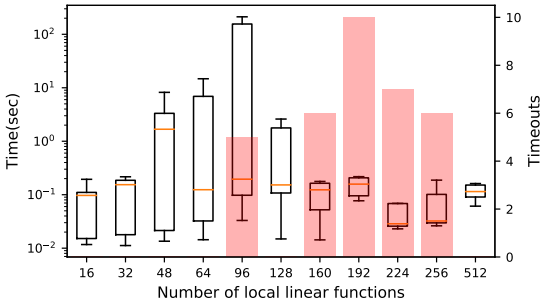

Figure 5: Experiment 2. Scaling # of Local Linear Functions

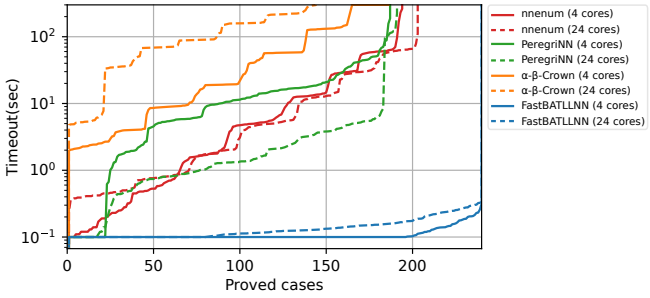

Figure 6: Experiment 3. General NN Verifiers

Figure 5 summarizes the results of this experiment with a box-and-whisker plot of verification times: each box-and-whisker summarizes the verification times for the twenty test cases of the corresponding size, much as in Section 6.2. However, since some verification problems timed out in this experiment, those time outs were excluded from the box-and-whisker; they are instead indicated by a superimposed bar graph, which displays a count of the number of timeouts obtained from each group of equally-sized TLLs. The data in this figure shows the expected trend of increasingly difficult verification as N increases; this is especially captured by the trend of experiencing more timeouts for larger networks. The outlier to this trend is the size $N = 512$, but this is most likely due to different method of generating properties for these networks (see above). Finally, note that the minimum verification times exhibit a much slower growth trend, as in Experiment 1.

6.4 Experiment 3: General NN Verifiers

In this experiment, we compared the verification performance of Fast BATLLNN with state-of-the-art (SOTA) NN verifiers designed to work on general deep NNs. For this experiment, we compared against generic verifiers α, β-Crown [32], nnenum [5] and PeregriNN [23] as a representative sample of SOTA NN verifiers. Moreover, we conducted this experiment on the same 240 networks used in [15], and described in Section 6.1.1; this further facilitates a limited comparison with that algorithm, subject to the caveats described in Section 6.1.1 and Sections 6.1.2-6.1.3.

In order to make this test suite of TLLs available to the generic verifiers, each network was first implemented as a TensorFlow model using a custom implementation tool. The intent was to export these TensorFlow models to the ONNX format, which each of the generic verifiers can read. However, most of the generic verifiers do not support implementing multiple feed-forward paths by tensor reshaping operations, as in the most straightforward implementation of a TLL; see Figure 1 and Section 2.3. Thus, we had to first "flatten" our TensorFlow implementation into an equivalent network where each min network accepts the outputs of *all* of the selector matrices, only to null the irrelevant ones with additional kernel zeros in the first layer. This is highly sub-optimal, since it results in neurons receiving many more inputs than are really required. However, we could not devise another method to circumvent this limitation present in most of the tools.

With all of the tools able to read the same NNs in our (borrowed) test suite, we randomly generated verification properties for each of the networks, as in the previous experiments. However, recall that the generic NN verifiers have a slightly different interpretation of properties compared with Fast BATLLNN. For scalar-output networks, this amounts to verifying the properties in Problem 1A and Problem 1B with the same interpretations, but with *non-strict* inequalities instead of *strict* inequalities. Since this will only generate divergent results when a property happens to be exactly equal to the maximum or minimum value on P_X, we elide this issue.

Thus, we ran each of the tools, Fast BATLLNN, α, β-Crown, nnenum and PeregriNN on this test suite of TLLs and properties. PeregriNN was configured with `SPLIT_RES=0.1`; nnenum was configured with `TRY_QUICK_APPROX=True` and all other parameters set to default values; and α, β-Crown was configured with `input space splitting`, `share_slopes=True`, `lr_alpha=0.01`, `no_solve_slopes=True` and `branching_method=sb`. All solvers used `float64` computations. Furthermore, we ran two versions of this experiment, one with a 4 core VM and one with a 24 core VM.

Figure 6 summarizes the results of this experiment with a cactus plot: a point on any one of the curves indicates the timeout that would be required to obtain the corresponding number of proved cases for that tool (from among of the test suite described above). As noted, each tool was run separately in two VMs, one with 4 cores and one with 24 cores; thus, each tool has two curves in Figure 6. The data shows that Fast BATLLNN is on average 960×/435× faster than nnenum, 1800×/1370× faster than α, β-Crown, and 1000×/500× faster than PeregriNN using 4 and 24 cores respectively. Fast BATLLNN proved all 240 properties in just 17 seconds (4 cores), whereas nnenum proved 193, α, β-Crown proved 153, and PeregriNN proved 186. Note that unlike the other tools, Fast BATLLNN doesn't exhibit exponential growth in execution time, which is consistent with the complexity analysis in Section 4.3. Despite the caveats noted above, Fast BATLLNN also compares favorably with the execution times shown in [15, Figure 1(b)], which end up in the 100's or 1000's of seconds for $N = 64$. Finally, note that Fast BATLLNN exhibited slightly worse overall performance with 24 cores. However, timeouts increased at a significantly slower *rate* with 24 cores than with 4 cores. This is best explained by setup overhead associated with using the additional cores—and that this overhead is quickly offset by the extra parallelism. α, β-Crown similarly suffered worse performance with more cores, since it apparently benefits from multiple cores only in MIP problems.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant numbers #2002405 and #2013824.

REFERENCES

[1] [n. d.]. *Charm4py*. https://github.com/UIUC-PPL/charm4py

[2] [n. d.]. International Verification of Neural Networks Competition 2020 (VNN-COMP'20). https://sites.google.com/view/vnn20.

[3] Ross Anderson, Joey Huchette, Will Ma, Christian Tjandraatmadja, and Juan Pablo Vielma. 2020. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming* (2020), 1–37. https://doi.org/10.1007/s10107-020-01474-5

[4] David Avis and Komei Fukuda. 1996. Reverse Search for Enumeration. *Discrete Applied Mathematics* 65, 1 (1996), 21–46. https://doi.org/10.1016/0166-218X(95)00026-N

[5] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. 2020. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Lecture Notes in Computer Science, Vol. 12224. Springer International Publishing, 66–96. https://doi.org/10.1007/978-3-030-53288-8_4

[6] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. 2016. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 2613–2621.

[7] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, P Kohli, P Torr, and P Mudigonda. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21, 2020 (2020).

[8] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, Narayan Kumar K. D'Souza D. (Ed.), Vol. 10482. Springer, 251–268. https://doi.org/10.1007/978-3-319-68167-2_18

[9] Ulices Santa Cruz, James Ferlez, and Yasser Shoukry. 2021. *Safe-by-Repair: A Convex Optimization Approach for Repairing Unsafe Two-Level Lattice Neural Network Controllers*. https://doi.org/10.48550/arXiv.2104.02788 arXiv:2104.02788 [cs, eess, math]

[10] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. 2018. A Dual Approach to Scalable Verification of Deep Networks.. In *Proceedings of the Thirty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-18)*, Vol. 1. AUAI Press, 2.

[11] Paul H. Edelman. 1984. A Partial Order on the Regions of R^n Dissected by Hyperplanes. *Trans. Amer. Math. Soc.* 283, 2 (1984), 617–631. https://doi.org/10.2307/1999150

[12] Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, Narayan Kumar K. D'Souza D. (Ed.), Vol. 10482. Springer, 269–286. https://doi.org/10.1007/978-3-319-68167-2_19

[13] Mahyar Fazlyab, Alexander Robey, Hamed Hassani, Manfred Morari, and George Pappas. 2019. Efficient and accurate estimation of lipschitz constants for deep neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 11423–11434.

[14] James Ferlez and Yasser Shoukry. 2020. AReN: Assured ReLU NN Architecture for Model Predictive Control of LTI Systems. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3365365.3382213

[15] James Ferlez and Yasser Shoukry. 2021. Bounding the Complexity of Formally Verifying Neural Networks: A Geometric Approach. In *2021 60th IEEE Conference on Decision and Control (CDC)* (2020-12-21). 5104–5109. https://doi.org/10.1109/CDC45484.2021.9683375

[16] James Ferlez, Xiaowu Sun, and Yasser Shoukry. 2020. Two-Level Lattice Neural Network Architectures for Control of Nonlinear Systems. In *2020 59th IEEE Conference on Decision and Control (CDC)*. 2198–2203. https://doi.org/10.1109/CDC42340.2020.9304079

[17] J.-A. Ferrez, K. Fukuda, and Th M. Liebling. 2001. *Cuts, Zonotopes and Arrangements*. Infoscience. http://infoscience.epfl.ch/record/77413

[18] Matteo Fischetti and Jason Jo. 2018. Deep neural networks and mixed integer linear optimization. *Constraints* 23, 3 (2018), 296–309. https://doi.org/10.1007/s10601-018-9285-6

[19] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18. https://doi.org/10.1109/SP.2018.00058

[20] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 2019. 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*. 169–178. https://doi.org/10.1145/3302504.3311806

[21] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification* (Cham, 2017) *(Lecture Notes in Computer Science)*. Springer International, 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[22] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11561. Springer, 443–452. https://doi.org/10.1007/978-3-030-25540-4_26

[23] Haitham Khedr, James Ferlez, and Yasser Shoukry. 2021. PEREGRiNN: Penalized-Relaxation Greedy Neural Network Verifier. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). 287–300. https://doi.org/10.1007/978-3-030-81685-8_13

[24] Alessio Lomuscio and Lalit Maganti. 2017. *An approach to reachability analysis for feed-forward relu neural networks*. https://doi.org/10.48550/arXiv.1706.07351 arXiv:1706.07351 [cs]

[25] Richard P Stanley. 2007. *Geometric Combinatorics*. IAS/Park City Mathematics Series, Vol. 13. American Mathematical Society, Chapter An introduction to hyperplane arrangements. https://doi.org/10.1090/pcms/013

[26] J. M. Tarela and M. V. Martínez. 1999. Region Configurations for Realizability of Lattice Piecewise-Linear Models. *Mathematical and Computer Modeling* 30, 11 (1999), 17–27. https://doi.org/10.1016/S0895-7177(99)00195-8

[27] Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2019. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*.

[28] H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L.-V. Nguyen, W. Xiang, and T. Johnson. 2019. Star-Based Reachability Analysis of Deep Neural Networks. In *Formal Methods – The Next 30 Years* (Cham, 2019) *(Lecture Notes in Computer Science)*. Springer International. https://doi.org/10.1007/978-3-030-30942-8_39

[29] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. 2020. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems. In *Computer Aided Verification* (Cham, 2020) *(Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, 3–17. https://doi.org/10.1007/978-3-030-53288-8_1

[30] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient formal safety analysis of neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 6367–6377.

[31] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, USA, 1599–1614.

[32] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2021. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Verification. In *Advances in Neural Information Processing Systems*, A. Beygelzimer and Y. Dauphin and P. Liang and J. Wortman Vaughan (Ed.).

[33] Yuh-Shyang Wang, Lily Weng, and Luca Daniel. 2020. Neural Network Control Policy Verification With Persistent Adversarial Perturbation. In *International Conference on Machine Learning* (2020-11-21), III, Hal Daumé and Singh, Aarti (Ed.), Vol. 119. PMLR, 10050–10059. https://proceedings.mlr.press/v119/wang20v.html

[34] Eric Wong and J Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proceedings of the 35th International Conference on Machine Learning*, Dy, Jennifer and Krause, Andreas (Ed.), Vol. 80. 5286–5295.

[35] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. 2017. *Reachable set computation and safety verification for neural networks with ReLU activations*. https://doi.org/10.48550/arXiv.1712.08163 arXiv:1712.08163 [cs]

[36] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. 2018. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems* 29, 11 (2018), 5777–5783. https://doi.org/10.1109/TNNLS.2018.2808470