Output Range Analysis for Deep Neural Networks

Souradeep Dutta *, **Susmit Jha**⋆, **Sriram Sanakaranarayanan**†, **Ashish Tiwari**‡
*souradeep.dutta@colorado.edu, ⋆susmit.jha@sri.com, † srirams@colorado.edu, ‡ tiwari@csl.sri.com

Abstract

Deep neural networks (NN) are extensively used for machine learning tasks such as image classification, perception and control of autonomous systems. Increasingly, these deep NNs are also been deployed in high-assurance applications. Thus, there is a pressing need for developing techniques to verify neural networks to check whether certain user-expected properties are satisfied. In this paper, we study a specific verification problem of computing a guaranteed range for the output of a deep neural network given a set of inputs represented as a convex polyhedron. Range estimation is a key primitive for verifying deep NNs. We present an efficient range estimation algorithm that uses a combination of local search and linear programming problems to efficiently find the maximum and minimum values taken by the outputs of the NN over the given input set. In contrast to recently proposed "monolithic" optimization approaches, we use local gradient descent to repeatedly find and eliminate local minima of the function. The final global optimum is certified using a mixed integer programming instance. We implement our approach and compare it with Reluplex, a recently proposed solver for deep neural networks. We demonstrate the effectiveness of the proposed approach for verification of NNs used in automated control as well as those used in classification.

Introduction

Deep neural networks have emerged as a versatile and popular representation model for machine learning due to their ability to approximate complex functions and the efficiency of 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 but more recently, deep NNs are also been adopted in high-assurance systems such as automated control and perception pipeline of autonomous vehicles (Kahn et al. 2016). While traditional system design approaches include rigorous system verification and analysis techniques to ensure the correctness of systems deployed in safety-critical applications (Baier and Katoen 2008), 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 $\texttt{range}(l_i, \phi)$ for each of the network's output l_i that subsumes all possible outputs and is "tight" within a given tolerance δ. We restrict our attention to feedforward deep NNs. Furthermore, the NNs are assumed to use only rectified linear units (ReLUs) (LeCun, Bengio, and Hinton 2015) as activation functions. Adapting to other activation functions will be discussed in an extended version.

Despite these restrictions, the range estimation problem has several applications. Consider a deep NN classification model with the last layer neural units, before the softmax computation to determine the class label, denoted by $l_0, l_1, \ldots, l_k$. Given an image i, $l_j(i)$ denotes the value of the j-th last layer neural unit for the input i and $\texttt{poly}(i)$ denote a compact polyhedron region around i modeling perturbations of the image i. The range of label l_i for the polyhedron input $\texttt{poly}(i)$ is denoted by $\texttt{range}(l_i, \texttt{poly}(i))$. Label j is possible for a perturbation of input in $\texttt{poly}(i)$ if and only if $(\texttt{range}(l_0) \times \texttt{range}(l_1) \times \ldots \times \texttt{range}(l_k)) \cap \{\forall k\ l_k \leq l_j\}$ is not empty. Hence, we can verify if a label j is possible for a given set of perturbations of image i by solving the range estimation problem. This will establish the robustness of the deep NN classification models. Another application of the range estimation problem is to prove the safety of deep NN controllers by proving bounds on the outputs generated by these models. 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 the overall system.

Related Work The importance of analytical certification methods for neural networks has been well-recognized in literature. (Kurd and Kelly 2003) present one of the first categorization of verification goals for NNs used in safety-critical applications. The proposed approach here targets criteria G4 and G5 in (Kurd and Kelly 2003) which aim at ensuring robustness of NNs to disturbances in inputs, and ensuring the output of NNs are not hazardous. The verification of neural networks is a hard problem, and even proving sim- ?

ple properties about $\overset{NN}{\text{them}}$ is known to be NP-complete (Katz et al. 2017). The nonlinearity and non-convexity of deep NNs make their analysis very difficult.

There has been few recent results reported for verifying neural networks. A methodology for the ① analysis of ReLU feed-forward networks is studied in (Katz et al. 2017). Their approach relies on Satisfiability Modulo Theory (SMT) solving (Barrett et al. 2009), while we only use a combination of gradient-based local search and MILP solving (Bixby 2012). The linear programming used for comparison in Reluplex (Katz et al. 2017) performs significantly less efficiently according to the experiments reported in this paper. Note, however, that the scenarios used by Katz et al. are different from those studied here, and are not publicly available for comparison. A related goal of ② 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 (Bastani et al. 2016). A related approach for finding adversarial inputs using SMT solvers that relies on a layer-by-layer analysis is presented in (Huang et al. 2016). The use of SMT solvers for analysis of NNs has also been studied in (Pulina and Tacchella 2012). In contrast, our goal is to not just find adversarial or failing inputs that violate some property of the output but instead, we aim at establishing the guaranteed range of the output for a given polyhedral region of possible inputs. An ③ abstraction-refinement based iterative approach for verification of NNs was proposed in (Pulina and Tacchella 2010). It relies on abstracting NNs as a Boolean combinations of linear arithmetic SMT constraints that is guaranteed to be conservative. Spurious counterexamples arising due to abstraction are used to refine the encoding. Note that our approach can fit well inside such an abstraction-refinement framework.

Contributions We present a novel algorithm for propagating convex polyhedral inputs through a feedforward deep neural network with ReLU activation units to establish ranges s for the outputs of the network. We have implemented our approach in a tool called SHERLOCK. We compare SHERLOCK with a recently proposed deep NN verification engine - Reluplex (Katz et al. 2017). We demonstrate the application of SHERLOCK to establish output range of deep NN controllers as well as to prove the robustness of deep NN image classification models. Our approach seems to scale *consistently* to randomly generated sparse networks with up to 250 neurons and random dense networks with up to 100 neurons. Furthermore, we demonstrate its applications on neural networks with as many as 1500 neurons.

Preliminaries

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

Deep Neural Networks

We will study feed forward neural networks (NN) using so-called "ReLU" units throughout this paper with $n > 0$ inputs and a single output. Let $\mathbf{x} \in \mathbb{R}^n$ denote the inputs and $y \in \mathbb{R}$

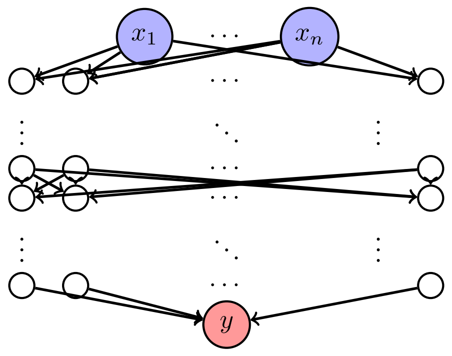

Figure 1: Feedforward Neural Network with ReLU Units.

be the output of the network. Structurally, a NN $\mathcal{N}$ consists of $k > 0$ hidden layers wherein we assume (for simplicity) that each layer has $N > 0$ neurons. We use N_{ij} to denote the j^{th} neuron of the i^{th} layer for $j \in [1, N]$ and $i \in [1, k]$.

A k layer neural network with N neurons in each hidden layer is described by a set of matrices:

$$(W_0, \mathbf{b}_0),\ \ldots\ , (W_{k-1}, \mathbf{b}_{k-1}), (W_k, \mathbf{b}_k)\,,$$

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.

Definition 1.1 (ReLU Unit)**.** *Each neuron in the network implements a nonlinear function σ linking its input value to the output value. In this paper, we consider ReLU units that implement the function $\sigma(z) :\ \max(z, 0)$.*

We extend the definition of σ to apply component-wise to vectors $\mathbf{z}$ as $\sigma(\mathbf{z}) : \begin{pmatrix} \sigma(z_1) \\ \vdots \\ \sigma(z_n) \end{pmatrix}$.

Taking σ to be the ReLU function, we describe the overall function defined by a given network $\mathcal{N}$.

Definition 1.2 (Function Computed by NN)**.** *Given a neural network $\mathcal{N}$ as described above, the function $F : \mathbb{R}^n \to \mathbb{R}$ computed by the neural network is given by the composition $F := F_k \circ \cdots \circ F_0$ wherein $F_i(\mathbf{z}) : \sigma(W_i\mathbf{z} + \mathbf{b}_i)$ is the function computed by the i^{th} hidden layer with F_0 denoting the function linking the inputs to the first layer and F_k linking the last layer to the output.*

For a fixed input $\mathbf{x}$, we say that a neuron N_{ij} is activated if the value input to it is nonnegative. This corresponds to the output of the ReLU unit being the same as the input. It is easily seen that the function F computed by a NN $\mathcal{N}$ is ① continuous and ② piecewise affine, and ③ differentiable almost everywhere in $\mathbb{R}^n$. If it exists, we denote the gradient of this function $\nabla F : (\partial_{x_1} F,\ \ldots,\ \partial_{x_n} F)$. Computing the gradient can be performed efficiently (as described subsequently). } check piece-wise affine

Mixed Integer Linear Programs

Throughout this paper, we will formulate linear optimization problems with integer variables. We briefly recall these

(1) optimization problems, (2) their computational complexity and (3) solution techniques used in practice.

Definition 1.3. *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:*

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

MILP

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 a MILP feasibility problem. It is well known that MILPs are NP-hard problems: the best known algorithms, thus far, have exponential time complexity in the worst case. In contrast, LPs can be solved efficiently using interior point methods.

Problem Definition

Let $\mathcal{N}$ be a neural network with n inputs $\mathbf{x}$, a single output y and weights $\langle (W_0, \mathbf{b}_0), \ldots, (W_k, \mathbf{b}_k)\rangle$. Let F_N be the function defined by such a network.

Definition 1.4 (Range Estimation Problem)**.** *The range estimation problem is defined as follows:*

- INPUTS: (1) *Neural network* $\mathcal{N}$, (2) *input constraints* P : $A\mathbf{x} \leq \mathbf{b}$ *and* (3) *a tolerance parameter* $\delta > 0$.
- OUTPUT: *An interval* $[\ell, u]$ *such that* (1) $(\forall\, \mathbf{x} \in P)\ F_N(\mathbf{x}) \in [\ell, u]$. *I.e,* $[\ell, u]$ *contains the range of* F_N *over inputs* $\mathbf{x} \in P$. *Furthermore, we require* (2) *the interval to be* tight*:*

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

We will assume that the input polyhedron P is compact: i.e, it is closed and has a bounded volume.

Overall Approach

Without loss of generality, we will focus on estimating the upper bound u. The case for the lower bound will be entirely analogous. First, we note that a single MILP can be used to directly compute u, but can be quite expensive in practice. Therefore, our approach will combine a series of MILP feasibility problems alternating with local search steps.

Figure 2 shows a pictorial representation of the overall approach. The approach incrementally finds a series of approximations to the upper bound $u_1 < u_2 < \cdots < u^*$, culminating in the final bound $u = u^*$.

1. The first level u_1 is found by choosing a randomly sampled point $\mathbf{x}_0 \in P$.
2. Next, we perform a series of local iteration steps resulting in samples $\mathbf{x}_1, \ldots, \mathbf{x}_i$ that perform gradient ascent until these steps cannot obtain any further improvement. We take $u_1 = F_N(\mathbf{x}_i)$.
3. A "global search" step is now performed to check if there is any point $\mathbf{x} \in P$ such that $F_N(\mathbf{x}) \geq u_1 + \delta$. This is obtained by solving a MILP feasibility problem.

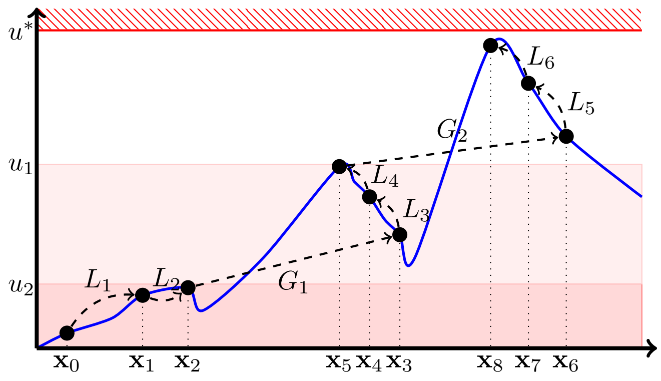

Figure 2: A schematic figure showing our approach showing alternating series of local search $L_1, \ldots, L_6$ and "global search" G_1, G_2 iterations. The points $\mathbf{x}_2, \mathbf{x}_5, \mathbf{x}_8$ represent local minima wherein our approach transitions from local search iterations to global search iterations.

4. If the global search fails to find a solution, then we declare $u^* = u_1 + \delta$.
5. Otherwise, we obtain a new *witness* point $\mathbf{x}_{i+1}$ such that $F_N(\mathbf{x}_{i+1}) \geq u_1 + \delta$.
6. We now go back to the local search step.

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

```
1:  procedure FINDUPPERBOUND($\mathcal{N}$, $P$, $\delta$)
2:      $\mathbf{x} \leftarrow$ Sample($P$)
3:      terminate ← false
4:      while not terminate do
5:          $(\mathbf{x}, u) \leftarrow$ LocalSearch($\mathcal{N}$, $\mathbf{x}$, $P$)
6:          $u \leftarrow u + \delta$
7:          $(\mathbf{x}', u', feasible) \leftarrow$ GlobalSearch($\mathcal{N}$, $u$, $P$)
8:          if feasible then
9:              $(\mathbf{x}, u) \leftarrow (\mathbf{x}', u')$
10:         else
11:             terminate ← true
12:         end if
13:     end while
14:     return $(\mathbf{x}, u)$
15: end procedure
```

Algorithm 1 presents the pseudocode for the overall algorithm. The procedure starts by generating a sample from the interior of P (line 2) using an interior-point LP solver. Next, we iterate between local search iterations (line 5) and global search (line 7). Each local search call provides a new point $\mathbf{x} \in P$ and $u = F_N(\mathbf{x})$ that is deemed a local minimum of the search procedure. We increment the current estimate u by δ, the given tolerance parameter (line 6) and ask the global solver to find a new point $\mathbf{x}'$ such that $u' = F_N(\mathbf{x}')$ is at least as large as the current value of u (line 7). If this succeeds, then we update the current values of $\mathbf{x}, u$ (line 9). Otherwise, we conclude that the current value of u is a valid upper bound within the tolerance δ.

We assume that the procedures **LocalSearch** and **GlobalSearch** satisfy the following properties:

- **(P1)** Given a point $\mathbf{x} \in P$, the **LocalSearch** procedure returns $\mathbf{x}' \in P$ such that $F_N(\mathbf{x}') \geq F_N(\mathbf{x})$.
- **(P2)** Given a neural network $\mathcal{N}$ and the current estimate u, the **GlobalSearch** procedure either ① declares `feasible` along with $\mathbf{x}' \in P$ such that $F_N(\mathbf{x}') \geq F_N(\mathbf{x})$, or ② declares `not feasible` if no $\mathbf{x}' \in P$ satisfies $F_N(\mathbf{x}') \geq F_N(\mathbf{x})$.

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

Theorem 1.1. *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 is always attained.

To see why it terminates, we note that the value of u increases by at least δ each time we execute the loop body of the While loop in line 4. 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 $\left\lceil \frac{(u^*-u_0)}{\delta} \right\rceil$.

We note that the procedure terminates only if **GlobalSearch** 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, $u^* \leq u$.

Likewise, consider the value of u returned by the call to **LocalSearch** in the last iteration of the loop, denoted as u_n along with the point $\mathbf{x}_n$. We note that $F_N(\mathbf{x}_n) = u_n$. Therefore, we have $u_n \leq u^* \leq u$. However, $u_n = u - \delta$. This completes the proof. □

Local and Global Search

In this section, we will describe the local and global search algorithms used in our approach for estimating upper bounds. The approach for estimating lower bounds is identical replacing ascent steps with descent steps.

Local Search Technique

The local search algorithm uses a steepest projected gradient ascent algorithm, starting from an input sample point $\mathbf{x}_0 \in P$ and $u = F_N(\mathbf{x}_0)$, iterating through a sequence of points $(\mathbf{x}_0, u_0), (\mathbf{x}_1, u_1), \ldots, (\mathbf{x}_n, u_n)$, such that $\mathbf{x}_i \in P$ and $u_0 < u_1 < u_2 < \cdots < u_n$.

The new iterate $\mathbf{x}_{i+1}$ is obtained from $\mathbf{x}_i$ as follows:

1. Compute the gradient $\mathbf{z}_i : \nabla F_N(\mathbf{x}_i)$.
2. Compute a "locally active region" $\mathcal{L}(\mathbf{x}_i)$.
3. Solve a linear program (LP) to compute $\mathbf{x}_{i+1}$.

We now describe the calculation of $\mathbf{z}_i$, the definition of a "locally active region" and the setup of the LP.

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

The computation of the gradient uses the chain rule to obtain the gradient as a product of matrices:

$$\mathbf{z} : J_0 \times J_1 \times \cdots \times J_k ,$$

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 following simple rule:
1. If the j^{th} entry $\mathbf{z}_{i+1,j} \geq 0$ then copy j^{th} row of W_i to J_i.
2. Otherwise, set the j^{th} row of J_i to be all zeros.

In practice, the gradient calculation can be *piggybacked* onto the function evaluation $F_N(\mathbf{x})$ so that we obtain both the output u and the gradient $\nabla F_N(\mathbf{x})$.

First order optimization approaches present numerous rules such as the Armijo step sizing rules for calculating the step size (Luenberger 1969). However, in our approach we exploit the local linear nature of the function F_N around the current sample input $\mathbf{x}$ to setup an LP.

Definition 1.5 (Locally Active Region)**.** *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 neurons as $\mathbf{x}$.*

Given the definition of a locally active region, we obtain the following property.

Lemma 1.1. *The region $\mathcal{L}(\mathbf{x})$ is described by a polyhedron with possibly strict inequality constraints. If $\mathbf{x}' \in \mathcal{L}(\mathbf{x})$, then $\nabla F_N(\mathbf{x}) = \nabla F_N(\mathbf{x}')$.*

Proof. (Sketch). To see why $\mathcal{L}(\mathbf{x})$ is a polyhedron, we proceed by induction on the number of hidden layers at the network. Let HN_1 be the subset of neurons in the first hidden layer that are activated by $\mathbf{x}$. We can write linear inequalities that ensure that for any other input $\mathbf{x}'$, the inputs to the neurons in HN_1 are ≥ 0 and likewise the inputs to the neurons not activated in the first layer are < 0. Proceeding layer by layer, we can write a series of constraints that describe $\mathcal{L}(\mathbf{x})$. Since the gradient is entirely dictated by the set of active neurons, it is clear that $\nabla F_N(\mathbf{x}) = \nabla F_N(\mathbf{x}')$. □

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

$$\max\ \mathbf{w}^T\mathbf{y}\ \text{s.t.}\ \mathbf{y} \in \overline{\mathcal{L}}(\mathbf{x}) \cap P\,.$$

The solution of the LP above yields a step of the local search.

Termination: The local search is terminated when ① each step no longer provides a sufficient increase, or alternatively ② the length of each step is deemed too small. These are controlled by user specified thresholds in practice. Another termination criterion simply ③ stops the local search when a preset maximum number of iterations is exceeded. In practical implementations, all three criteria are used.

We note that local search described thus far satisfies property **P1** recalled below.

Lemma 1.2. *Given a point $\mathbf{x} \in P$, the **LocalSearch** procedure returns a new $\mathbf{x}' \in P$, such that $F_N(\mathbf{x}') \geq F_N(\mathbf{x})$.*

Global Search

We will now detail the global search procedure. The overall goal of the global search procedure is to search for a point $\mathbf{x} \in P$ such that $F_N(\mathbf{x}) \geq u$ for a given estimate u of the current upper bound. The approach formulates a MILP feasibility problem, whose real-valued variables are:

1. $\mathbf{x}$: the inputs to the network with n variables.
2. $\mathbf{z}_1, \ldots, \mathbf{z}_{k-1}$, the outputs of the hidden layer. Each $\mathbf{z}_i$ is a vector of N variables.
3. y: the overall output of the network.

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

Now we encode the constraints. The first set of constraints ensure that $\mathbf{x} \in P$. Suppose P is defined as $A\mathbf{x} \leq \mathbf{b}$ then we simply add these as constraints.

We wish to add constraints so that for each hidden layer i, $\mathbf{z}_{i+1} = \sigma(W_i\mathbf{z}_i + \mathbf{b}_i)$ holds. Since σ is not linear, we use the binary variables $\mathbf{t}_{i+1}$ to encode the same behavior:

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

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 (Vanderbei 2001, Ch. 22.4) (Williams 2013, Ch. 9). Here M is taken to be a very large constant as a placeholder for ∞. Additionally, we can derive fast estimates for M by using the norms $||W_i||_\infty$ and the bounding box of the input polyhedron.

The output y is constrained as: $y = W_k\mathbf{z}_k + \mathbf{b}_k$. Finally, we require that $y \geq u$.

The MILP, obtained by combining these constraints, is a feasibility problem without any objective.

Lemma 1.3. *The MILP encoding is feasible if and only if there is an input $\mathbf{x} \in P$ such that $y = F_N(\mathbf{x}) \geq u$.*

Experimental Evaluation

We first describe our C++ implementation of the ideas described thus far, called SHERLOCK. SHERLOCK combines local search with the commercial MILP solver Gurobi, freely available for academic use (Gurobi Optimization 2016). The tolerance parameter δ is fixed to 10^{-3}.

For comparison with Reluplex, we used the proof of concept implementation available at (Katz et al. 2017). However, at its core, Reluplex does not compute intervals. Therefore, we use a bisection scheme to repeatedly query Reluplex with ever smaller intervals until, we have narrowed down the actual range within a tolerance $\hat{\delta} = 10^{-2}$.

Randomly Generated Networks

First, we generate numerous randomly generated networks with 1 output by varying the number of inputs (n), the number of hidden layers (k), the number of neurons per hidden layer (N) and the fraction of non-zero edge weights (s). Columns 1-4 of Table 1 shows the set of values chosen for $\langle n, k, N, s\rangle$. For each chosen set, we generated 100 random neural networks. The zero weight edges were chosen by flipping a coin with probability $1 - s$ of choosing a weight to be zero. The nonzero weights were uniformly chosen in the range $[-1, 1]$. Next, we compare our approach against Reluplex solver assuming that the inputs are in the range $[-1, 1]$. Each tool is run with a timeout of 20 minutes. Table 1 shows the comparison. SHERLOCK outperforms Reluplex in terms of the number of examples completed within the given timeout. However, we note that many of the Reluplex instances failed before the timeout due to an "error" reported by the solver. From the table we conclude that the total number of neurons and the sparsity of the network have a large effect on the overall performance. In particular, no approach is able to handle dense examples with 250+ neurons.

Microbenchmarks on Known Functions

Next, we consider a set of 6 microbenchmarks that consist of networks trained on known 2 input, single output functions shown in Figure 3. These functions provide varying numbers of local minima. We then sampled inputs/output pairs, and trained a neural network with 1 hidden layer for the first 5 functions and 6 hidden layer for the last. Next, we compared SHERLOCK against the Reluplex solver in terms of the ability to predict the output range of the function given the input range $[-1, 1]^2$ for the two inputs.

Table 2 summarizes the results of our comparison. We note that in all cases, our approach can solve the resulting problem within 10 minutes. However, with a timeout of 1 hour, Reluplex is unable to deduce a range for 4 out of 6 instances.

Illustrative Applications

We illustrate the use of SHERLOCK to infer properties of deep NNs for important applications, starting with a neural network that controls a nonlinear system. As mentioned earlier, such networks are increasingly popular using deep policy learning (Kahn et al. 2016).

We trained a NN to control a nonlinear plant model (Example 7 from (Sassi, Bartocci, and Sankaranarayanan 2017)) whose dynamics are describe by the ODE:

$$\dot{x} = z^3 - y,\ \dot{y} = z,\ \dot{z} = u\,.$$

We first① devise a model predictive control (MPC) scheme to stabilize this system to the origin, and② train the NN by sampling inputs from the state space $X : [-0.5, 0.5]^3$ and ③ using the MPC to provide the corresponding control.

We trained a 3 input, 1 output network, with 2 hidden layers, having 300 neurons in the first layer and 200 neurons in the second layer. For any state $(x, y, z) \in X$, we seek to know the range of the output of the network to deduce the maximum/minimum control input. This is a direct application of the range estimation problem. SHERLOCK is able to deduce a tight range for the control input: $[-2.68772, 3.00263]$.

Handwriting Recognition Networks

We illustrate an application of range estimation to large neural networks involved in pattern classification, wherein, we

Network Params.				SHERLOCK							Reluplex			
n	k	N	s	N_c	T			# Iters			N_c	T		
					avg	min	max	avg	min	max		avg	min	max
2	2	10	0.5	100	0.47	0.0	1.95	3.5	2	17	66	2.5	1.0	2.6
3	2	10	0.5	100	0.72	0.0	4.6	4.8	2	15	71	4	0.85	11.0
4	5	10	0.5	100	4.6	0.01	25.2	4.8	2	19	53	207.5	21.8	1036.3
5	8	10	0.05	100	0.01	0	0.02	2	2	2	3	1.65	1.2	1.89
5	8	10	0.2	100	0.16	0.01	1.93	2	2	3	7	6.1	1.9	11
5	8	10	0.5	64	40	0.03	1023.4	4.5	2	31	12 (*8)	621.6	121.7	1025.7
5	8	10	0.8	44	167	0.18	1068.8	3	2	11	0	x	x	x
5	8	10	0.95	40	51	0.2	1201	2.5	2	4	0	x	x	x
10	2	10	0.5	99	1.65	0.0	7.74	24	2	104	86 (*1)	42.1	1.6	339.7
10	5	10	0.5	91	16.5	0.02	86.4	22	2	407	7	433.2	23.2	1111.2
10	5	50	0.05	99	6.7	0.08	76.5	3	2	12	32 (*1)	70.3	9	452
10	5	50	0.2	8	354.8	0.2	1429.1	2	2	3	0	x	x	x
10	5	50	0.4	8	35.4	1.6	114.5	3	3	3	0	x	x	x
10	5	50	0.5	0	x	x	x	x	x	x	0	x	x	x

Table 1: Performance on randomly generated neural networks. **Legend:** n: # inputs, k: # hidden layers, N: # neurons/layer, s: sparsity fraction, N_c: successfully solved (out of 100), T: running time, # Iters: number of iterations. The number $(*m)$ denotes that Reluplex solved m instances that were not solved by our approach. All experiments were run on a Linux workstation running Ubuntu 17.04 with 64 GB RAM and 23 cores.

ID	$k \times N$	T_s	T_{rplex}
N_0	1×100	1.9s	1m 55s
N_1	1×200	2.4s	13m 58s
N_2	1×500	17.8s	timeout
N_3	1×500	7.6s	timeout
N_4	1×1000	7m 57.8s	timeout
N_5	6×250	9m 48.4s	timeout

Table 2: Performance results on networks trained from the 2 input functions shown in Fig. 3. **Legend:** k number of layers, N: number of neurons/layer, T_s: Time taken by SHERLOCK, T_{rplex}: Time taken by Reluplex solver. All results in this table were obtained on a Macbook Pro running ubuntu 16.04 with 3 cores and 8 GB RAM.

wish to infer classification labels for images. The MNIST handwriting recognition data set has emerged as a prototype application (LeCun and Cortes 2010).

Given a neural network classifier N, and a given image input I, we wish to explore a set of possible perturbations to the image to find a nearby input that can alter the label that N provides to the perturbed image. Alternatively, given a space of possible perturbations, we wish to prove that the network N is *robust* to these perturbations: i.e, no perturbation can alter the label that N provides to the perturbed image.

We trained a NN with 28×28 inputs, and 10 outputs, and 3 hidden layers with $200, 100$ and 50 neurons in the layers respectively on the MNIST digit recognition data set (LeCun and Cortes 2010). The network has 10 outputs that are then fed to a "softmax" output layer to provide the final classification. For the purposes of our application, we remove this softmax layer and examine these 10 outputs that represent numerical scores corresponding to the digits $0 - 9$.

We first ① use our approach to generate adversarial perturbations to the images, which can flip the output. To do so, we define for each image a set of pixels and a range of possible perturbation of these pixels. This forms a polyhedron that we will call the perturbation space. Next, we use our approach to ② explore points in this perturbation space, stopping as soon as we obtain a point with a different label. This is achieved simply using local search iterations. Figure 4, demonstrates such an effect, we started with images that are classified correctly as the digits "0" and "1". The perturbations are applied to selected pixels resulting in the digit "0" labeled as "8", and "1" labeled as "2".

However, rather finding a perturbation, we wish to *prove* robustness to a class of perturbations. I.e, for all possible perturbations, the label ascribed by the network to an image remains the same.

We chose an image labeled as 5 and specified a set of perturbations to a block of size 5×10 on the top left corner of the image. One such perturbation is shown in Figure 6. We used SHERLOCK to compute a range over the output labels for all such perturbations. Figure 5 shows the output ranges for each label. It is clear that all perturbations will lead the softmax layer to choose the label 5 over all other labels for the image.

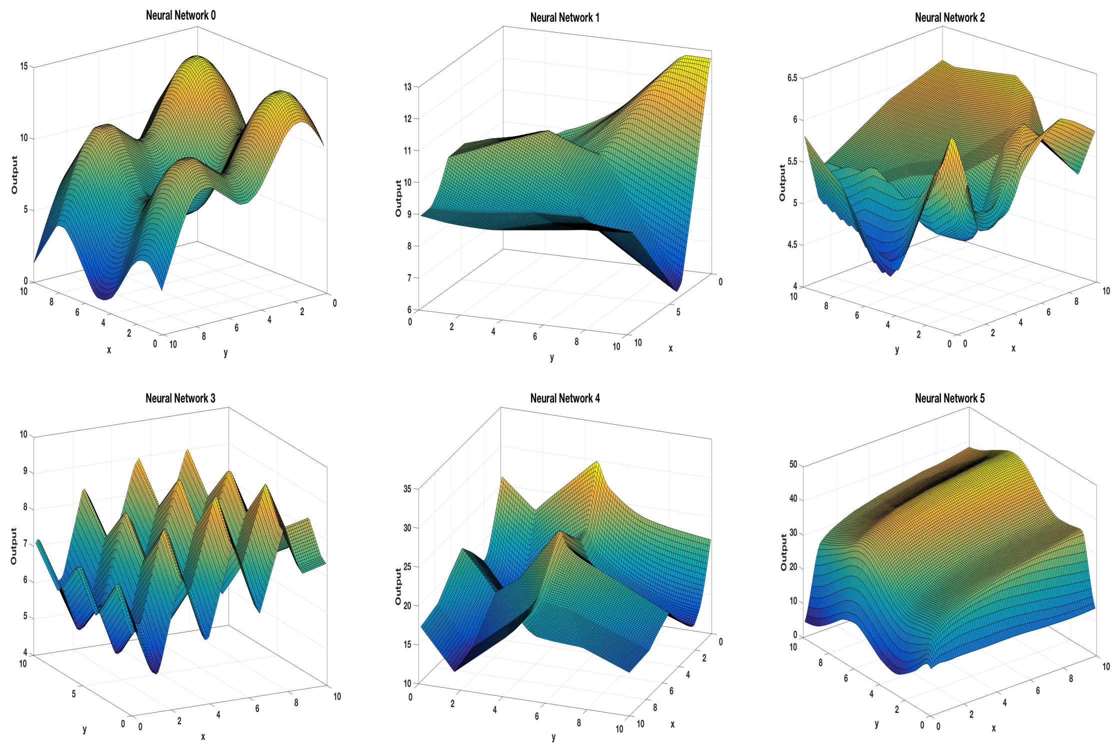

Figure 3: Plots of the benchmark functions used to train the neural networks shown in Table 2.

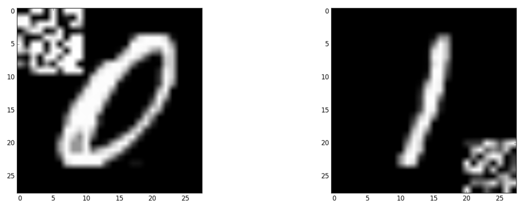

Figure 4: (**Left**) Perturbing pixels in the top right corner of a 28×28 image changes the network's output label from 0 to 8. (**Right**) Perturbation changes label from 1 to 2.

?

$range(l_0)$	$\subseteq [0.0, 0.763]$	$range(l_1)$	$\subseteq [0, 0]$
$range(l_2)$	$\subseteq [0, 0]$	$range(l_3)$	$\subseteq [3.365, 5.71]$
$range(l_4)$	$\subseteq [0, 0]$	$range(l_5)$	$\subseteq [9.62, 13.4]$
$range(l_6)$	$\subseteq [1.24, 2.25]$	$range(l_7)$	$\subseteq [0.016, 0.592]$
$range(l_8)$	$\subseteq [2.31, 3.5]$	$range(l_9)$	$\subseteq [3.39, 5.3]$

Figure 5: Output range computed by SHERLOCK for all possible perturbations of 5×10 block as shown in Figure 6.

Conclusion

Thus, we have 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 compared the results obtained with the solver Reluplex. We have also demonstrated the application of our approach to interesting applications to control and classification problems.

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 the MILP for global search.

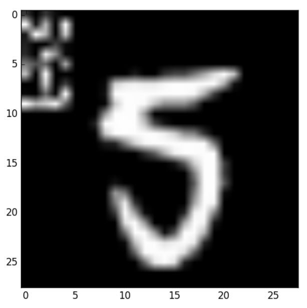

Figure 6: The image above shows the effect of perturbation on an image that the trained NN is robust to. The perturbation were only allowed to effect a block of size 5×10 on the top left of the image

References

[Baier and Katoen 2008] Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. MIT Press.

[Barrett et al. 2009] Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 2009. Satisfiability modulo theories. *Handbook of satisfiability* 185:825–885.

[Bastani et al. 2016] Bastani, O.; Ioannou, Y.; Lampropoulos, L.; Vytiniotis, D.; Nori, A.; and Criminisi, A. 2016. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems*, 2613–2621.

[Bixby 2012] Bixby, R. E. 2012. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica* 107–121.

[Gurobi Optimization 2016] Gurobi Optimization, I. 2016. Gurobi optimizer reference manual.

[Huang et al. 2016] Huang, X.; Kwiatkowska, M.; Wang, S.; and Wu, M. 2016. Safety verification of deep neural networks. *CoRR* abs/1610.06940.

[Kahn et al. 2016] Kahn, G.; Zhang, T.; Levine, S.; and Abbeel, P. 2016. Plato: Policy learning using adaptive trajectory optimization. *arXiv preprint arXiv:1603.00622*.

[Katz et al. 2017] Katz, G.; Barrett, C.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. Cham: Springer International Publishing. 97–117.

[Katz et al. 2017] Katz et al. 2017. Reluplex: CAV 2017 prototype. `https://github.com/guykatzz/ReluplexCav2017`.

[Kurd and Kelly 2003] Kurd, Z., and Kelly, T. 2003. Establishing safety criteria for artificial neural networks. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, 163–169. Springer.

[LeCun and Cortes 2010] LeCun, Y., and Cortes, C. 2010. MNIST handwritten digit database.

[LeCun, Bengio, and Hinton 2015] LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *Nature* 521(7553):436–444.

[Luenberger 1969] Luenberger, D. G. 1969. *Optimization By Vector Space Methods*. Wiley.

[Pulina and Tacchella 2010] Pulina, L., and Tacchella, A. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification*, 243–257. Springer.

[Pulina and Tacchella 2012] Pulina, L., and Tacchella, A. 2012. Challenging smt solvers to verify neural networks. *AI Commun.* 25(2):117–135.

[Sassi, Bartocci, and Sankaranarayanan 2017] Sassi, M. A. B.; Bartocci, E.; and Sankaranarayanan, S. 2017. A linear programming-based iterative approach to stabilizing polynomial dynamics. In *Proc. IFAC'17*. Elsevier.

[Vanderbei 2001] Vanderbei, R. J. 2001. *Linear Programming: Foundations & Extensions (Second Edition)*. Springer. Cf. `http://www.princeton.edu/~rvdb/LPbook/`.

[Williams 2013] Williams, H. P. 2013. *Model Building in Mathematical Programming (Fifth Edition)*. Wiley.