NeuralVerification.jl: Algorithms for Verifying Deep Neural Networks

Changliu Liu
Robotics Institute
Carnegie Mellon University
cliu6@andrew.cmu.edu

Tomer Arnon, Christopher Lazarus & Mykel J. Kochenderfer
Stanford Intelligent Systems Laboratory
Stanford University
{tarnon,clazarus,mykel}@stanford.edu

Abstract

Deep neural networks (DNNs) are widely used for nonlinear function approximation with applications ranging from computer vision to control. Although DNNs involve the composition of simple arithmetic operations, it can be very challenging to verify whether a particular network satisfies certain input-output properties. This work introduces NeuralVerification.jl, a software package that implements methods that have emerged recently for soundly verifying such properties. These methods borrow insights from reachability analysis, optimization, and search. We present the formal problem definition and briefly discuss the fundamental differences between the implemented algorithms. In addition, we provide a pedagogical example of how to use the library.

1 Introduction

Neural networks (Goodfellow et al., 2016) have been widely used in many applications, such as image classification and understanding (He et al., 2016), language processing (Manning et al., 2014), and control of autonomous systems (Mnih et al., 2015). These networks map inputs to outputs through a sequence of layers. At each layer, the input to that layer undergoes an affine transformation followed by a nonlinear transformation. These nonlinear transformations are called activation functions, and a common example is the rectified linear unit (ReLU), which sets any negative values to zero. Although the computation involved in a neural network is quite simple, these networks can represent complex nonlinear functions by appropriately choosing the matrices that define the affine transformations.

Neural networks are being used for increasingly important tasks, and in some cases, incorrect outputs can lead to costly consequences. Traditionally, validation of neural networks has largely focused on evaluating the network on a large collection of points in the input space and determining whether the outputs are as desired. However, since the input space is effectively infinite in cardinality, it is not feasible to check all possible inputs. Even networks that perform well on a large sample of inputs may not correctly generalize to new situations and may be vulnerable to adversarial attacks (Papernot et al., 2016).

NeuralVerification.jl implements methods that are capable of formally verifying properties of DNNs. A property can be formulated as a statement that if the input belongs to some set $\mathcal{X}$, then the output will belong to some set $\mathcal{Y}$. To illustrate, in classification problems, it can be useful to verify that points near a training example belong to the same class as that example. As another example, in the control of physical systems, it can be useful to verify that the outputs from a network satisfy a safety constraint.

The verification algorithms implemented in the library are all sound, meaning that they will only report that a property holds if the property actually holds. Some of the algorithms are also complete, meaning that whenever the property holds, the algorithm will always correctly state that it holds. However, some algorithms compromise completeness since they use approximations to improve computational efficiency. A full discussion of all of the algorithms, their properties, and mathematical derivations can be found in (Liu et al., 2019).

1.1 Property definition

Consider an n-layer feedforward neural network that represents a function $\mathbf{f}$ with input $\mathbf{x} \in \mathcal{D}_\mathbf{x} \subseteq \mathbb{R}^{k_0}$ and output $\mathbf{y} \in \mathcal{D}_\mathbf{y} \subseteq \mathbb{R}^{k_n}$, i.e. $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where k_0 is the input dimension and k_n is the output dimension.

Verification entails checking whether input-output relationships of the network hold. The input constraint is imposed by a set $\mathcal{X} \subseteq \mathcal{D}_\mathbf{x}$. The corresponding output constraint is imposed by a set $\mathcal{Y} \subseteq \mathcal{D}_\mathbf{y}$. In this context, we call the sets $\mathcal{X}$ and $\mathcal{Y}$ constraints. Solving the verification problem requires showing that the following assertion holds:

$$\mathbf{x} \in \mathcal{X} \Rightarrow \mathbf{y} = \mathbf{f}(\mathbf{x}) \in \mathcal{Y}. \tag{1}$$

Following this definition, encoding problems for NeuralVerification.jl requires that the user specify the input set $\mathcal{X}$ and the output set $\mathcal{Y}$.

2 Algorithms

NeuralVerification.jl currently implements 17 algorithms which draw insight from three different categories of analysis:

2.1 Classification by technique

1. *Reachability.* These methods use layer-by-layer reachability analysis of the network. Representative methods are ExactReach (Xiang et al., 2017), MaxSens (Xiang et al., 2018), and Ai2 (Gehr et al., 2018). Some other approaches also use reachability methods (such as interval arithmetic) to compute the bounds on the values of the nodes.
2. *Optimization.* These methods use optimization to falsify the assertion. The function represented by the neural network is a constraint to be considered in the optimization problem. As a result, the optimization problem is not convex. In the primal formulation, different methods are developed to encode the nonlinear activation functions as linear constraints. Examples include NSVerify (Lomuscio & Maganti, 2017), MIPVerify (Tjeng et al., 2017), and ILP (Bastani et al., 2016). The constraints can also be simplified using the dual formulation. Representative methods are Duality (Dvijotham et al., 2018), ConvDual (Wong & Kolter, 2018), and Certify (Raghunathan et al., 2018).
3. *Search.* These methods search for a case to falsify the assertion. Search is usually combined with either reachability or optimization, as the latter two methods provide possible search directions. Representative methods for search and reachability are ReluVal (Wang et al., 2018), DLV (Huang et al., 2017), Fast-Lin (Weng et al., 2018), and Fast-Lip (Weng et al., 2018). Representative methods for search and optimization are Reluplex (Katz et al., 2017), Planet (Ehlers, 2017), BaB (Bunel et al., 2017), and Sherlock (Dutta et al., 2017). Some of these methods call Boolean satisfiability (SAT) or satisfiability modulo theories (SMT) solvers (Barrett & Tinelli, 2018) to verify ReLU networks.

2.2 Classification by Input and Output support

In general, the input set $\mathcal{X}$ and the output set $\mathcal{Y}$ can have any geometry. However, as a simplification, our implementation assumes that $\mathcal{X}$ is a polytope, and $\mathcal{Y}$ is either a polytope or the complement of a polytope. A polytope is a generalization of a polygon to higher dimensions and is defined as the intersection of a set of half-spaces. Since any compact domain can be approximated by a finite set of polytopes for any required accuracy, this formulation can be easily extended to arbitrary geometries. Moreover, the complement of a polytope allows the encoding of unbounded sets.

NeuralVerification.jl supports the following subclasses of polytopes:[1]

- *Halfspace-polytope* (or *H-Polytope*), which represents polytopes using a set of linear inequality constraints
$$C\mathbf{x} \leq \mathbf{d}, \tag{2}$$
where $C \in \mathbb{R}^{n \times k_0}$, $\mathbf{d} \in \mathbb{R}^k$, and k is the number of inequality constraints that are defining the polytope. A point $\mathbf{x}$ is in the polytope if and only if $C\mathbf{x} \leq \mathbf{d}$ is satisfied.
- *Hyperrectangle*, which corresponds to a high-dimensional rectangle, defined by
$$|\mathbf{x} - \mathbf{c}| \leq \mathbf{r}, \tag{3}$$
where $\mathbf{c} \in \mathbb{R}^{k_0}$ is the center of the hyperrectangle and $\mathbf{r} \in \mathbb{R}^{k_0}$ is the radius of the hyperrectangle.
- *Halfspace*, which is represented by a single linear inequality constraint
$$\mathbf{c}^T\mathbf{x} \leq d, \tag{4}$$
where $\mathbf{c} \in \mathbb{R}^{k_0}$ and $d \in \mathbb{R}$.

2.3 Description of Algorithms

The following table summarizes the characteristics of the algorithms implemented in NeuralVerification.jl.

Table 1: List of existing methods. The entries under "approach" summarize the key idea of each method. Note that for DLV, the original implementation is complete but may not be sound, while our implementation is sound but not complete. HR, HP, HS, and PC stand for Hyperrectangle, H-Polytope, Halfspace, and Polytope-Complement, respectively. Since H-Polytopes are more general than Halfspaces and Hyperrectangles, any algorithm that supports the former should implicitly support the latter two as well.

Method Name	Approach	Input/Output	Completeness
ExactReach (Xiang et al., 2017)	Exact Reachability	HP/HP	✓
AI2 (Gehr et al., 2018)	Split and Join	HP/HP	×
MaxSens (Xiang et al., 2018)	Interval Arithmatic	HP/HP	×
NSVerify (Lomuscio & Maganti, 2017)	Naive MILP	HR/PC	✓
MIPVerify (Tjeng et al., 2017)	MILP with bounds	HR/PC	✓
ILP (Bastani et al., 2016)	Iterative LP	HR/PC	×
Duality (Dvijotham et al., 2018)	Lagrangian Relaxation	HR/HS	×
ConvDual (Wong & Kolter, 2018)	Convex Relaxation	HR(uniform)/HS	×
Certify (Raghunathan et al., 2018)	Semidefinite Relaxation	HR/HS	×
Fast-Lin (Weng et al., 2018)	Network Relaxation	HR/HS	×
Fast-Lip (Weng et al., 2018)	Lipschitz Estimation	HR/HS	×
ReluVal (Wang et al., 2018)	Symbolic Interval	HR/HR	✓
DLV (Huang et al., 2017)	Search in Hidden Layers	HR/HR(1-D)	✓*
Sherlock (Dutta et al., 2017)	Local and Global Search	HR/HR(1-D)	×
BaB (Bunel et al., 2017)	Branch and Bound	HR/HR(1-D)	×
Planet (Ehlers, 2017)	Satisfiability (SAT)	HR/PC	✓
Reluplex (Katz et al., 2017)	Simplex	HR/PC	✓

3 NeuralVerification.jl

NeuralVerification.jl is a software library implemented in Julia for verifying deep neural networks. At the moment, all of the algorithms are written under the assumption of feedforward, fully-connected networks. While a few of the methods can handle arbitrary activation functions, most

[1]Our implementation relies on geometric definitions provided by LazySets.jl, a Julia package for calculus with convex sets. That library can be found at `https://github.com/JuliaReach/LazySets.jl`.

are restricted to ReLU. The library is open source under the MIT license and can be found at `https://github.com/sisl/NeuralVerification.jl`.

We now provide a minimal tutorial to illustrate how to get started using NeuralVerification.jl

3.1 Installation

To download this library, clone it from the julia package manager:

```
(v1.0) pkg> add https://github.com/sisl/NeuralVerification.jl.git
```

3.2 Usage

A `Problem` consists of a network to be tested, which can be loaded using the NNet format (Julian, 2016) or created using the `Network` object of NeuralVerification.jl, a set representing the input $\mathcal{X}$, and another set representing the output $\mathcal{Y}$.

Note that the input and output sets may be of different types for different solvers as noted in Section 1. In this case, we use hyperrectangles to represent out sets, and therefore implicitly restrict ourselves to only those solvers that support them.

```
using NeuralVerification, LazySets
nnet = read_nnet("examples/networks/small_nnet.nnet");
input   = Hyperrectangle(low = [-1.0], high = [1.0]);
output  = Hyperrectangle(low = [-1.0], high = [100.0]);
problem = Problem(nnet, input, output);
```

For this example, we initialize an instance of `MaxSens` with a custom resolution, which determines how small the input set is partitioned for the search. For more information about MaxSens (and other solvers,) please refer to the library's documentation at: `https://sisl.github.io/NeuralVerification.jl/latest/`, as well as to (Liu et al., 2019). Note that often, since the characteristics of the Problem dictate the input and output constraint types, this will also dictate which solvers are most appropriate.

```
solver = MaxSens(resolution = 0.3);
```

To solve the Problem, we simply call the solve function:

```
result = solve(solver, problem);
>> ReachabilityResult(:holds)
```

In this case, the solver returns a `ReachabilityResult` and indicates that the property holds. A result status of `:holds` means that the specified input-output relationship is satisfied. In terms of the problem formulation presented in Section 1.1, we can say that no input in the input region can produce a point that is outside of the specified output region.

Conversely, a result status of `:violated` would mean that the input-output relationship is not satisfied, i.e. that the property being tested for in the network does not hold. In the following example, the output set is altered so that the property is violated. When a property is violated, the solver returns the cause of the violation for reference. Since MaxSens is a reachability based method, it returns the calculated reachable output set:

```
output = Hyperrectangle(low = [-10.0], high = [0.0]);
problem = Problem(nnet, input, output);
result = solve(solver, problem);
result.status
>> :violated
result.reachable
>> 1-element Array{Hyperrectangle{Float64},1}:
    Hyperrectangle{Float64}([54.5], [24.0])
```

A status of `:unknown` is also possible for some algorithms.

4 Conclusion

We have introduced NeuralVerification.jl along with the minimal mathematical definitions required to specify a Neural Network Verification problem. This work introduces a mathematical framework for verifying neural networks, classifies existing methods under this framework, and provides a library for their implementation.

References

Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pp. 305–343. Springer, 2018.

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 (NIPS)*, pp. 2613–2621, 2016.

Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. A unified view of piecewise linear neural network verification. *ArXiv*, (1711.00455), 2017.

Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan, and Ashish Tiwari. Output range analysis for deep neural networks. *ArXiv*, (1709.09130), 2017.

Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 162–171, Corvallis, Oregon, 2018. AUAI Press.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pp. 269–286. Springer, 2017.

Timon Gehr, Matthew Mirman, Dana Drashsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pp. 3–29. Springer, 2017.

Kyle Julian. NNet: nnet file format for fully connected relu networks, 2016. URL `https://github.com/sisl/NNet`.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pp. 97–117. Springer, 2017.

Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *CoRR*, abs/1903.06758, 2019. URL `http://arxiv.org/abs/1903.06758`.

Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *ArXiv*, (1706.07351), 2017.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60, 2014.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 372–387. IEEE, 2016.

Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=Bys4ob-Rb`.

Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *ArXiv*, (1711.07356), 2017.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi`.

Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for ReLU networks. In Jennifer Dy and Andreas Krause (eds.), *International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5276–5285, Stockholmsmassan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/weng18a.html`.

Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In Jennifer Dy and Andreas Krause (eds.), *International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research, pp. 5286–5295, Stockholmsmassan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/wong18a.html`.

W. Xiang, H. Tran, and T. T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5777–5783, Nov 2018. ISSN 2162-237X. doi: 10.1109/TNNLS.2018.2808470.

Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Reachable set computation and safety verification for neural networks with relu activations. *ArXiv*, (1712.08163), 2017.