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 \( f \) with input \( x \in \mathcal{D}_x \subseteq \mathbb{R}^{k_0} \) and output \( y \in \mathcal{D}_y \subseteq \mathbb{R}^{k_n} \), i.e. \( y = f(x) \), 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 \( \mathcal{X} \subseteq \mathcal{D}_x \). The corresponding output constraint is imposed by a set \( \mathcal{Y} \subseteq \mathcal{D}_y \). In this context, we call the sets \( \mathcal{X} \) and \( \mathcal{Y} \) constraints. Solving the verification problem requires showing that the following assertion holds:

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

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:

- **Halfspace-polytope** (or H-Polytope), which represents polytopes using a set of linear inequality constraints
  \[ Cx \leq d, \tag{2} \]
  where \( C \in \mathbb{R}^{n \times k_0} \), \( d \in \mathbb{R}^k \), and \( k \) is the number of inequality constraints that are defining the polytope. A point \( x \) is in the polytope if and only if \( Cx \leq d \) is satisfied.

- **Hyperrectangle**, which corresponds to a high-dimensional rectangle, defined by
  \[ |x - c| \leq r, \tag{3} \]
  where \( c \in \mathbb{R}^{k_0} \) is the center of the hyperrectangle and \( r \in \mathbb{R}^{k_0} \) is the radius of the hyperrectangle.

- **Halfspace**, which is represented by a single linear inequality constraint
  \[ c^T x \leq d, \tag{4} \]
  where \( c \in \mathbb{R}^{k_0} \) and \( d \in \mathbb{R} \).

### 2.3 Description of Algorithms

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

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

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...
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 1.1. 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


