Enhancing Symbolic Execution for More Efficient Neural Network Verification

Brian Ozawa Burns, Anushka Lodha, Erin DeLong

November 2021

1 Research Context and Problem Statement

In recent years, neural networks have become increasingly important due to their role in artificial intelligence and machine learning. Artificial intelligence has had a huge impact on many fields, from automation to medicine– it has paved the way for a safer future through self-driving cars, and has significantly improved healthcare by making diagnoses more accurate through image analysis. For example, neural networks have been used to estimate the recurrence rate of tumors in patients with breast cancer after judging data from lymphatic node positive patients [7]. Evidently, neural networks have many important applications.

Neural networks are modeled off the human brain in the way connections are made, and are composed of layers of interconnected nodes. An input is passed to the first layer of the network, and is multiplied by a weight depending on the node. The input is then passed through an activation function, which determines the values used as input for the next layer. This continues until the network reaches the last layer and produces an output. To ensure that a neural network is valid–that it performs as expected–it is necessary to verify it. To test a neural network is to feed the network an input with a known output in order to see if the output generated by the network matches the expected output. However, even after testing a network, we cannot provide any guarantees about its behavior unless we can test every input. To test all possible inputs is the process of network verification.

A single network can be interpreted in various ways based on its numerical precision. Figure 1 demonstrates three types: idealized (real-valued) networks, floating-point networks, and quantized networks. Idealized networks are those where each value in the network has infinite precision–in reality, this is impossible, but there has been research dedicated to verifying this abstraction. In floating-point networks, values are rounded based on the "IEEE 754 standard," but verification methods are highly inefficient [5]. Finally, quantized, or fixed-point, networks have become increasingly common–these networks have weights

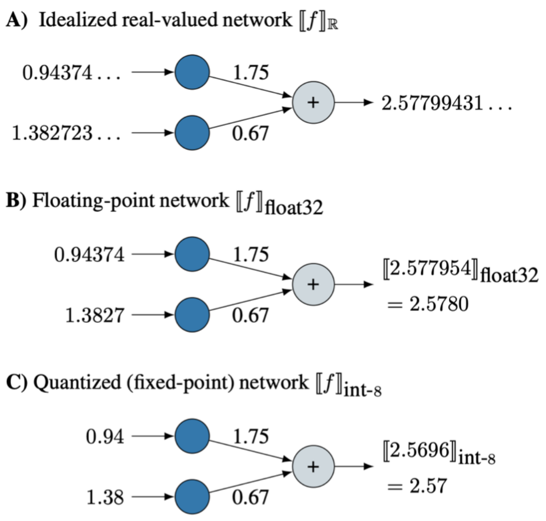

A) Idealized real-valued network $[\![f]\!]_{\mathbb{R}}$
0.94374...
1.382723...
1.75
0.67
+
2.57799431...
B) Floating-point network $[\![f]\!]_{\text{float32}}$
0.94374
1.3827
1.75
0.67
+
$[\![2.577954]\!]_{\text{float32}}$
$= 2.5780$
C) Quantized (fixed-point) network $[\![f]\!]_{\text{int-8}}$
0.94
1.38
1.75
0.67
+
$[\![2.5696]\!]_{\text{int-8}}$
$= 2.57$

Figure 1: This diagram depicts three representations of the same network. Note that these values are hypothetical; computations are actually run with base-2 representations of these decimal numbers [5].

with reduced precision; by lessening the memory needed to represent information, computation efficiency increases [6].

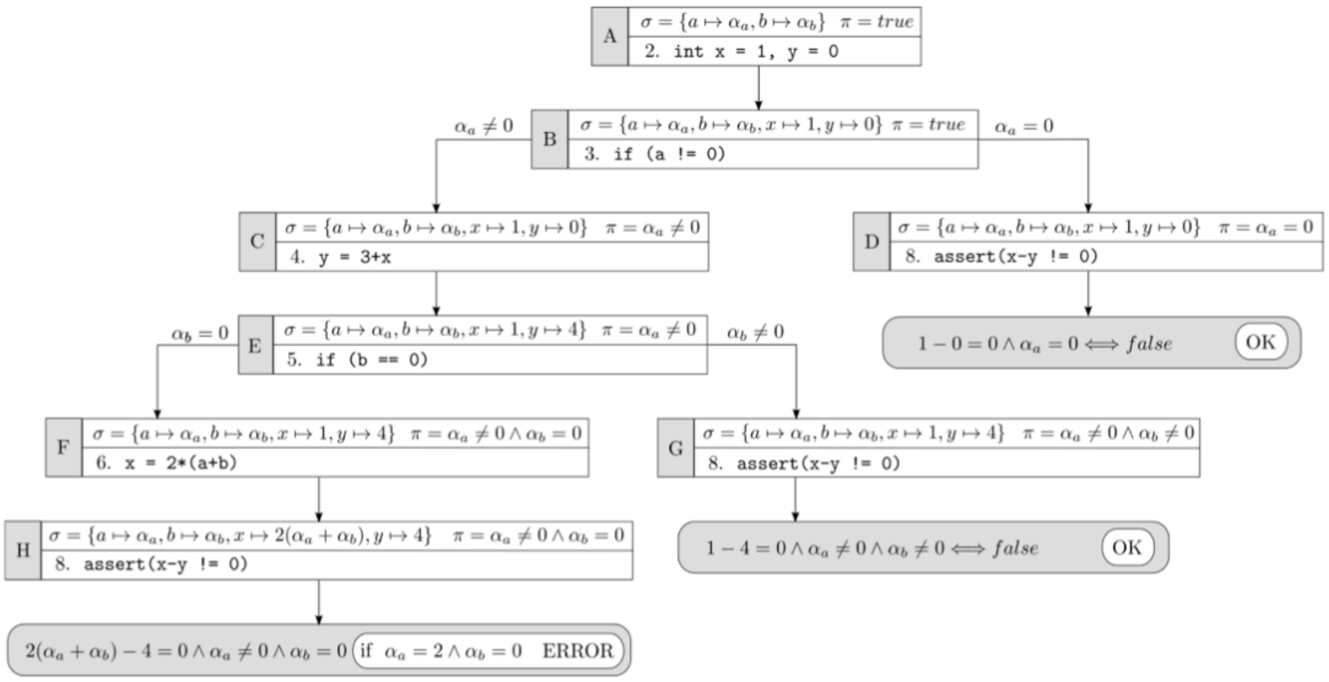
Scalability is one of the main issues that arises when attempting to verify a neural network. Despite the advantages of quantized networks, current verification methods cannot handle large quantized neural networks efficiently–it has been shown that verification is PSPACE-hard, which is very complex. Therefore we wish to create new verification techniques to expedite this process [5].

To verify a neural network, one could, in theory, run every possible input through a neural network. However, there are often far too many possible inputs to test, and verifying a neural network with brute force is usually unrealistic. One method of verifying a neural network is symbolic execution, which is the method our proposed solution relies on.

Like concrete inputs, symbolic execution tests a program; however, it abstractly expresses concrete inputs using symbols like variables and inequalities [5]. These variables can then be used as inputs to the program and can trace every possible path that the program could take. These paths are all different because there are many different ways a program can execute based on a given input. For example, some inputs will cause programs to enter if-statements, while others will not. The specific method by which a symbolic execution engine accomplishes this is that it maintains a formula, also called a constraint, that describes the

```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.         y = 3+x;
5.         if (b == 0)
6.            x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```

Figure 2: [2]



A $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$
2. int x = 1, y = 0
$\alpha_a \neq 0$
B $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$
3. if (a != 0)
$\alpha_a = 0$
C $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$
4. y = 3+x
D $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a = 0$
8. assert(x-y != 0)
$1 - 0 = 0 \wedge \alpha_a = 0 \Longleftrightarrow false$ OK
$\alpha_b = 0$
E $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0$
5. if (b == 0)
$\alpha_b \neq 0$
F $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
6. x = 2*(a+b)
G $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b \neq 0$
8. assert(x-y != 0)
H $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
8. assert(x-y != 0)
$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \Longleftrightarrow false$ OK
$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$ if $\alpha_a = 2 \wedge \alpha_b = 0$ ERROR

Figure 3: The symbolic execution tree branches at every condition in Figure 2. σ represents the symbolic store that matches variables with expressions of concrete or symbolic values, while π denotes the constraints for the path [2].

conditions satisfied for each explored control flow path (each possible path that an input could take through a program). At every step in the symbolic execution tree, we use a Satisfiability Modulo Theory (SMT) solver, specifically Microsoft Research's Z3, to check whether there are any violations of the property along the path. The SMT solver essentially verifies whether the formula can be satisfied by substituting the program's symbolic arguments with concrete values [2]. If the formula can be satisfied, the SMT solver returns one of the possible values [2].

Model counting is another technique that we will use in our research. The one we will be using is the Automata-Based Model Counter, also called the ABC model counter [1]. A model counter is used to count how many possible concrete inputs satisfy a particular constraint. However, model counts do not return any specific values that will satisfy the constraints.

Model counting is beneficial as it allows for quantitative verification, which determines how often a property is satisfied. This is useful because it offers a probabilistic guarantee of the behavior of the network. Such a probability is necessary when, for example, testing the safety of autonomous vehicles that rely on neural networks [8]. A very simplified way of looking at it is, rather than knowing whether a car will crash or not, it is more helpful to know how likely it is to crash. Currently, quantitative verification techniques exist for binary and floating-point neural networks, but do not exist with greater than binary precision [3, 4, 9]. We hypothesize that we can implement this and use model counting to decrease the runtime of symbolic execution.

Overview of our proposal:

Symbolic execution is an effective verification technique, and we will attempt to optimize it through model counting so that it can handle larger quantized neural networks in a more reasonable amount of time; this will help us achieve our goal of making verification more scalable. In addition to using model counting to optimize quantized network verification, we will attempt to determine a way to know for which situations using brute force would be more efficient than running symbolic execution. Sometimes, if there are few enough inputs, it would be possible to get through all of them using brute force in a reasonable amount of time. If we can find this threshold of the number of inputs for which brute force is faster, then we could test the network using the concrete inputs rather than following every possible path in the symbolic execution tree that may or may not lead to a solution.

Finally, we will attempt to implement increased flexibility of user input for symbolic execution. Through methods that will be discussed in the following section, adding flexibility may also reduce the runtime of verification and, ultimately, make it more scalable.

2 Proposed Solution

There are two solutions that could be implemented in parallel and would potentially make network verification more efficient and flexible.

2.1 Optimization

The first solution involves running a model count before symbolic execution to find the maximum number of solutions for a set of constraints. This leads to two possible optimizations.

First, by knowing the maximum number of solutions prior to searching, the solver will be able to determine if it is appropriate to stop and will not search the paths of the remaining branches. For instance, if the model counter returns ten total solutions, then the solver will stop after finding the tenth solution, ignoring the other branches which could not possibly be satisfiable because this would mean there are more than ten solutions.

Second, with constraints that allow for fewer inputs, verification with brute force is actually more efficient than verification with symbolic execution. Hence, we would also like to look into the maximum number of solutions (determined by model counting) for which brute force techniques are more efficient than symbolic execution. Then, for networks that would be more efficiently verified using brute force, we will use this technique instead of symbolic execution to speed up verification.

2.2 Flexibility

When a neural network is given input, the result is a set of probabilities representing the chance of each distinct output. Currently, the verifier allows for constraints where variables are compared to constant values. By instead comparing the probabilities of different output variables to one another, we could gain an understanding of which single output most accurately matches the input. For instance, consider a network that determines that a character has a 0.6 probability of being ‘C’ and a 0.7 probability of being ‘O’. With only one variable comparisons with numbers allowed, it would be difficult to set up a constraint that checks if ‘C’ is most likely to be the correct solution. For example, we would have to test multiple constraints with different thresholds, such as $C > 0.1$ and $O <= 0.1$, then $C > 0.2$ and $O <= 0.2$, and so on, where each of these is a different test. This is time-intensive. Implementing the ability to compare variables to each other in constraints would let the user check this property with one simple set of constraints; we would just need to run one verification with $C > O$. Because implementing this flexibility will reduce testing time, it will contribute to our goal of optimizing verification.

Expanding the type of constraints that the user can give to the neural network allows for more flexible verification. Currently, “or”’s are not allowed in constraints. Without “or”’s, we need to quantitatively verify a constraint “a or b” by first verifying “a”, then “b”, then “a and b”, and getting the respective model counts for each. To get the count for “a or b”, we have to add the model count for the verification of “a” to the model count for the verification of “b,” then subtract the model count for the verification of “a and b.”

The SMT solver we will be using, Z3, automatically converts constraints to the ideal form known as conjunctive normal form (CNF), in which Boolean clauses of disjunctions (logical “or”s) are joined by conjunctions (logical “and”s), before solving. However, we may be able to speed up execution by checking if user input is already in CNF or by converting the input to CNF ourselves as it is being read in.

3 Evaluation and Implementation

3.1 Evaluation

In order to evaluate the effectiveness of model counting, we plan to find a constraint where, after applying a model count, we discover that we can stop searching for solutions to the path early since we have found all possible inputs that satisfy the constraints for that path according to the model counter. We intend to compute how much time was taken with the early stop as well as without it, and compare the times to show how the model counting method is more efficient.

In order to evaluate the effectiveness of using brute force in certain situations, we will first find a rule that dictates when to use brute force versus when to use symbolic execution. Then, we will find a number of sets of constraints that satisfy this rule; sets of constraints that should be faster when evaluated using brute force. We will run the constraints using brute force, run them again with symbolic execution, and see how much time was saved.

The flexibility in user input methods does not lend itself to numerical evaluation quite as easily. We intend to find meaningful constraints that can only be expressed using the new input method to show that our method of expressing user input allows the user to check constraints that they otherwise would not be able to without multiple time-consuming full verification runs [5].

3.2 Timeline

Fall 2021

- Week 8-10
  - Read Z3 documentation and become familiar with it

Winter 2022

- Weeks 1-2
  - Implement flexibility to compare variables to each other in the input constraints, not just constants.
- Weeks 3-6
  - Add model counting before symbolic execution
- Weeks 7-10
  - Run tests to determine good rule for brute force solution.

Spring 2022

- Weeks 1-2
  - If brute force testing has concluded: include ORs to the conditions for increased flexibility, incorporate CNF checking, and add CNF conversion
- Weeks 3-4
  - Organize data, form conclusions, make figures
- Weeks 5-7
  - Paper draft planning
- Weeks 8-10
  - Final drafts

Note 1: Several weeks have been dedicated to brute force testing because despite it seeming really simple (just running the program with inputs), it actually gets really complicated and time-consuming for each test we will want to run.

Note 2: The current timeline is subject to change depending on how much time is spent on testing the brute force approach. In addition, the current Spring 2022, weeks 1-2, tasks would be ideal to get to, but are of less significance compared to variable comparison, model counting, and brute force solutions.

4 Changes Made After to Peer Reviews

- One concern was that our research problem was not well-motivated, and our peers did not understand the reason why optimizing verification is important. In the introduction, we motivated the problem better by explaining applications of neural networks, how neural networks work, and why they are important.

- Some sentences and term descriptions were unclear. We changed phrasing and sentence ordering in several places, and added more detail about terms like neural network weights and symbolic execution constraints.
- We updated the structure of our proposed solution so that our two sub-solutions were more clear (one focused on optimization, the other on flexibility).
- We added more to the timeline after thinking more about what we would really need to do during Winter and Spring quarters to stay on track.
- We added a proper title.

5 Changes Made After to Instructor Feedback

- We went back to all the literature search sources we found earlier in the quarter and looked at some new articles that our grad mentor Mara sent to us. We then referred to these sources in our paper and tried providing a stronger background on the issue.
- We explained some concepts more and added many simplified examples that might make understanding the topic easier.
- We restructured and rephrased some of the Problem Statement to show that finding the brute force threshold was also a potential solution (not just model counting). Earlier, it seemed as if model counting was our entire proposed solution.
- Before, it was unclear how flexibility related to optimization. Thus, we explained how increased flexibility contributes to more scalable verification, and altered the structure so it is clear that flexibility is also an important part of our proposed solution.
- We resized and centered the images; more specifically, we made Figure 2 smaller and Figure 3 bigger.

References

[1] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 255–272, Cham, 2015. Springer International Publishing.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018.

[3] Teodora Baluta, Zheng Leong Chua, Kuldeep S. Meel, and Prateek Saxena. Scalable quantitative verification for deep neural networks. *CoRR*, abs/2002.06864, 2020.

[4] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1249–1264, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Thomas A. Henzinger, Mathias Lechner, and Dorde Zikelic. Scalable verification of quantized neural networks (technical report). *ArXiv*, abs/2012.08185, 2020.

[6] 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*, pages 97–117. Springer, 2017.

[7] Kornel Papik, Bela Molnar, Rainer Schaefer, Zalan Dombovari, Zsolt Tulassay, and Janos Feher. Application of neural networks in medicine-a review. *Medical Science Monitor*, 4(3):MT538–MT546, 1998.

[8] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, and Taolue Chen. Bdd4bnn: A bdd-based quantitative analysis framework for binarized neural networks. *Computer Aided Verification Lecture Notes in Computer Science*, page 175–200, 2021.

[9] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, and Taolue Chen. BDD4BNN: A bdd-based quantitative analysis framework for binarized neural networks. *CoRR*, abs/2103.07224, 2021.