# Sherlock : A Tool for Verification of Deep Neural Networks

Souradeep Dutta*, Taisa Kushner *, Susmit Jha †, Sriram Sankaranarayanan*,
Natarajan Shankar†, Ashish Tiwari #
**University of Colorado, Boulder, USA*
† *SRI International, Menlo Park, USA*
# *Microsoft, Redmond, USA*

***Abstract*—We present Sherlock a tool for verification of deep neural networks along with some recent applications of the tool to verification and control problems for autonomous systems. Deep Neural Networks have emerged as an ubiquitous choice of concept representation in machine learning, and are commonly used as function approximators in many learning tasks. Learning methods using deep neural networks are often referred to as deep learning, and they have reached human-level accuracy on benchmark data-sets for several tasks such as image classification and speech recognition. This has inspired the adoption and integration of deep learning methods into safety-critical systems. For example, deep learning is often used in perception and several decision-making modules in automated vehicles. This has created a challenge in verifying and certifying these systems using traditional approaches developed for software and hardware verification. Further, these models are known to be brittle and prone to adversarial examples. Establishing robustness of these models is, thus, an important challenge. For safe integration of learning into critical systems, we must be able to encapsulate these models into assumption-guarantee contracts by verifying guarantees on the outputs of the model given the assumptions on its inputs. We discuss some of our recent work on this topic of verifying deep neural networks that is currently available as an open-source tool, Sherlock.**

***Index Terms*—neural networks, verification, optimization.**

## I. Introduction and Motivation

A large of part of machine learning is about building good function approximations from input-output data. Deep Neural Networks have emerged as an obvious choice for this task since they are "universal" function approximators. Deep NNs are also being adopted in high-assurance systems, such as automated control, perception pipeline of autonomous vehicles, and aircraft collision avoidance. But unlike other traditional system design approaches, there are few known and scalable methods to verify these systems. We address this problem by proposing novel techniques to solve this problem. This paper is a survey of the techniques presented in [1], [2], [11] that has been implemented in a publicly available open-source tool, Sherlock [1].

Sherlock uses mixed-integer linear programming (MILP) solver but it does not merely compile the verification into an MILP problem. Sherlock first uses 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. Such an encoding into MILP is a standard approach to handling piecewise linear functions. 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 the corresponding guarantee that holds on the output. This enables us to infer an assume-guarantee contract on the overall deep neural network. Sherlock augments this simple use of MILP solving with a local search that exploits the local continuity and differentiability properties of the function represented by the network. These properties are not exploited by MILP solvers which typically use a branch-and-cut approach. On the other hand, local search alone may get "stuck" in local minima. Sherlock 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, Sherlock can exploit local properties of the neural network function but at the same time avoid the problem of local minima, and thus, solve the verification of deep neural networks more efficiently.

## II. Verification problem / Set Propagation

At an abstract level a deep neural network computes a function from a set of inputs to some set of outputs. The question that we address here is as follows:

> Given a neural network (NN), and constraints (assumptions) which define a set of inputs to the network, provide a *tight* over-approximation (guarantee) of the output set.

This serves as one of the main primitives in verification of neural networks. Deep neural networks are very common in applications such as image classification and autonomous control. In image classification networks, since each image is a point in the high dimensional pixel space, a polyhedral set may be used to represent all possible bounded perturbations to a given image. If, for such a set, we can guarantee that the output of the classification remains unaltered, then we have proved that the neural network is *robust* to bounded pixel noise. Besides image classification, neural networks are increasingly used in the control of autonomous systems, such as self-driving cars, unmanned aerial vehicles, and other robotic systems. A

[1] https://github.com/souradeep-111/sherlock

typical approach to verify these systems involves a *reachability computation* to estimate the forward reachable set as time evolves. Using this, it is possible to prove that, no matter what the initial condition of a system is, it always reaches a target region in finite time. For instance, we wish prove that, an autonomous car whose inputs are provided by a neural network controller's feedback, will remain within a fixed lateral distance from the center of the road (desired trajectory), while remaining under the speed limit.

### A. *Preliminaries*

We will address the output range analysis problem for a neural network with a single output. The extension to multiple output neural networks will be straightforward. Let $x \in \Re^n$ be an input to a NN, and $y \in \Re$ be the output of the network. A typical neural network consists of layers, where each layer computes some function on the outputs of the previous layer and feeds it's output to the next layer. That is, for a $k$ layer neural network, we get a composition of $k$ functions. Each function is a matrix multiplication, followed by an element wise computation of an activation function. A $k$ layer neural network with $N$ neurons in each hidden layer is described by a set of matrices: $[(W_0, b_0), \ldots, (W_{k-1}, b_{k-1}), (W_k, b_k)]$.

**Definition II.1** (ReLU Unit)**.** Each neuron in the network implements a nonlinear function $\sigma$ 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 $\sigma$ to apply component-wise to vectors $z$ as $\sigma(z) : \left(\begin{array}{c} \sigma(z_1) \\ \vdots \\ \sigma(z_n) \end{array}\right)$.

Taking $\sigma$ to be the ReLU function, we describe the overall function defined by a given network $N$ as follows:

**Definition II.2** (Function Computed by neural networks)**.** Given a neural network $N$ as described above, the function $F : \Re^n \to \Re$ computed by the neural network is given by the composition $F := F_k \circ \cdots \circ F_0$ wherein $F_i(z) : \sigma(W_i z + b_i)$ 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.

## III. Methodology

Let $N$ be a neural network with $n$ input vector, $x$, a single output $y$, and weights $[(W_0, b_0), \ldots, (W_k, b_k)$. Let $F_N$ be the function defined by such a network. The general problem of verifying neural network and establishing an assume-guarantee contract on its inputs and outputs can be simplified to range estimation problem by suitably transforming the inputs and outputs such that the assumption constraints are described by a polyhedron and the guarantee constraints to be derived over the outputs can be represented as intervals. The universal approximation property of neural networks can be used to approximately encode such transformation as a part of the network itself. Thus, we focus on range estimation problem and rely on reducing other verification problems to it.

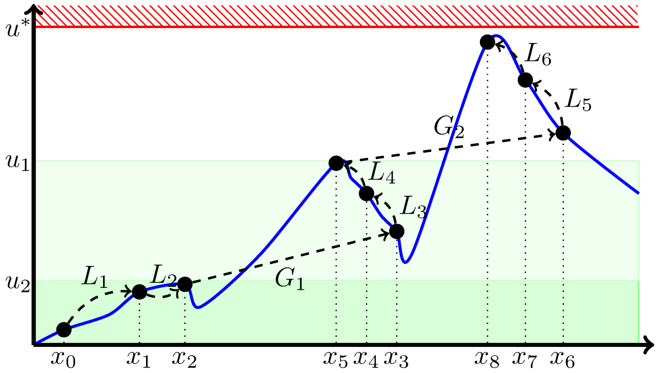

Fig. 1. A schematic figure showing our approach showing alternating series of local search $L_1, \ldots, L_6$ and "global search" $G_1, G_2$ iterations. The points $x_2, x_5, x_8$ represent local minima wherein our approach transitions from local search iterations to global search iterations.

**Problem Statement :** The *range estimation* problem is defined as follows:

- Inputs: Neural Network $N$, input constraints $P : Ax \leq b$ and a tolerance parameter $\delta > 0$.
- Output: An interval $[\ell, u]$ such that $(\forall\ x \in P)\ F_N(x) \in [\ell, u]$. i.e, $[\ell, u]$ contains the range of $F_N$ over inputs $x \in P$. Furthermore, we require the interval to be *tight*:

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

We will assume that the input polyhedron $P$ is compact: i.e, it is closed and has a bounded volume. It was shown in [3] that even proving simple properties is NP complete. Simple properties, like proving that there exists an assignment from input set to an output set, which respects the constraints imposed by the neural network. So, one of the fundamental challenges in this problem is to tackle the exponential nature.

### A. *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 Mixed Integer Linear Programming (MILP) problem can be formulated, and then query a solver to directly compute $u$. Unfortunately, that can be quite expensive in practice. Therefore, our approach will combine a series of MILP feasibility problems alternating with local search steps.

Figure 1 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 $x_0 \in P$.
2) Next, we perform a series of local iteration steps resulting in samples $x_1, \ldots, x_i$ that perform gradient ascent until these steps cannot obtain any further improvement. We take $u_2 = F_N(x_i)$.
3) A "global search" step is now performed to check if there is any point $x \in P$ such that $F_N(x) \geq u_2 + \delta$. This is obtained by solving a MILP feasibility problem.

4) If the global search fails to find a solution, then we declare $u^* = u_2 + \delta$.
5) Otherwise, we obtain a new *witness* point $x_{i+1}$ such that $F_N(x_{i+1}) \geq u_2 + \delta$.
6) We now go back to the local search step.

The ideas discussed here for the output range analysis have been implemented in Sherlock [1]. For neural networks with multiple outputs, we can individually find the bounds for each of the network outputs, and then combine them to form a hyper-rectange in the output dimensional space. This can be extended to using a template polyhedron to obtain tighter bounds, in the output dimension, described in the next section. In general, we can obtain guarantees on the output from a given class defined by the constraint template used in the minimization/maximization step of the presented approach. Our current implementation in Sherlock built on top of MILP solvers requires the template to be linear.

## IV. Application 1: Reachability Analysis

In this section, we briefly describe how we can use the above algorithm to verify behaviors of autonomous systems, with neural networks as controllers. Details are presented in [2]. A closed loop system, $\mathcal{C}$, is described by the neural networks $f_p$, for the system model, and $f_h$, for the control law. The plant model function $f_p$, gives the state of the system in the next time step, given the states of the system at the current time step. That is, $x(t+1) = f_p(x(t), f_h(x(t)))$, where $x(t) \in \Re^n$, is the state of the system at time time $t$, in an $n$ dimensional space.

Thus, given an initial state $X_0$ (represented as a polyhedron over the state space), we wish to compute symbolic representations for sets $X_1, X_2, \ldots, X_K$ wherein $X_i$ represents the reachable states of the closed loop system given by the composition of the plant $f_p$ and the feedback law $f_h$, in $i$ steps. Here $K$ is some fixed time horizon. We will use range computation, as a primitive for checking reachability, invariance and stability properties.

### A. Post-Condition Operator

The computation of the reach sets of the closed loop system starts with an effort to compute over-approximation of the post operator:

$$\mathsf{post}(X; f_p, f_h):\ \{\mathbf{x} \in \mathbb{R}^n \mid (\exists \mathbf{x}_0 \in X)\, \mathbf{x} = f_p(\mathbf{x}_0, f_h(\mathbf{x}_0))\}\,.$$

For an input set $X$, the exact set of the output map of the post operator can be prohibitively expensive to compute: in general, it's a union of polyhedrons, the count of which is exponential in the number of neurons in the two given networks $f_p$ and $f_h$. Instead, we use a single polyhedron $P(X)$ that approximates the post condition.

For that purpose, we use a template polyhedra:

**Definition IV.1** (Template Polyhedra)**.** A template $T$ is a set of expressions $T:\ \{\mathbf{e}_1, \ldots, \mathbf{e}_r\}$ wherein each $\mathbf{e}_i$ is an linear expression of the form $\mathbf{c}_i^t\mathbf{x}$ over the state variables. A template polyhedron $P$ over a template $T$ is of the form:

$$\bigwedge_{j=1}^{r} \ell_j \leq \mathbf{e}_j \leq u_j\,,$$

for bounds $\ell_j, u_j$ over each template expression $\mathbf{e}_j$.

For a fixed template $T$, the reachable sets are represented by template polyhedra over the above templates. The post condition operation $\mathsf{post}(X; f_p, f_h)$, can now be substituted by a template-based post-condition operator $\mathsf{post}_T(X; f_p, f_h)$ that produces bounds $\ell_j, u_j$ for each $\mathbf{e}_j \in T$ by solving the following optimization problem:

$$\ell_j(u_j):\ \min(\max)\, \mathbf{e}_j[\mathbf{x}]\ \mathsf{s.t.}\mathbf{x}_0 \in X_0,\ \mathbf{u} = f_h(\mathbf{x}_0),\ \mathbf{x} = f_p(\mathbf{x}_0, \mathbf{u})\,.$$

The above optimization problem, is defined using neural network functions $f_h$ and $f_p$. However, the combination of local search and MILP encoding used in our tool Sherlock can be modified almost trivially to solve this optimization problem. Furthermore, the guarantees used in Sherlock extend. Thus, we guarantee that the reported result is no more than $\epsilon$ away from the true value, for the given tolerance parameter $\epsilon$.

### B. Accelerating Reachable Set Computation

The computation of reach sets can be extended beyond single step using Sherlock. It is possible to use the tool for a $k$ step reachability $\mathsf{post}_T^{(k)}(X; f_p, h)$ with the tolerance factor $\epsilon$, in a very straight forward manner. To achieve this, we calculate the bounds $\ell_j^{(k)}, u_j^{(k)}$ as

$$\ell_j^{(k)}(u_j^{(k)}):\ \left[\begin{array}{rl} \min(\max) & \mathbf{e}_j[\mathbf{x}_k] \\ \mathsf{s.t.} & \mathbf{x}_0 \in X, \\ \mathbf{x}_1 & = f_p(\mathbf{x}_0, f_h(\mathbf{x}_0)), \\ \ldots,\ \mathbf{x}_k & = f_p(\mathbf{x}_{k-1}, f_h(\mathbf{x}_{k-1})) \end{array}\right].$$

### C. Checking Reachability Property:

A fundamental goal in computing reachable sets is to prove that the system trajectories converge to a target set, starting from the initial set. It suffices to show that the reach sets computed eventually land inside the target set $T$, in a finite number of time steps. Thus, the problem of checking reachability of a target set $T$ is performed iteratively as

## V. Application 2: Robust Control

Range propagation ideas can be useful much beyond verification goals. An artificial pancreas is a device which is used for the automatic regulation of blood glucose in patients with Type 1 Diabetes. In [11], the authors trained a neural network to predict the blood glucose values of a patient 1 hour into the future, from the blood glucose and insulin values, in the past 3 hours. That is, $G(t + 60mins) = F_{nn}(G[\ldots], I[\ldots])$, where $G[\ldots]$ and $I[\ldots]$ represents an array of blood glucose, and insulin input values in the past 3 hours at 5 mins intervals, and $F_{nn}$ is a neural network model of the system. Thus, given the array $G[\ldots]$ and $I[\ldots]$, we used the techniques implemented in Sherlock to compute *SAFE* values of the insulin inputs such that the blood glucose stays within the euglycemic range.

TABLE I

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)

| | | | | 23 cores | | | | single core | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Monolithic | | | | Monolithic | | Reluplex |
| ID | $x$ | $k$ | $N$ | $T$ | $Nc$ | $T$ | $Nc$ | $T$ | $Nc$ | $T$ | $Nc$ | $T$ |
| $N_0$ | 2 | 1 | 100 | 1**s** | 94 | 2.3**s** | 24 | 0.4**s** | 44 | 0.3**s** | 25 | 9.0 |
| $N_1$ | 2 | 1 | 200 | 2.2**s** | 166 | 3.6**s** | 29 | 0.9**s** | 71 | 0.8**s** | 38 | 1**m**50**s** |
| $N_2$ | 2 | 1 | 500 | 7.8**s** | 961 | 12.6**s** | 236 | 2s | 138 | 2.9**s** | 257 | 15**m**59**s** |
| $N_3$ | 2 | 1 | 500 | 1.5**s** | 189 | 0.5**s** | 43 | 0.6**s** | 95 | 0.2**s** | 53 | 12**m**25**s** |
| $N_4$ | 2 | 1 | 1000 | 3**m**52**s** | $32E3$ | 3**m**52**s** | $3E3$ | 1**m**20**s** | 4.8E3 | 35.6**s** | 5.3E3 | 1**h**06**m** |
| $N_5$ | 3 | 7 | 425 | 4**s** | 6 | 6.1**s** | 2 | 1.7**s** | 2 | 0.9**s** | 2 | DNC |
| $N_6$ | 3 | 4 | 762 | 3**m**47**s** | 3.3E3 | 4**m**41**s** | 3.6E3 | 37.8**s** | 685 | 56.4**s** | 2.2E3 | DNC |
| $N_7$ | 4 | 7 | 731 | 3.7**s** | 1 | 7.7s | 2 | 3.9**s** | 1 | 3.1**s** | 2 | 1**h**35**m** |
| $N_8$ | 3 | 8 | 478 | 6.5**s** | 3 | 40.8**s** | 2 | 3.6**s** | 3 | 3.3**s** | 2 | DNC |
| $N_9$ | 3 | 8 | 778 | 18.3**s** | 114 | 1**m**11**s** | 2 | 12.5**s** | 12 | 4.3**s** | 73 | DNC |
| $N_{10}$ | 3 | 26 | 2340 | 50**m**18**s** | $4.6E4$ | 1**h**26**m** | $6E4$ | 17**m**12**s** | 2.4E4 | 18**m**58**s** | 1.9E4 | DNC |
| $N_{11}$ | 3 | 9 | 1527 | 5**m**44**s** | 450 | 55**m**12**s** | 6.4E3 | 56.4**s** | 483 | 130.7**s** | 560 | DNC |
| $N_{12}$ | 3 | 14 | 2292 | 24**m**17**s** | 1.8E3 | 3**h**46**m** | 2.4E4 | 8**m**11**s** | 2.3E3 | 1**h**01**m** | 1.6E4 | DNC |
| $N_{13}$ | 3 | 19 | 3057 | 4**h**10**m** | 2.2E4 | 61**h**08**m** | 6.6E4 | 1**h**7**m** | 1.5E4 | 15**h**1**m** | 1.5E5 | DNC |
| $N_{14}$ | 3 | 24 | 3822 | 72**h**39**m** | 8.4E4 | 111**h**35**m** | 1.1E5 | 5**h**57**m** | 3E4 | timeout | - | DNC |
| $N_{15}$ | 3 | 127 | 6845 | 2**m**51**s** | 1 | timeout | - | 3**m**27**s** | 1 | timeout | - | DNC |

## VI. Results and Tool

We did some preliminary comparisons with a recent solver for deep neural networks called Reluplex [3]. Even though Reluplex is an SMT solver, it can be used to perform set propagation using a binary search wrapper. The preliminary comparison shows that Sherlock is much faster than Reluplex used with a binary search wrapper [1]. Another set of comparisons was using Sherlock, against a monolithic mixed integer linear programming (MILP) solver. The results of the comparison has been presented in Table I

We used Sherlock for verifying properties of various closed-loop cyberphysical systems that have neural networks as controllers [2]. We could prove that in finite number of steps, the sets did converge to the goal region, for a suite of benchmarks. This was done by set propagation using template polyhedrons.

In [11], we used the neural network model to predict blood glucose values, and then used techniques presented in Sherlock to implement an MPC controller. It was shown that the controller implemented with Sherlock, had a much better performance than the current state of the art.

## VII. Conclusions and Future Work

We are currently working on scaling our approach from a few thousands neural units to hundreds of thousands required to reason over state of the art deep learning models. We are also working on using our approach to complement black-box explaining-mining methods [9] to generate interpretable approximations of the neural network model required in applications such as medical diagnosis.

Several recent efforts have achieved significant success in tackling this important problem of verifying or robustly deep neural networks. Our own previous work [1], [2], [11] and recent papers such as [4]–[8], present a landscape of recent progress on this problem. We have made the implementation Sherlock available as an open-source tool to help accelerate the progress on formally analyzing deep learning networks.

## References

[1] Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: NASA Formal Methods. pp. 121138. (2018).

[2] Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Learning and Verification of Feedback Control System using Feedforward Neural Networks. In Proceedings of Analysis and Design of Hybrid Systems, 2018.

[3] G. Katz, C. Barrett, D. Dill, K. Julian, M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks , https://arxiv.org/abs/1702.01135

[4] F Leofante, N Narodytska, L Pulina, A Tacchella. Automated Verification of Neural Networks: Advances, Challenges and Perspectives, https://arxiv.org/abs/1805.09938

[5] W Xiang, P Musau, A A Wild, D M Lopez, N Hamilton, X Yang, J Rosenfeld, T. Johnson. Verification for Machine Learning, Autonomy, and Neural Networks Survey. https://arxiv.org/abs/1810.01989

[6] X. Huang, M. Kwiatkowska, S. Wang and M. Wu. Safety Verification of Deep Neural Networks. Computer Aided Verification, 2017.

[7] M. Mirman, T. Gehr, M. Vechev. Differentiable Abstract Interpretation for Provably Robust Neural Networks. International Conference on Machine Learning, 2018.

[8] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-Driven Approach for Assessing Robustness of Neural Networks. In: ATVA 2018

[9] S Jha , V Raman, A Pinto, T Sahai, and M Francis. On Learning Sparse Boolean Formulae For Explaining AI Decisions, NASA Formal Methods, 2017.

[10] M Abadi et al, Tensorflow : Large-Scale Machine Learning on Heterogeneous Systems, https://www.tensorflow.org/

[11] Dutta S., Kushner T., Sankaranarayanan S. Robust Data-Driven Control of Artificial Pancreas Systems Using Neural Networks. In: CMSB 2018.