Engineering A Verifier for Deep Neural Networks

ThanhVu (Vu) Nguyen

October 31, 2024 (latest version available on Github)

Preface

Having been involved in PhD admission committees for many years, I've realized that many **international** students, especially those in smaller countries or less well-known universities, lack a clear understanding of the Computer Science PhD admission process at US universities. This confusion not only discourages students from applying but also creates the perception that getting admitted to a CS PhD program in the US is difficult compared to other countries.

So I want to share some details about the admission process and advice for those who are interested in applying for a **PhD in Computer Science in the US**. Originally, this document was intended for international students, but I have expanded it to include information that might also be useful for *US domestic students*. Moreover, while this is primarily intended for students interested in CS, it might be relevant to students from various STEM (Science, Technologies, Engineering, and Mathematics) disciplines. Furthermore, although many examples are specifics for schools that I and other contributors of this document know about, the information should be generalizable to other R1[1] institutions in the US.

This information can also help **US faculty and admission committee** gain a better understanding of international students and their cultural differences. By recognizing and leveraging these differences, CS programs in the US can attract larger and more competitive application pools from international students.

I wish you the best of luck. Happy school hunting!

> This document will be updated regularly to reflect the latest information and updates in the admission process. Its latest version is available at
>
> nguyenthanhvuh.github.io/phd-cs-us/demystify.pdf,
>
> and its LaTeX source is also on GitHub. If you have questions or comments, feel free to create new GitHub issues or discussions.

[1] An R1 institution in the US is a research-intensive university with a high level of research activity across various disciplines. Currently, 146 (out of 4000) US universities are classified as R1.

Contents

1 Basic of Neural Network **4**
- 1.1 Affine Transformation . . . 4
- 1.2 Activation Functions . . . 5
- 1.3 Example . . . 6
- 1.4 Types of Neural Networks . . . 6
- 1.5 Properties of Neural Networks . . . 7
  - 1.5.1 Challenges . . . 7

2 Verification of Neural Networks **8**
- 2.1 The Neural Network Verification Problem and Its Formalization . . . 8
- 2.2 Satisfiability and Activation Pattern Search . . . 8
- 2.3 Activation Pattern Search . . . 9
- 2.4 Complexity . . . 9

3 Common Search Algorithms **10**
- 3.1 Symbolic Search and Constraint solving . . . 10
- 3.2 Branch-and-Bound (BnB) . . . 10

4 Constraint Solving **13**
- 4.1 SMT . . . 13
- 4.2 MILP . . . 13

5 Abstraction **14**
- 5.1 Interval . . . 14
- 5.2 Zotope . . . 14
- 5.3 Polytope . . . 14

6 Popular Techniques and Tools **15**

7 Verifying the Verifiers **16**

8 Conclusion **17**

Chapter 1

Basic of Neural Network

A *neural network* (**NN**) [6] consists of an input layer, multiple hidden layers, and an output layer. Each layer has a number of neurons, each connected to neurons in the next layer through a predefined set of weights (derived by training the network with data). A *Deep Neural Network* (**DNN**) is an NN with two or more hidden layers.

The output of an NN is obtained by iteratively computing the values of neurons in each layer. The value of a neuron in the input layer is the input data. The value of a neuron in the hidden layers is computed by applying an *affine transformation* (§1.1) to values of neurons in the previous layers, then followed by an *activation function* (§1.2) such as ReLU and Sigmoid. The value of a neuron in the output layer is computed similarly but may skip the activation function.

1.1 Affine Transformation

The affine transformation (AF) of a neuron is the sum of the products of the weights of the incoming edges and the values of the neurons in the previous layer, plus the bias of the neuron. More specifically, the AF of a neuron y with weights $w_1, \ldots, w_n$ and bias b and the values of neurons in the previous layer $v_1, \ldots, v_n$ is $w_1 v_1 + \cdots + w_n v_n + b$.

For example, the AF of a neuron x_3 in Fig. 1.1 with (incoming arrows) weights $-0.5, 0.5$ and bias 1.0 and the values of neurons in the previous layer x_1, x_2 is $-0.5x_1 + 0.5x_2 + 1.0$.

For DNN verification, AF is straightforward to reason about because it is a linear function. However, AFs are often followed by non-linear activation functions, described next in §1.2, which make the verification problem more challenging.

1.2 Activation Functions

Several popular activation functions used in DNNs include ReLU, Sigmoid, Tanh, and Softmax. All of these are non-linear[1] functions that introduce non-linearity to the network, allowing it to learn complex patterns in the data.

- ReLU (Rectified Linear Unit): ReLU returns 0 if the input is less than zero, and the input itself otherwise. A ReLU activated neuron is said to be *active* if its input value is greater than zero and *inactive* otherwise. ReLU is the most popular activation function in DNNs.

$$ReLU(x) = \max(x, 0)$$

- Sigmoid: Sigmoid is a smooth function that maps any real value to the range (0,1). It is often used in the output layer of a binary classification problem.

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

- Tanh: Tanh is similar to the sigmoid function but maps any real value to the range (-1,1). It is often used in the output layer of a multi-class classification problem.

$$Tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

- Softmax: Softmax is a generalization of the sigmoid function that maps any real value to the range (0,1) and ensures that the sum of the output values is 1. It is often used in the output layer of a multi-class classification problem.

$$Softmax(x)_i = \frac{e^{x}i}{\sum_{j=1}^{n} e^{x_j}}$$

For DNN verification, these non-linear activation functions make verification difficult because it introduces multiple possible outcomes for any input, making it hard to reason about the output of the network. For example, ReLU has two possible outputs for any input: 0 if the input is less than zero, and the input itself otherwise, and Sigmoid has a smooth curve with infinite possible outputs for any input.

[1] Non-linear means that the output of the function is not a linear combination of its inputs.

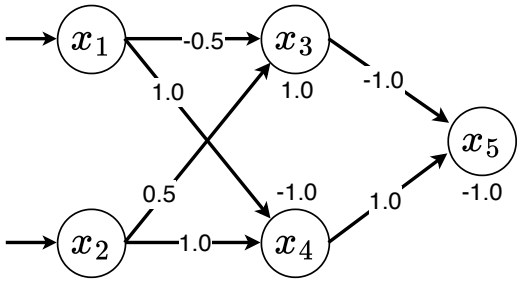


Fig. 1.1: An FNN with ReLU.

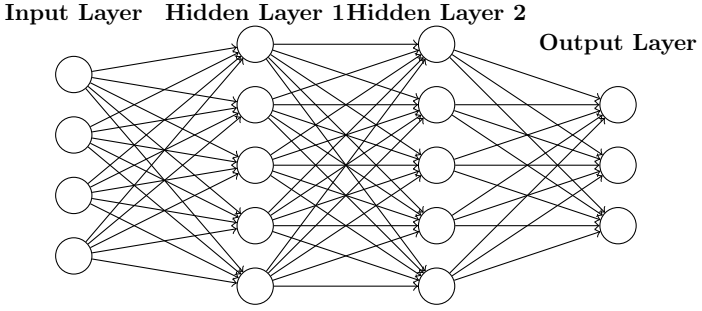
1.3 Example

Fig. 1.1 shows a simple DNN with two inputs x_1, x_2, two hidden neurons x_3, x_4, and one output x_5. The weights of a neuron are shown on its incoming edges , and the bias is shown above or below each neuron. The outputs of the hidden neurons are computed the affine transformation and ReLU, e.g., $x_3 = ReLU(-0.5x_1 + 0.5x_2 + 1.0)$. The output neuron is computed with just the affine transformation, i.e., $x_5 = -x_3 + x_4 - 1$.

1.4 Types of Neural Networks

Feed-Forward Network (FFN) In an FFN information flows in one direction, from the input layer to hidden layers to the output layer (and thus no cycle).

A fully connected feed-forward neural network (FNN), shown below, is an FFN where every neuron in a layer is connected to every neuron in the next layer.



Convolutional Neural Networks (CNNs) CNNs, often used in image recognition and classification, consist of neurons that have learnable weights and biases. Each neuron receives several inputs, takes a weighted sum over them, passes it through an activation function, and responds with an output.

Recurrent Neural Networks (RNNs) RNNs, often used in natural language process and speech recognition, are designed to recognize patterns in sequences of data. RNNs have *loops* in them, allowing information to be sent forward and backward.

Residual Networks Residual Networks (ResNets) are a type of neural network that is often used in image recognition and classification. ResNets introduce skip connections that allow the gradient to flow directly through the network, making it easier to train deep networks.

1.5 Properties of Neural Networks

Similar to software programs, neural networks have desirable properties to ensure the network behaves as expected. These could be specific to the applications modeled by the network, e.g., safety properties in a collision avoidance system or general properties that are desired by all networks, e.g., robustness to adversarial attacks.

Robustness Properties *Robustness*, a desirable property for all networks, ensures that small perturbations in the input data do not cause major changes in the output of the network. For example, if a few pixels in an image are changed, the network should still classify the image correctly. *Adversarial attacks* are a common way to test the robustness of a neural network. In an adversarial attack, an attacker makes small changes to the input data to cause the network to misclassify the data.

Local robustness refers to robustness of a neural network within a *small neighborhood or region* of the input data. In contrast, *global* robustness refers to robustness of a network across the *entire input space*. Global robustness is harder to achieve than local robustness, as it requires the network to be robust to all possible inputs.

ϵ-robustness A neural network is ϵ-robust if the difference between any two inputs x and x' is within a small range ϵ, the output f of the network does not change significantly (or remain the same), i.e., $\|x - x'\| \leq \epsilon \implies f(x) \approx f(x')$.

Safety Properties Safety properties are specific to the application modeled by the network. For example, a safety property in a collision avoidance system might be that if the intruder is distant and significantly slower than us, then we stay below a certain threshold, i.e., $d_{intruder} > d_{threshold} \wedge v_{intruder} < v_{threshold} \implies v_{us} < v_{threshold}$.

1.5.1 Challenges

Formalization

Expressiveness

Chapter 2

Verification of Neural Networks

2.1 The Neural Network Verification Problem and Its Formalization

DNN Verification Given a DNN N and a property ϕ, the *DNN verification problem* asks if ϕ is a valid property of N. Typically, ϕ is a formula of the form $\phi_{in} \Rightarrow \phi_{out}$, where ϕ_{in} is a property over the inputs of N and ϕ_{out} is a property over the outputs of N. A DNN verifier attempts to find a *counterexample* input to N that satisfies ϕ_{in} but violates ϕ_{out}. If no such counterexample exists, ϕ is a valid property of N. Otherwise, ϕ is not valid and the counterexample can be used to retrain or debug the DNN [7].

Example A valid property for the DNN in Fig. 1.1 is that the output is $x_5 \leq 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$. An invalid property for this network is that $x_5 > 0$ for those similar inputs. A counterexample showing this property violation is $\{x_1 = -1, x_2 = 2\}$, from which the network evaluates to $x_5 = -3.5$. Such properties can capture *safety requirements* (e.g., a rule in an collision avoidance system in [8,11] is "if the intruder is distant and significantly slower than us, then we stay below a certain threshold") or *local robustness* [9] conditions (a form of adversarial robustness stating that small perturbations of a given input all yield the same output).

2.2 Satisfiability and Activation Pattern Search

As with traditional software, DNN verification is often represented as a satisfiability problem. We thus need to define a formula to represent the network. Typically this formula is a conjunction of constraints representing the affine transformation and activation function of each neuron in the network. For a network with L layers, N

neurons per layer, and ReLU activations this formula is:

$$\alpha = \bigwedge_{\substack{i\in[1,L]\\ j\in[1,N]}} v_{i,j} = \max\Big(\sum_{k\in[1,N]} (w_{i-1,j,k}\cdot v_{i-1,j}) + b_{i,j}, 0\Big)$$

With this definition DNN verification can be formulated as checking the satisfiability of:

$$\alpha \wedge \phi_{in} \wedge \neg\phi_{out} \tag{2.1}$$

If Eq. 2.1 is unsatisfiable, ϕ is a valid property of $\mathcal{N}$. Otherwise, ϕ is not valid (and the counterexample of the original problem is any satisfying assignment that makes Eq. 2.1 true).

2.3 Activation Pattern Search

For the widely-used ReLU activation problem, the DNN verification problem becomes a search for *activation patterns*, i.e., boolean assignments representing activation status of neurons, that lead to satisfaction the formula in Eq. 2.1.

Modern DNN verification techniques [1–5,10,13,14] all adopt this idea and search for satisfying activation patterns.

2.4 Complexity

Chapter 3

Common Search Algorithms

3.1 Symbolic Search and Constraint solving

3.2 Branch-and-Bound (BnB)

Most modern DNN verifiers adopt the branch-and-bound (BnB) approach to search for activation patterns for DNN verification. A BnB search consists of two main components: (branch) splitting into the problem smaller subproblems by using *neuron splitting*, which decides boolean values representing neuron activation status, and (bound) using abstraction and LP solving to approximate bounds on neuron values to determine the satisfiability of the partial activation pattern formed by the split.

Alg. 1 shows $\texttt{BaB}_{\texttt{NV}}$, a reference BnB architecture [12] for modern DNN verifiers. $\texttt{BaB}_{\texttt{NV}}$ takes as input a ReLU-based DNN $\mathcal{N}$ and a formulae $\phi_{in} \Rightarrow \phi_{out}$ representing the property of interest. $\texttt{BaB}_{\texttt{NV}}$ iterates between (i) branching (line 6), which *decides* (assigns) an activation status value for a neuron, and (ii) bounding (line 5), which *deduces* or checks the feasibility of the current activation pattern.

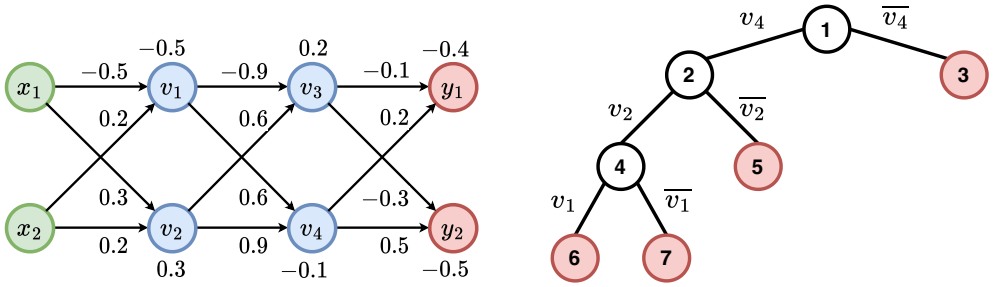
Example Fig. 3.1a illustrates a DNN and how $\texttt{BaB}_{\texttt{NV}}$ determines unsatisfiability (i.e., verifies the problem) and generates the unsat proof in Fig. 3.1b.

First, $\texttt{BaB}_{\texttt{NV}}$ initializes the activation pattern set `ActPatterns` with an empty activation pattern $\emptyset$. Then $\texttt{BaB}_{\texttt{NV}}$ enters a loop (line 3-line 9) to search for a satisfying assignment or a proof of unsatisfiability. In the first iteration, $\texttt{BaB}_{\texttt{NV}}$ selects the only available activation pattern $\emptyset \in$ `ActPatterns`. It calls `Deduce` to check the feasibility of the problem based on the current activation pattern. `Deduce` uses abstraction to approximate that from the input constraints the output values are feasible for the given network. Since `Deduce` cannot decide infeasibility, $\texttt{BaB}_{\texttt{NV}}$ randomly selects a neuron to split (`Decide`). Let us assume that it chooses v_4 to split, which essentially means the problem is split into two independent subproblems: one with v_4 active and the other with v_4 inactive. $\texttt{BaB}_{\texttt{NV}}$ then adds $\{v_4\}$ and $\{\overline{v_4}\}$ to `ActPatterns`.

Alg. 1. The BaB$_{\text{NV}}$ algorithm with proof generation.

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$
output : unsat if property is valid, otherwise (sat, cex)

```
1  ActPatterns ← {∅} // initialize verification problems
2  proof ← { } // initialize proof tree
3  while ActPatterns do // main loop
4      σ_i ← Select(ActPatterns) // process problem i-th
5      if Deduce(N, φ_in, φ_out, σ_i) then
6          (cex, v_i) ← Decide(N, φ_in, φ_out, σ_i)
7          if cex then // found a valid counter-example
8              return (sat, cex)
           // create new activation patterns
9          ActPatterns ← ActPatterns ∪ {σ_i ∧ v_i ; σ_i ∧ ¬v_i}
10 return unsat
```



(a) (b)

Fig. 3.1: (a) A simple DNN. (b) A proof tree produced verifying the property $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow (y_1 > y_2)$.

In the second iteration, BaB$_{\text{NV}}$ has two subproblems (that can be processed in parallel). For the first subproblem with v_4, `Deduce` cannot decide infeasibility, so it selects v_2 to split. It then conjoins v_4 with v_2 and then with $\overline{v_2}$ and adds both conjuncts to `ActPatterns`. For the second subproblem with $\overline{v_4}$ inactive, `Deduce` determines that the problem is unsatisfiable and BaB$_{\text{NV}}$ saves the node v_4 to the proof tree, as node 3, to indicate one unsatisfiable pattern, i.e., whenever the network has v_4 being inactive, the problem is unsatisfiable.

In the third iteration, BaB$_{\text{NV}}$ has two subproblems for $v_4 \wedge v_2$ and $v_4 \wedge \overline{v_2}$. For the first subproblem, `Deduce` cannot decide infeasibility, so it selects v_1 to split. It then conjoins v_1 and then $\overline{v_1}$ to the current activation pattern and adds them to `ActPatterns`. For the second subproblem, `Deduce` determines that the problem is unsatisfiable and BaB$_{\text{NV}}$ saves the node $v_4 \wedge \overline{v_2}$ to the proof tree, as node 5.

In the fourth iteration, BaB$_{\text{NV}}$ has two subproblems for $v_4 \wedge v_2 \wedge v_1$ and $v_4 \wedge v_2 \wedge \overline{v_1}$. Both subproblems are determined to be unsatisfiable, and BaB$_{\text{NV}}$ saves them to the

proof tree as nodes 6 and 7, respectively.

Finally, $\texttt{BaB}_{\texttt{NV}}$ has an empty `ActPatterns`, stops the search, and returns `unsat` and the proof tree.

Chapter 4

Constraint Solving

4.1 SMT

4.2 MILP

Chapter 5

Abstraction

5.1 Interval

An interval abstraction is a simple and widely-used abstraction technique in DNN verification. It approximates the range of values that a neuron can take by using a lower and upper bound $[l, u]$. The interval abstraction is efficient and easy to implement, but it can be imprecise, especially for non-linear activation functions like ReLU.

Example : For the DNN in Fig. 1.1, the interval abstraction for the input layer is $x_1 \in [-1, 1]$ and $x_2 \in [-2, 2]$. The interval abstraction for the hidden layer is $x_3 \in [0, 2]$ because $x_3 = [ReLU(-0.5x_1+0.5x_2+1.0), ReLU(-0.5x_1+0.5x_2+1.0)] = [ReLU(-0.5\times-1+0.5\times-2+1.0), ReLU(-0.5\times1+0.5\times2+1.0)] = [0, 2]$. Similarly, $x_4 \in [0, 2]$. The interval abstraction for the output layer is $x_5 \in [-3, 3]$.

5.2 Zotope

5.3 Polytope

Chapter 6

Popular Techniques and Tools

Chapter 7

Verifying the Verifiers

Chapter 8

Conclusion

Bibliography

[1] S. Bak. nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement. In *NASA Formal Methods Symposium*, pages 19–36. Springer, 2021.

[2] R. Bunel, P. Mudigonda, I. Turkaslan, P. Torr, J. Lu, and P. Kohli. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020.

[3] H. Duong, T. Nguyen, and M. Dwyer. A DPLL(T) Framework for Verifying Deep Neural Networks. *arXiv preprint arXiv:2307.10266*, 2024.

[4] H. Duong, D. Xu, T. Nguyen, and M. Dwyer. Harnessing Neuron Stability to Improve DNN Verification. *Proceedings of the ACM on Software Engineering (PACMSE)*, FSE:to appear, 2024.

[5] C. Ferrari, M. N. Mueller, N. Jovanović, and M. Vechev. Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound. In *International Conference on Learning Representations*, 2022.

[6] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. https://www.deeplearningbook.org, last accessed October 31, 2024.

[7] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer, 2017.

[8] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[9] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Towards proving the adversarial robustness of deep neural networks. *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV), pp. 19-26*, 2017.

[10] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, et al. The marabou framework for verification

and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[11] M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos. Next-generation airborne collision avoidance system. Technical report, Massachusetts Institute of Technology-Lincoln Laboratory Lexington United States, 2012.

[12] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo. Consolidating a process for the design, representation, and evaluation of reference architectures. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 143–152. IEEE, 2014.

[13] OVAL-group. OVAL - Branch-and-Bound-based Neural Network Verification, 2023. https://github.com/oval-group/oval-bab.

[14] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Robustness Verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.