

# Output Range Analysis for Deep Feedforward Neural Networks

Souradeep Dutta<sup>1</sup>, Susmit Jha<sup>2</sup>, Sriram Sankaranarayanan<sup>1</sup> and Ashish Tiwari<sup>2</sup>.

1. University of Colorado, Boulder, USA.

2. SRI International, Menlo Park, USA.

{souradeep.dutta,sriram.sankaranarayanan}@colorado.edu,  
{tiwari,susmit.jha}@csl.sri.com

**Abstract.** Given a neural network (NN) and a set of possible inputs to the network described by polyhedral constraints, we aim to compute a safe over-approximation of the set of possible output values. This operation is a fundamental primitive enabling the formal analysis of neural networks that are extensively used in a variety of machine learning tasks such as perception and control of autonomous systems. Increasingly, they are deployed in high-assurance applications, leading to a compelling use case for formal verification approaches. In this paper, we present an efficient range estimation algorithm that iterates between an expensive global combinatorial search using mixed-integer linear programming problems, and a relatively inexpensive local optimization that repeatedly seeks a local optimum of the function represented by the NN. We implement our approach and compare it with Reluplex, a recently proposed solver for deep neural networks. We demonstrate applications of our approach to computing flowpipes for neural network-based feedback controllers. We show that the use of local search in conjunction with mixed-integer linear programming solvers effectively reduces the combinatorial search over possible combinations of active neurons in the network by pruning away suboptimal nodes.

## 1 Introduction

Deep neural networks have emerged as a versatile and popular representation for machine learning models. This is due to their ability to approximate complex functions, as well as the availability of efficient methods for learning these from large data sets. The black box nature of NN models and the absence of effective methods for their analysis has confined their use in systems with low integrity requirements. However, more recently, deep NNs are also being adopted in high-assurance systems, such as automated control and perception pipeline of autonomous vehicles [13] or aircraft collision avoidance [12]. While traditional system design approaches include rigorous system verification and analysis techniques to ensure the correctness of systems deployed in safety-critical applications [1], the inclusion of complex machine learning models in the form of deep NNs has created a new challenge to verify these models. In this paper, we focus on the *range estimation problem*, wherein, given a neural network  $N$  and a polyhedron  $\phi(\mathbf{x})$  representing a set of inputs to the network, we wish to estimate a range, denoted as  $\text{range}(l_i, \phi)$ , for each of the network's output  $l_i$  that subsumes all

possible outputs and is tight within a given tolerance  $\delta$ . We restrict our attention to feed-forward deep NNs. While we focus on NNs that use rectified linear units (ReLUs) [17] as activation functions, we also discuss extensions to other activation functions through piecewise linear approximations.

Our approach is based on augmenting a mixed-integer linear programming (MILP) solver. First of all, we use a sound piecewise linearization of the nonlinear activation function to define an encoding of the neural network semantics into mixed-integer constraints involving real-valued variables and binary variables that arise from the (piecewise) linearized activation functions. The encoding into MILP is a standard approach to handling piecewise linear functions [28]. As such, the input constraints  $\phi(\mathbf{x})$  are added to the MILP and next, the output variable is separately maximized and minimized to infer a range. Our approach combines the MILP solver with a local search that exploits the local continuity and differentiability properties of the function represented by the network. These properties are not implicit in the MILP encoding that typically relies on a branch-and-cut approach to solve the problem at hand. On the other hand, local search alone may get “stuck” in local minima. Our approach handles local minima by using the MILP solver to search for a solution that is “better” than the current local minimum or conclude that no such solution exists. Thus, by alternating between inexpensive local search iterations and relatively expensive MILP solver calls, we seek an approach that can exploit local properties of the neural network function but at the same time avoid the problem of local minima.

The range estimation problem has several applications. For instance, a safety focused application of the range estimation problem arises when we have deep neural networks implementing a controller. In this case, the range estimation problem enables us to prove bounds on the output of the NN controller. This is important because out-of-bounds outputs can drive the physical system into undesirable configurations, such as the locking of robotic arm, or command a car’s throttle beyond its rated limits. Finding these errors through verification will enable design-time detection of potential failures instead of relying on runtime monitoring which can have significant overhead and also may not allow graceful recovery. Additionally, range analysis can be useful in proving the safety of a closed loop system by integrating the action of a neural network controller with that of a plant model. In this paper, we focus on the application of range estimation problem to proving safety of several neural network plant models along with neural network feedback controllers. Other applications include proving the robustness of classifiers by showing that all possible input perturbations within some range do not change the output classification of the network.

**Related Work** The importance of analytical certification methods for neural networks has been well-recognized in literature. Neural networks have been observed to be very sensitive to slight perturbations in their inputs producing incorrect outputs [26, 21]. This creates a pressing need for techniques to provide formal guarantees on the neural networks. The verification of neural networks is a hard problem, and even proving simple properties about them is known to be NP-complete [14]. The complexity of verifying neural networks arises primarily from two sources: the nonlinear activation functions used in the network as elementary neural units and the structural complexity that can

be measured using depth and size of the network. Kurd [16] presented one of the first categorization of verification goals for NNs used in safety-critical applications. The proposed approach here targets a subset of these goals, G4 and G5, which aim at ensuring robustness of NNs to disturbances in inputs, and ensuring the output of NNs are not hazardous.

Recently, there has been a surge of interest in formal verification tools for neural networks [14, 10, 23, 22, 30, 31, 8, 25, 18]. A detailed discussion of these approaches to neural networks with piecewise linear activation functions, and empirical evaluations over benchmark networks has been carried out by Bunel et al [5]. Our approach relies on a piecewise linearization of the nonlinear activation function. This idea has been studied in the past, notably by Pulina et al [22, 23]. The key differences include: (a) our approach do not perform a refinement operation. As such, no refinement is needed for networks with piecewise linear activation functions, since the activation functions are encoded precisely. For other kinds of functions such as sigmoid or tanh, a refinement may be needed to improve the inferred ranges, but is not considered in our work. (b) We do not rely on existing Satisfiability-Modulo Theory (SMT) solvers [2]. Instead, our approach uses a mixed-integer linear programming (MILP) solver in combination with a local search. Recently, Lomuscio and Maganti present an approach that encodes neural networks into MILP constraints [18]. A similar encoding is also presented by Tjeng and Tedrake [27] for verifying robustness of neural network classifiers under a class of perturbations. These encodings are similar to ours. The optimization problems are solved directly using an off-the-shelf MILP solver [28, 4]. Additionally, our approach augments the MILP solver with a local search scheme. We note that the use of local search can potentially speed up our approach, since neural networks represent continuous, piecewise-differentiable functions. On the flip side, these functions may have a large number of local minima/maxima. Nevertheless, depending on the network, the function it approximates and the input range, the local search used in conjunction with a MILP solver can yield rapid improvements to the objective function.

Augmenting existing LP solvers has been at the center of two recent approaches to the problem. The Reluplex approach by Katz et al focuses on ReLU feed-forward networks [14]. Their work augments the Simplex algorithm with special functions and rules that handle the constraints involving ReLU activation functions. The linear programming used for comparison in Reluplex performs significantly less efficiently according to the experiments reported in this paper [14]. Note, however, that the scenarios used by Katz et al. are different from those studied here, and were not publicly available for comparison at the time of writing. Ehlers augments a LP solver with a SAT solver that maintains partial assignments to decide the linear region for each individual neuron. The solver is instantiated using facts inferred from a convexification of the activation function [8], much in the style of conflict clauses and lemmas used by SAT solvers. In fact, many ideas used by Ehlers can be potentially used to complement our approach in the form of cuts that are specific to neural networks. Such specialized cuts are very commonly used in MILP solvers.

A related goal of finding adversarial inputs for deep NNs has received a lot of attention, and can be viewed as a testing approach to NNs instead of verification method discussed in this paper. A linear programming based approach for finding adversarial

inputs is presented in [3]. A related approach for finding adversarial inputs using SMT solvers that relies on a layer-by-layer analysis is presented in [10]. Simulation-based approaches [30] for neural network verification have also been proposed in literature. This relies on turning the reachable set estimation problem into a neural network maximal sensitivity computation, and solving it using a sequence of convex optimization problems. In contrast, our proposed approach combines numerical gradient-based optimization with mixed-integer linear programming for more efficient verification.

**Contributions** We present a novel algorithm for propagating convex polyhedral inputs through a feedforward deep neural network with ReLU activation units to establish ranges for the outputs of the network. We have implemented our approach in a tool called SHERLOCK [6]. We compare SHERLOCK with a recently proposed deep NN verification engine - Reluplex [14]. We demonstrate the application of SHERLOCK to establish output range of deep NN controllers. Our approach seems to scale *consistently* to neural networks having 100 neurons to as many as over 6000 neurons.

## 2 Preliminaries

We present the preliminary notions including deep neural networks, polyhedra, and mixed integer linear programs.

We will study feed forward neural networks (NN) throughout this paper with  $n > 0$  inputs and  $m > 0$  outputs. For simplicity, we will present our techniques primarily for the single output case ( $m = 1$ ), explaining how they can be extended to networks with multiple outputs.

Let  $\mathbf{x} \in \mathbb{R}^n$  denote the inputs and  $y \in \mathbb{R}$  be the output of the network. Structurally, a NN  $\mathcal{N}$  consists of  $k > 0$  hidden layers, wherein we assume that each layer has the same number of neurons  $N > 0$ . We use  $N_{ij}$  to denote the  $j^{th}$  neuron of the  $i^{th}$  layer for  $j \in \{1, \dots, N\}$  and  $i \in \{1, \dots, k\}$ .

**Definition 1 (Neural Network).** A  $k$  layer neural network with  $N$  neurons per hidden layer is described by matrices:  $(W_0, \mathbf{b}_0), \dots, (W_{k-1}, \mathbf{b}_{k-1}), (W_k, \mathbf{b}_k)$ , wherein (a)  $W_0, \mathbf{b}_0$  are  $N \times n$  and  $N \times 1$  matrices denoting the weights connecting the inputs to the first hidden layer; (b)  $W_i, \mathbf{b}_i$  for  $i \in [1, k-1]$  connect layer  $i$  to layer  $i+1$  and (c)  $W_k, \mathbf{b}_k$  connect the last layer  $k$  to the output.

Each neuron is defined using its *activation function*  $\sigma$  linking its input value to the output value. Although this can be any function, there are a few common activation functions:

1. **ReLU:** The ReLU unit is defined by the activation function  $\sigma(z) : \max(z, 0)$ .
2. **Sigmoid:** The sigmoid unit is defined by the activation function  $\sigma(z) : \frac{1}{1+e^{-z}}$ .
3. **Tanh:** The activation function for this unit is  $\sigma(z) : \tanh(z)$ .

Figure 1 shows these functions graphically. We will assume that all the neurons of the network  $\mathcal{N}$  have the same activation function  $\sigma$ . Furthermore, we assume that  $\sigma$  is a continuous function and differentiable almost everywhere.

Given a neural network  $\mathcal{N}$  as described above, the function  $F : \mathbb{R}^n \rightarrow \mathbb{R}$  computed by the neural network is given by the composition  $F := F_k \circ \dots \circ F_0$  wherein  $F_i(\mathbf{z}) :$

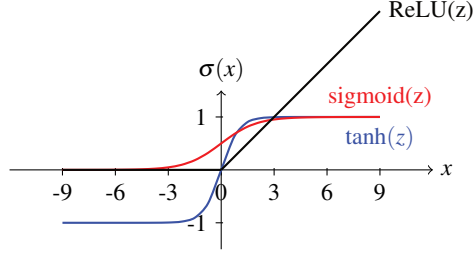


Fig. 1: Activation functions commonly used in neural networks.

$\sigma(W_i \mathbf{z} + \mathbf{b}_i)$  is the function computed by the  $i^{\text{th}}$  hidden layer,  $F_0$  the function linking the inputs to the first layer, and  $F_k$  linking the last layer to the output.

For a fixed input  $\mathbf{x}$ , it is easily seen that the function  $F$  computed by a NN  $\mathcal{N}$  is continuous and nonlinear, due to the activation function  $\sigma$ . For the case of neural networks with ReLU units, this function is piecewise affine, and differentiable almost everywhere in  $\mathbb{R}^n$ . For smooth activation functions such as tanh and sigmoid, the function is differentiable as well. If it exists, we denote the gradient of this function  $\nabla F : (\partial_{x_1} F, \dots, \partial_{x_n} F)$ . Computing the gradient can be performed efficiently (as described subsequently).

## 2.1 Mixed Integer Linear Programs

Throughout this paper, we will formulate linear optimization problems with integer variables. We briefly recall these optimization problems, their computational complexity and solution techniques used in practice.

**Definition 2 (Mixed Integer Program).** A mixed integer linear program (MILP) involves a set of real-valued variables  $\mathbf{x}$  and integer valued variables  $\mathbf{w}$  of the following form:

$$\begin{aligned} \max \quad & \mathbf{a}^T \mathbf{x} + \mathbf{b}^T \mathbf{w} \\ \text{s.t.} \quad & A\mathbf{x} + B\mathbf{w} \leq \mathbf{c} \\ & \mathbf{x} \in \mathbb{R}^n, \mathbf{w} \in \mathbb{Z}^m \end{aligned}$$

The problem is called a linear program (LP) if there are no integer variables  $\mathbf{w}$ . The special case wherein  $\mathbf{w} \in \{0, 1\}^m$  is called a binary MILP. Finally, the case without an explicit objective function is called an MILP feasibility problem.

It is well known that MILPs are NP-hard problems: the best known algorithms, thus far, have exponential time worst case complexity. We will later briefly review the popular branch-and-cut class of algorithms for solving MILPs at a high level. These algorithms along with the associated heuristics underlie highly successful, commercial MILP solvers such as Gurobi [9] and CPLEX [11].

### 3 Problem Definition and MILP Encoding

Let  $\mathcal{N}$  be a neural network with inputs  $\mathbf{x} \in \mathbb{R}^n$ , output  $y \in \mathbb{R}$  and weights  $(W_0, \mathbf{b}_0), \dots, (W_k, \mathbf{b}_k)$ , activation function  $\sigma$  for each neuron unit, defining the function  $F_N : \mathbb{R}^n \rightarrow \mathbb{R}$ .

**Definition 3 (Range Estimation Problem).** *The problem is defined as follows:*

- INPUTS: Neural network  $\mathcal{N}$ , and input constraints  $P : \mathbf{Ax} \leq \mathbf{b}$  that is compact: i.e., closed and bounded in  $\mathbb{R}^n$ . A tolerance parameter is a real number  $\delta > 0$ .
- OUTPUT: An interval  $[\ell, u]$  such that  $(\forall \mathbf{x} \in P) F_N(\mathbf{x}) \in [\ell, u]$ . I.e.,  $[\ell, u]$  contains the range of  $F_N$  over inputs  $\mathbf{x} \in P$ . Furthermore, the interval is  $\delta$ -tight:

$$u - \delta \leq \max_{\mathbf{x} \in P} F_N(\mathbf{x}) \text{ and } \ell + \delta \geq \min_{\mathbf{x} \in P} F_N(\mathbf{x}).$$

Without loss of generality, we will focus on estimating the upper bound  $u$ . The case for the lower bound will be entirely analogous.

#### 3.1 MILP Encoding

We will first describe the MILP encoding when  $\sigma$  is defined by a ReLU unit. The treatment of more general activation functions will be described subsequently. The real-valued variables of the MILP are as follows:

1.  $\mathbf{x} \in \mathbb{R}^n$ : the inputs to the network with  $n$  variables.
2.  $\mathbf{z}_1, \dots, \mathbf{z}_{k-1}$ , the outputs of the hidden layer. Each  $\mathbf{z}_i \in \mathbb{R}^N$ .
3.  $y \in \mathbb{R}$ : the overall output of the network.

Additionally, we introduce binary (0/1) variables  $\mathbf{t}_1, \dots, \mathbf{t}_{k-1}$ , wherein each vector  $\mathbf{t}_i \in \mathbb{Z}^N$  (the same size as  $\mathbf{z}_i$ ). These variables will be used to model the piecewise behavior of the ReLU units.

Next, we encode the constraints. The first set of constraints ensure that  $\mathbf{x} \in P$ . Suppose  $P$  is defined as  $\mathbf{Ax} \leq \mathbf{b}$  then we simply add the constraints  $C_0 : \mathbf{Ax} \leq \mathbf{b}$ .

For each hidden layer  $i$ , we require that  $\mathbf{z}_{i+1} = \sigma(W_i \mathbf{z}_i + \mathbf{b}_i)$ . Since  $\sigma$  is not linear, we use the binary variables  $\mathbf{t}_{i+1}$  to encode the same behavior:

$$C_{i+1} : \begin{cases} \mathbf{z}_{i+1} \geq W_i \mathbf{z}_i + \mathbf{b}_i, \\ \mathbf{z}_{i+1} \leq W_i \mathbf{z}_i + \mathbf{b}_i + M \mathbf{t}_{i+1}, \\ \mathbf{z}_{i+1} \geq 0, \\ \mathbf{z}_{i+1} \leq M(\mathbf{1} - \mathbf{t}_{i+1}) \end{cases}$$

Note that for the first hidden layer, we simply substitute  $\mathbf{x}$  for  $\mathbf{z}_0$ . This trick of using binary variables to encode piecewise linear function is standard in optimization [28, Ch. 22.4] [29, Ch. 9]. Here  $M$  needs to be larger than the maximum possible output at any node. We can derive fast estimates for  $M$  through interval analysis by using the norms  $\|W_i\|_\infty$  and the bounding box of the input polyhedron.

The output  $y$  is constrained as:  $C_{k+1} : y = W_k \mathbf{z}_k + \mathbf{b}_k$ .

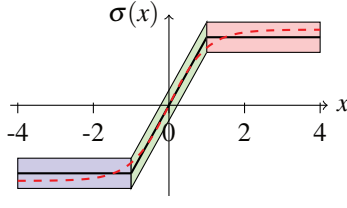
The MILP, obtained by combining these constraints, is of the form:

$$\begin{aligned} \max y \text{ s.t. constraints } C_0, \dots, C_{k+1} (\text{see above}) \\ \mathbf{x}, \mathbf{z}_1, \dots, \mathbf{z}_k, y \in \mathbb{R}^{kN+n+1} \\ \mathbf{t}_1, \dots, \mathbf{t}_{k-1} \in \mathbb{Z}^{(k-1)N} \end{aligned} \tag{3.1}$$

**Theorem 1.** *The MILP encoding in (3.1) is always feasible and bounded. Its optimal solution  $u^*$  corresponds to an input to the network  $\mathbf{x}^* \in P$  such that  $y = F_N(\mathbf{x}) = u^*$ . Furthermore, for all  $\mathbf{x} \in P$ ,  $F_N(\mathbf{x}) \leq u^*$ .*

*Encoding Other Activation Functions:* We will now describe the encoding for more general activation functions including tanh and sigmoid functions. Unlike a ReLU unit, that is described by a two piecewise linear function, approximating these functions may require three or more linear “pieces”. Furthermore, we would like our approximation to include an error estimate that bounds away the differences between the original function and its piecewise approximation.

We will encode the constraint  $y = \sigma(x)$  for a single neuron with  $x \in [-M_x, M_x]$ . The activation function  $y : \sigma(x)$  is approximated by a piecewise linear function:

$$\hat{y} : \begin{cases} a_0x + b_0 + [-\epsilon_0, \epsilon_0] & -M_x \leq x \leq x_1 \\ \dots & \dots \\ a_ix + b_i + [-\epsilon_i, \epsilon_i] & x_i \leq x \leq x_{i+1} \\ \dots & \dots \\ a_kx + b_k + [-\epsilon_k, \epsilon_k] & x_k \leq x \leq M_x \end{cases}$$


Also, the output  $y$  is bounded inside the range  $[-M_y, M_y]$ . This bound is inferred by bounding  $\sigma(x)$  over inputs  $[-M_x, M_x]$ . The bound  $M_x$  is estimated conservatively for a given network through interval analysis.

To encode the constraint  $y = \sigma(x)$  as an MILP, we now introduce binary variables  $t_0, \dots, t_k \in \{0, 1\}^{k+1}$ , wherein  $t_i = 1$  encodes the case when  $x_i \leq x \leq x_{i+1}$ . For convenience, set  $x_0 = -M_x$  and  $x_{k+1} = M_x$ .

We encode that at most one of the cases can apply for any given  $x$ .

$$t_0 + \dots + t_k = 1$$

Next,  $t_i = 1$  must imply  $x_i \leq x \leq x_{i+1}$ :

$$x_i - 2(1 - t_i)M_x \leq x \leq x_{i+1} + 2(1 - t_i)M_x$$

Thus, if  $t_i = 1$  then the bounds are simply  $x_i \leq x \leq x_{i+1}$ . For  $t_i = 0$ , we get  $x_i - 2M_x \leq x \leq x_{i+1} + 2M_x$ , which follows from  $-M_x \leq x \leq M_x$ .

The output  $y$  is related to the inputs as

$$a_ix + b_i - \epsilon_i - 2(1 - t_i)M_y \leq y \leq a_ix + b_i + \epsilon_i + 2(1 - t_i)M_y.$$

Given the encoding for a single unit, we can now write down constraints for encoding an entire neural network with these activation units as an MILP, as shown earlier for ReLU units.

*Solving Monolithic MILP (Branch-and-Cut Algorithm):* Once the MILP is formulated, the overall problem can be handed off to a generic MILP solver, which yields an optimal solution. Most high performance MILP solvers are based on a branch-and-cut approach that will be briefly described here [20].

First, the approach solves the *LP relaxation* of the problem in (3.1) by temporarily treating the binary variables  $\mathbf{t}_1, \dots, \mathbf{t}_k$  as real-valued. The optimal solution of the relaxation is an upper bound to that of the original MILP. The two solutions are equal if the LP solver yields binary values for  $\mathbf{t}_1, \dots, \mathbf{t}_k$ . However, failing this, the algorithm has two choices to *eliminate* the invalid fractional solution:

- (a) Choose a fractional variable  $\mathbf{t}_{i,j}$ , and branch into two subproblems by adding the constraint  $\mathbf{t}_{i,j} = 0$  to one problem, and  $\mathbf{t}_{i,j} = 1$  to the other.
- (b) Add some valid inequalities (cutting planes) that remove the current fractional solutions but preserve all integer solutions to the problem.

In effect, the overall execution of a branch-and-cut solver resembles a tree. Each node represents an MILP instance with the root being the original instance. Figure 2 depicts such a tree visually providing some representative values for the solution of the LP relaxation at each node. The leaves may represent many possibilities:

1. The LP relaxation yields an integral solution. In this case, we use this solution to potentially update the best solution encountered thus far, denoted  $z_{\max}$ . The leaves colored blue in Figure 2 depict such nodes.
2. The LP relaxation is infeasible, eg., the yellow leaf in Figure 2.
3. The LP relaxation’s objective is less than or equal to  $z_{\max}$ , the best feasible solution seen thus far. The leaves colored red in Figure 2 depict this possibility.

Therefore, the key to solving MILPs fast lies in discovering feasible MILP solutions, early on, whose objective function values are as large as possible. In the subsequent section, we will describe how a local search procedure can be used to improve feasible solutions found by the solver (blue leaves in Fig. 2).

## 4 Combining MILP Solvers with Local Search

In this section, we will describe how local search can be used alongside an MILP solver to yield a more efficient solver for the range estimation problem. The key idea is to use local search on a connected subspace of the search space to improve any solution found by the global non-convex optimizer (MILP in our case).

### 4.1 Overall Approach

The overall approach is shown in Algo 1. It consists of two major components: A *local search* represented by the call to **LocalSearch** in line 7 and the call to **SolveMILPUptoThreshold** in line 9.

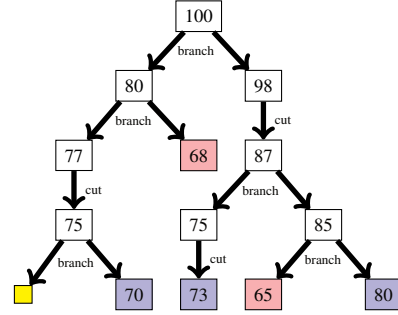


Fig. 2: A tree representation of a branch-and-cut solver execution: each node shows the optimal value of the LP relaxation, if feasible. The leaves are color coded as yellow: infeasible, blue: feasible solution for MILP found, red: suboptimal to already discovered feasible solution.



---

**Algorithm 1** Maximum value  $u$  for a neural network  $\mathcal{N}$  over  $\mathbf{x} \in P$  with tolerance  $\delta > 0$ .

---

```

1: procedure FINDUPPERBOUND( $\mathcal{N}, P, \delta$ )
2:    $\mathbf{x} \leftarrow \text{Sample}(P)$  ▷ Sample an input at random
3:    $u \leftarrow \text{EvalNetwork}(\mathcal{N}, \mathbf{x})$ 
4:    $I \leftarrow \text{FormulateMILPEncoding}(\mathcal{N}, P)$  ▷ See (3.1)
5:   terminate  $\leftarrow$  false
6:   while not terminate do
7:      $(\hat{\mathbf{x}}, \hat{u}) \leftarrow \text{LocalSearch}(\mathcal{N}, \mathbf{x}, P)$  ▷ Note:  $\hat{u} = F_N(\hat{\mathbf{x}})$ 
8:      $u \leftarrow \hat{u} + \delta$ 
9:      $(\mathbf{x}', u', \text{feas}) \leftarrow \text{SolveMILPUptoThreshold}(I, u)$  ▷ Note: If feas then  $u' = F_N(\mathbf{x}')$ 
10:    if feas then
11:       $(\mathbf{x}, u) \leftarrow (\mathbf{x}', u')$ 
12:    else
13:      terminate  $\leftarrow$  true
14:  return  $u$  ▷ return the upper bound  $u$ 

```

---

Local search uses gradient ascent over the neural network, starting from the current input  $\mathbf{x} \in P$  with  $u : F_N(\mathbf{x})$  to yield a new input  $\hat{\mathbf{x}}$  with  $\hat{u} : F_N(\hat{\mathbf{x}})$ , such that  $\hat{u} \geq u$ .

The MILP solver works over the MILP encoding (3.1), formulated in line 4. However, instead of solving the entire problem in one shot, the solver is provided a *target threshold*  $u$  as input. It searches for a feasible solution whose objective is at least  $u$ , and stops as soon as it finds one. Otherwise, it declares that no such solution is possible.

Packages such as Gurobi support such a functionality using the branch-and-cut solver by incrementally maintaining the current search tree. When called upon to find a solution that exceeds a given threshold, the solver performs sufficiently many steps of the branch-and-cut algorithm to either find a feasible solution that is above the requested threshold, or solve the problem to completion without finding such a solution. The approach shown in Algo 1 simply alternates between the local solver (line 7) and the MILP solver (line 9), while incrementing the current threshold by  $\delta$ , the tolerance parameter (line 8). We assume that the procedures **LocalSearch** and **SolveMILPUptoThreshold** satisfy the following properties:

- (P1) Given  $\mathbf{x} \in P$ , **LocalSearch** returns  $\hat{\mathbf{x}} \in P$  such that  $F_N(\hat{\mathbf{x}}) \geq F_N(\mathbf{x})$ .
- (P2) Given the encoding  $I$  and the threshold  $u$ , the **SolveMILPUptoThreshold** procedure either declares **feasible** along with  $\mathbf{x}' \in P$  such that  $u' = F_N(\mathbf{x}') \geq u$ , or declares **not feasible** if no  $\mathbf{x}' \in P$  satisfies  $F_N(\mathbf{x}') \geq u$ .

We recall the basic assumptions thus far: (a)  $P$  is compact, (b)  $\delta > 0$  and (c) properties **P1**, **P2** apply to the **LocalSearch** and **SolveMILPUptoThreshold** procedures. Let us denote the ideal upper bound by  $u^* : \max_{\mathbf{x} \in P} F_N(\mathbf{x})$ .

**Theorem 2.** *Algorithm 1 always terminates. Furthermore, the output  $u$  satisfies  $u \geq u^*$  and  $u \leq u^* + \delta$ .*

*Proof.* Since  $P$  is compact and  $F_N$  is a continuous function. Therefore, the maximum  $u^*$  is always attained for some  $\mathbf{x}^* \in P$ .

The value of  $u$  increases by at least  $\delta$  each time we execute the loop body of the While loop in line 6. Furthermore, letting  $u_0$  be the value of  $u$  attained by the sample obtained in line 2, we can upper bound the number of steps by  $\left\lceil \frac{(u^* - u_0)}{\delta} \right\rceil$ . This proves termination.

We note that the procedure terminates only if **SolveMILPUptoThreshold** returns infeasible. Therefore, appealing to property **P2**, we note that  $(\forall \mathbf{x} \in P) F_N(\mathbf{x}) \leq u$ . Or in other words,  $u^* \leq u$ .

Let  $u_n$  denote the value  $\hat{u}$  returned by **LocalSearch** in the final iteration of the loop and let the corresponding input be  $\mathbf{x}_n$ , so that  $F_N(\mathbf{x}_n) = u_n$ . We have  $u_n \leq u^* \leq u$ . However,  $u_n = u - \delta$ . Therefore,  $u \leq u^* + \delta$ .

## 4.2 Local Search Improvement

The local search uses a gradient ascent algorithm, starting from an input point  $\mathbf{x}_0 \in P$  and  $u = F_N(\mathbf{x}_0)$ , iterating through a sequence of points  $(\mathbf{x}_0, u_0), \dots, (\mathbf{x}_n, u_n)$ , such that  $\mathbf{x}_i \in P$  and  $u_0 < \dots < u_n$ . The new iterate  $\mathbf{x}_{i+1}$  is obtained from  $\mathbf{x}_i$ , in general, as follows:

1. Compute the gradient  $\mathbf{p}_i : \nabla F_N(\mathbf{x}_i)$ .
2. Find a new point  $\mathbf{x}_{i+1} := \mathbf{x}_i + s_i \mathbf{p}_i$  for a step size  $s_i > 0$ .

*Gradient Calculation:* Technically, the gradient of  $F_N(\mathbf{x})$  need not exist for each input  $\mathbf{x}$  if  $\sigma$  is a ReLU function. However, this happens for a set of points of measure 0, and is dealt with by using a smoothed version of the function  $\sigma$  defining the ReLU units.

The computation of the gradient uses the chain rule to obtain the gradient as a product of matrices:  $\mathbf{p} : J_0 \times J_1 \times \dots \times J_k$ , wherein  $J_i$  represents the Jacobian matrix of partial derivatives of the output of the  $(i+1)^{th}$  layer  $\mathbf{z}_{i+1}$  with respect to those of the  $i^{th}$  layer  $\mathbf{z}_i$ . Since  $\mathbf{z}_{i+1} = \sigma(W_i \mathbf{z}_i + \mathbf{b}_i)$  we can compute the gradient  $J_i$  using the chain rule. In practice, the gradient calculation can be piggybacked with function evaluation  $F_N(\mathbf{x})$  so that function evaluation returns both the output  $u$  and the gradient  $\nabla F_N(\mathbf{x})$ .

*Step Size Calculation:* First order optimization approaches present numerous rules such as the Armijo step sizing rules for calculating the step size [19]. Using these rules, we can use an off-the-shelf solver to compute a local maximum of  $F_N$ .

*Locally Active Regions:* Rather than perform steps using a step-sizing rule, we can perform longer steps for the special case of piecewise linear activation functions by defining a locally active region for the input  $\mathbf{x}$ . For the remainder of the discussion, we assume that  $\sigma$  is a piecewise linear function.

We first describe the concept for a ReLU unit. A ReLU unit is *active* if its input  $x \geq 0$ , and inactive otherwise.

**Definition 4 (Locally Active Region (ReLU)).** For an input  $\mathbf{x}$  to the neural network  $\mathcal{N}$ , the locally active region  $\mathcal{L}(\mathbf{x})$  describes the set of all inputs  $\mathbf{x}'$  such that  $\mathbf{x}'$  activates exactly the same ReLU units as  $\mathbf{x}$ .

The concept of locally active region can be generalized to any piecewise linear function  $\sigma$  that has  $J > 0$  pieces, say  $X_1, \dots, X_J$ , where  $\bigcup_i X_i$  is the input space and  $\sigma$

is linear on the input subspace  $X_i$ . For such a piecewise linear function  $\sigma$ , the *locally active region corresponding to an input  $\mathbf{x}$* ,  $\mathcal{L}(\mathbf{x})$ , is the set  $X_i$  such that  $\mathbf{x} \in X_i$ .

Given the definition of a locally active region, we obtain the following property for piecewise linear activation functions.

**Lemma 1.** *For all  $\mathbf{x}' \in \mathcal{L}(\mathbf{x})$ , we have  $\nabla F_N(\mathbf{x}) = \nabla F_N(\mathbf{x}')$ . Furthermore, for a ReLU neural net, the region  $\mathcal{L}(\mathbf{x})$  is described by a polyhedron with possibly strict inequality constraints.*

Let  $\overline{\mathcal{L}}(\mathbf{x}_i)$  denote the closure of the local active set obtained by converting the strict constraints (with  $>$ ) to their non-strict versions (with  $\geq$ ). Therefore, the local maximum is simply obtained by solving the following LP.

$$\max \mathbf{p}_i^T \mathbf{x} \text{ s.t. } \mathbf{x} \in \overline{\mathcal{L}}(\mathbf{x}_i) \cap P, \text{ where } \mathbf{p}_i : \nabla F_N(\mathbf{x}_i).$$

The solution of the LP above ( $\mathbf{x}_{i+1}$ ) yields the next iterate for local search. Note that this solution  $\mathbf{x}_{i+1}$  will typically be at the boundary of  $\mathcal{L}(\mathbf{x}_i)$ . We randomly perturb this solution (or small numerical errors in the solver achieve the same effect) so that  $\nabla F_N(\mathbf{x}_{i+1}) \neq \nabla F_N(\mathbf{x}_i)$ .

*Termination:* The local search is terminated when each step no longer provides a sufficient increase, or alternatively the length of each step is deemed too small. These are controlled by user specified thresholds in practice. Another termination criterion simply stops the local search when a preset maximum number of iterations is exceeded. In our implementation, all three criteria are used.

**Lemma 2.** *Given a starting input  $\mathbf{x}_0 \in P$ , the **LocalSearch** procedure returns a new  $\mathbf{x}' \in P$ , such that  $F_N(\mathbf{x}') \geq F_N(\mathbf{x})$ .*

*Analysis:* For a given neural network  $\mathcal{N}$  and its corresponding MILP instance  $I$ , let us compare the number of nodes  $N_1$  explored by a monolithic solution to the MILP instance with the total number of nodes  $N_2$  explored collectively by the calls to the **SolveMILPUptoThreshold** routine in Algorithm 1. Additionally, let  $K_l$  denote the number of **LocalSearch** calls made by this algorithm. We expect each call to the local search to provide an improved feasible solution to the MILP solver, enabling it to potentially prune more nodes during its search. Therefore, the addition of local search is advantageous whenever

$$N_1 T_{lp} > N_2 T_{lp} + K_l T_{loc},$$

wherein  $T_{lp}$  is the average time taken to solve an LP relaxation (assumed to be the same) and  $T_{loc}$  is the average time for a local search.

A precise analysis is complicated since the future heuristic choices made by the solver can be different, due to the newly added local search iteration. Thus, we resort to an empirical comparison of the original monolithic MILP versus the solution procedure that uses the local search iterations.

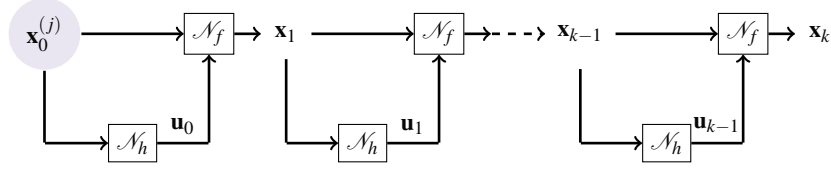


Fig. 3: Unwinding of the closed loop model with plant  $\mathcal{N}_f$  and controller  $\mathcal{N}_h$ , used to estimate reachable sets.

## 5 Application: Reachability Analysis

Neural networks are increasingly used as models of physical dynamics and feedback control laws to achieve objectives such as safety, reachability and stability [13]. However, doing so in a verified manner is a challenging problem. We illustrate the computation reachable set over-approximations for such systems over a finite time horizon in order to prove bounded time temporal properties.

Figure 3 shows the unwinding of the plant network  $\mathcal{N}_f$  and the controller  $\mathcal{N}_h$ , for  $N > 0$  steps. Estimating an over-approximation of the reachable state  $\mathbf{x}_N$  at time  $N$  reduces to solving an output range analysis problem over the unwound network.

We trained a NN to control a nonlinear plant model (Example 17 from [24]) whose dynamics are describe by the ODE:  $\dot{x} = -x^3 + y$ ,  $\dot{y} = y^3 + z$ ,  $\dot{z} = u$ . We approximated the discrete time dynamics of the non linear system using a 4 input, 3 output neural network with 1 hidden layers having 300 neurons. This approximation ensures that unwinding, as shown in Figure 3, results in a neural network.

Next, we devise a model predictive control (MPC) scheme to stabilize this system to the origin, and train the NN by sampling inputs from the state space  $X : [-0.5, 0.5]^3$  and using the MPC to provide the corresponding control. We trained a 3 input, 1 output network, with 5 hidden layers, with the first layer having 100 neurons and the remaining 4 layers to saturate out the control output range.

For illustrative purpose, we compute the reach sets starting from the initial set,  $[0.3, 0.35] \times [-0.35, -0.3] \times [0.35, 0.4]$ , and compute the evolution as shown in Figure 4. Note that, we stop the computation of reach sets once the sets are contained within the target box given by :  $[-0.05, 0.05] \times [-0.05, 0.05] \times [-0.05, 0.05]$ . The reach sets have been superimposed on numerous concrete system trajectories.

## 6 Experimental Evaluation

We have implemented the ideas described thus far in a C++-based tool called SHERLOCK. SHERLOCK combines local search with the commercial parallel MILP solver Gurobi, freely available for academic use [9]. Currently, our implementation supports neural networks with ReLU units. We hope to extend this to other activation functions, as described in the paper.

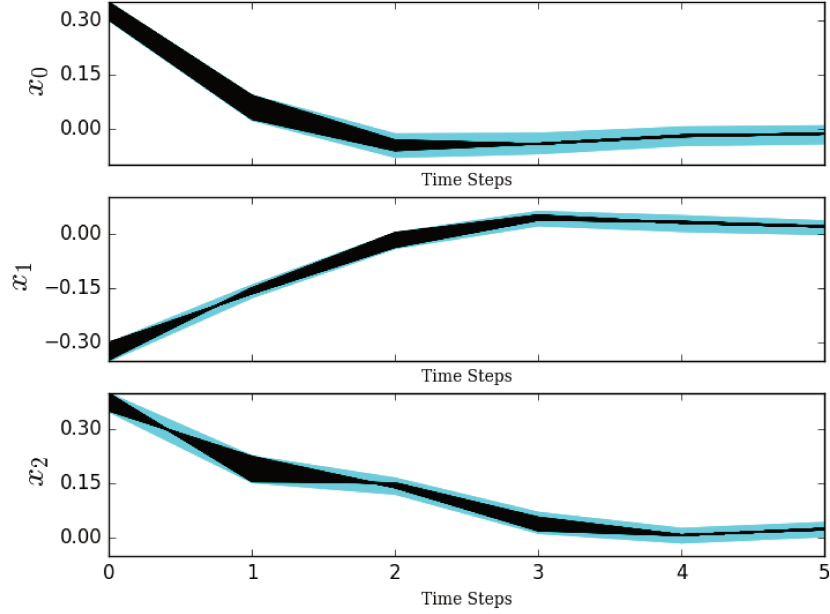


Fig. 4: Evolution of reach sets for the neural network feedback system.

An interval analysis was used to set the  $M$  parameter in the MILP encoding. The tolerance parameter  $\delta$  in Algorithm 1, was set to  $5 \times 10^{-2}$  for all the test cases.

For comparison with Reluplex, we used the implementation available online [15]. However, at its core, Reluplex solves a satisfiability problem that checks if the output  $y$  lies inside a given range for constraints  $P$  over the inputs to the network. To facilitate comparisons, we simply use Reluplex to check if the output range computed by our approach is a valid over-approximation.

We consider a set of 16 microbenchmarks that consist of neural networks obtained from two different sources discussed below.

1. *Known Analytical Functions*: We formulated four simple analytical functions  $y = f(\mathbf{x})$  as shown in Figure 5 controlling for the number of local minima seen over the chosen input range for each function. We then trained a neural network model based on input output samples  $(\mathbf{x}_i, f(\mathbf{x}_i))_{i=1}^N$  for each function. The result yielded networks  $N_0 - N_4$  along with the input constraints.
2. *Unwindings of closed loop systems*: We formulated 12 examples that come from the “unwinding” of a closed loop controller and plant models. The plant models are obtained from our previous work on controller synthesis [24]. The process of training the controller network is discussed elsewhere [7].

Table 1 summarizes the comparison of SHERLOCK against a “monolithic” MILP approach and the Reluplex solver. Since gurobi supports parallel branch-and-bound as a default, we report on the comparison over multicore (23 parallel cores) as well as

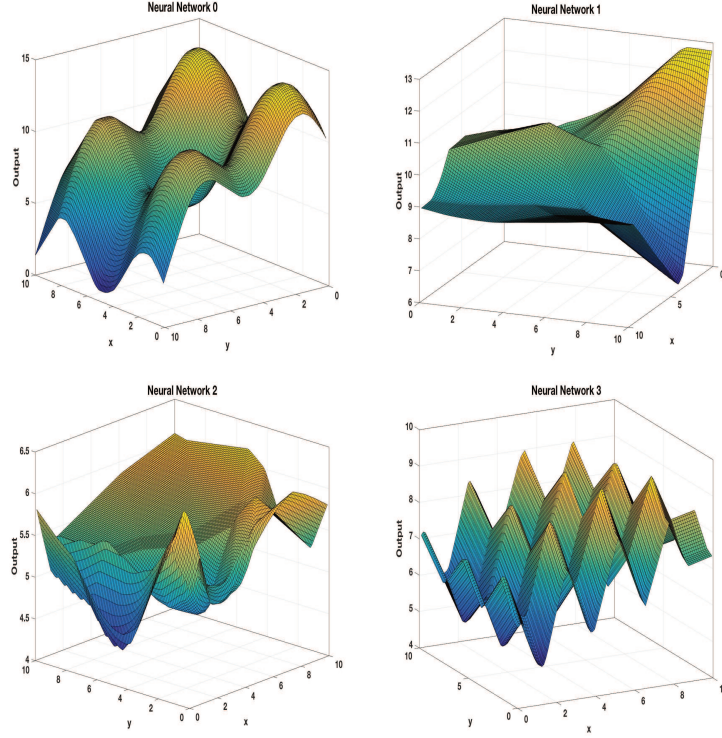


Fig. 5: Plots of the first 4 benchmark functions used to train the neural networks shown in Table 1.

single core deployments. Note that the comparison uses the total CPU time rather than wall clock time.

SHERLOCK on the multicore deployment is faster on all save one of the benchmarks in terms of CPU time. However, comparing the number of nodes explored, we observe that SHERLOCK explores fewer nodes in just 7 out of the 16 cases. We attribute this to the more complex nature of parallel branch-and-cut heuristics, wherein parallel threads may explore more nodes than strictly necessary. For the single core deployment, we note that the total CPU time is strongly correlated with the number of nodes explored. Here, SHERLOCK outperforms the monolithic MILP on the six largest examples with over 1000 neurons ( $N_{10} - N_{15}$ ) in terms of time and number of nodes explored. For the smaller examples, the monolithic solver outperforms our approach, but the running times remain small for both approaches. For two of the networks, ( $N_7$  and  $N_{15}$ ), our starting sample followed by a local search resulted in the global maximum, which was certified by the LP relaxation. This leads to a node count of 1.

Comparing with Reluplex, we note that Reluplex was able to verify the bound in 6 instances but at a larger time cost than SHERLOCK or the monolithic MILP approach. For 10 out of 16 instances, the solver terminates due to an internal error.

ID $x$ $k$ $N$				23 cores				single core				Reluplex $T$
				SHERLOCK		Monolithic		SHERLOCK		Monolithic		
				$T$	$N_c$	$T$	$N_c$	$T$	$N_c$	$T$	$N_c$	
$N_0$	2	1	100	1s	94	2.3s	24	0.4s	44	0.3s	25	9.0
$N_1$	2	1	200	2.2s	166	3.6s	29	0.9s	71	0.8s	38	1m50s
$N_2$	2	1	500	7.8s	961	12.6s	236	2s	138	2.9s	257	15m59s
$N_3$	2	1	500	1.5s	189	0.5s	43	0.6s	95	0.2s	53	12m25s
$N_4$	2	1	1000	3m52s	32E3	3m52s	3E3	1m20s	4.8E3	35.6s	5.3E3	1h06m
$N_5$	3	7	425	4s	6	6.1s	2	1.7s	2	0.9s	2	DNC
$N_6$	3	4	762	3m47s	3.3E3	4m41s	3.6E3	37.8s	685	56.4s	2.2E3	DNC
$N_7$	4	7	731	3.7s	1	7.7s	2	3.9s	1	3.1s	2	1h35m
$N_8$	3	8	478	6.5s	3	40.8s	2	3.6s	3	3.3s	2	DNC
$N_9$	3	8	778	18.3s	114	1m11s	2	12.5s	12	4.3s	73	DNC
$N_{10}$	3	26	2340	50m18s	4.6E4	1h26m	6E4	17m12s	2.4E4	18m58s	1.9E4	DNC
$N_{11}$	3	9	1527	5m44s	450	55m12s	6.4E3	56.4s	483	130.7s	560	DNC
$N_{12}$	3	14	2292	24m17s	1.8E3	3h46m	2.4E4	8m11s	2.3E3	1h01m	1.6E4	DNC
$N_{13}$	3	19	3057	4h10m	2.2E4	61h08m	6.6E4	1h7m	1.5E4	15h1m	1.5E5	DNC
$N_{14}$	3	24	3822	72h39m	8.4E4	111h35m	1.1E5	5h57m	3E4	timeout	-	DNC
$N_{15}$	3	127	6845	2m51s	1	timeout	-	3m27s	1	timeout	-	DNC

Table 1: Performance results on networks trained on functions with known maxima and minima. **Legend:**  $x$  number of inputs,  $k$  number of layers,  $N$ : total number of neurons,  $T$ : CPU time taken,  $Nc$ : number of nodes explored. All the tests were run on a Linux server running Ubuntu 17.04 with 24 cores, and 64GB RAM (DNC : Did Not Complete)

## 7 Conclusion

We presented a combination of local and global search for estimating the output ranges of neural networks given constraints on the input. Our approach has been implemented inside the tool SHERLOCK and we compared our results with those obtained using the solver Reluplex. We also demonstrated the application of our approach to verification of NN-based control systems. Our approach can potentially be applied to verify controllers learned by reinforcement learning techniques.

Our main insight here is to supplement search over a nonconvex space by using local search over known convex subspaces. This idea is generally applicable. In this paper, we showed how this idea can be applied to range estimation of neural networks. The convex subspaces are obtained by fixing the subset of active neurons.

In the future, we wish to improve SHERLOCK in many directions, including the treatment of recurrent neural networks, handling activation functions beyond ReLU units and providing faster alternatives to MILP for global search.

**Acknowledgments:** We gratefully acknowledge inputs from Sergio Mover and Marco Gario for their helpful comments on an earlier version of this paper. This work was funded in part by the US National Science Foundation (NSF) under award numbers

CNS-1646556, CNS-1750009, CNS-1740079 and US ARL Cooperative Agreement W911NF-17-2-0196. All opinions expressed are those of the authors and not necessarily of the US NSF or ARL.

## References

1. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
2. Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
3. Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2016.
4. Robert E Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.
5. Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Piece-wise linear neural network verification: A comparative study. *CoRR*, abs/1711.00455, 2017.
6. Souradeep Dutta. SHERLOCK: An output range analysis tool for neural networks. Available from <https://github.com/souradeep-111/sherlock>.
7. Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Verified inference of feedback control systems using feedforward neural networks. Draft (2017), Available upon request.
8. Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *ATVA*, volume 10482 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2017.
9. Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2016.
10. Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. *CoRR*, abs/1610.06940, 2016.
11. IBM ILOG Inc. CPLEX MILP Solver, 1992.
12. Kyle Julian and Mykel J. Kochenderfer. Neural network guidance for UAVs. In *AIAA Guidance Navigation and Control Conference (GNC)*, 2017.
13. Gregory Kahn, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Plato: Policy learning using adaptive trajectory optimization. *arXiv preprint arXiv:1603.00622*, 2016.
14. Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*, pages 97–117. Springer International Publishing, Cham, 2017.
15. Katz et al. Reluplex: CAV 2017 prototype. <https://github.com/guykatzz/ReluplexCav2017>, 2017.
16. Zeshan Kurd and Tim Kelly. Establishing safety criteria for artificial neural networks. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 163–169. Springer, 2003.
17. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
18. Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.
19. David G. Luenberger. *Optimization By Vector Space Methods*. Wiley, 1969.
20. John E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of Applied Optimization*, page 6577, 2002.
21. Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against deep learning systems using adversarial examples. *CoRR*, abs/1602.02697, 2016.



22. Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification*, pages 243–257. Springer, 2010.
23. Luca Pulina and Armando Tacchella. Challenging smt solvers to verify neural networks. *AI Commun.*, 25(2):117–135, 2012.
24. Mohamed Amin Ben Sassi, Ezio Bartocci, and Sriram Sankaranarayanan. A linear programming-based iterative approach to stabilizing polynomial dynamics. In *Proc. IFAC’17*. Elsevier, 2017.
25. Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards verification of artificial neural networks. In *MBMV Workshop*, page 3040, 2015.
26. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.
27. Vincent Tjeng and Russ Tedrake. Verifying neural networks with mixed integer programming. *CoRR*, abs/1711.07356, 2017.
28. Robert J. Vanderbei. *Linear Programming: Foundations & Extensions (Second Edition)*. Springer, 2001. Cf. <http://www.princeton.edu/~rvdb/LPbook/>.
29. H. Paul Williams. *Model Building in Mathematical Programming (Fifth Edition)*. Wiley, 2013.
30. Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Output reachable set estimation and verification for multi-layer neural networks. *CoRR*, abs/1708.03322, 2017.
31. Weiming Xiang, Hoang-Dung Tran, Joel A. Rosenfeld, and Taylor T. Johnson. Reachable set estimation and verification for a class of piecewise linear systems with neural network controllers, 2018. To Appear in the American Control Conference (ACC), invited session on Formal Methods in Controller Synthesis.