The Marabou Framework for Verification and Analysis of Deep Neural Networks

Guy Katz[1], Derek A. Huang[2], Duligur Ibeling[2], Kyle Julian[2], Christopher Lazarus[2], Rachel Lim[2], Parth Shah[2], Shantanu Thakoor[2], Haoze Wu[2], Aleksandar Zeljić[2], David L. Dill[2], Mykel Kochenderfer[2], and Clark Barrett[2]

[1] The Hebrew University of Jerusalem, Israel
guykatz@cs.huji.ac.il
[2] Stanford University, USA
{huangda, duligur, kjulian3, clazarus, parth95, thakoor, haozewu, zeljic, dill, mykel, clarkbarrett}@stanford.edu
rachelim@cs.stanford.edu

Abstract. Deep neural networks are revolutionizing the way complex systems are designed. Consequently, there is a pressing need for tools and techniques for network analysis and certification. To help in addressing that need, we present *Marabou*, a framework for verifying deep neural networks. Marabou is an SMT-based tool that can answer queries about a network's properties by transforming these queries into constraint satisfaction problems. It can accommodate networks with different activation functions and topologies, and it performs high-level reasoning on the network that can curtail the search space and improve performance. It also supports parallel execution to further enhance scalability. Marabou accepts multiple input formats, including protocol buffer files generated by the popular TensorFlow framework for neural networks. We describe the system architecture and main components, evaluate the technique and discuss ongoing work.

1 Introduction

Recent years have brought about a major change in the way complex systems are being developed. Instead of spending long hours hand-crafting complex software, many engineers now opt to use *deep neural networks* (*DNNs*) [6,19]. DNNs are machine learning models, created by training algorithms that generalize from a finite set of examples to previously unseen inputs. Their performance can often surpass that of manually created software as demonstrated in fields such as image classification [16], speech recognition [8], and game playing [21].

Despite their overall success, the opacity of DNNs is a cause for concern, and there is an urgent need for certification procedures that can provide rigorous guarantees about network behavior. The formal methods community has taken initial steps in this direction, by developing algorithms and tools for neural network verification [5, 9, 10, 12, 18, 20, 23, 24]. A DNN verification query consists of two parts: (i) a neural network, and (ii) a property to be checked; and its result is either a formal guarantee that the network satisfies the property, or a concrete input for which the property is violated (a counter-example). A verification query can encode the fact, e.g., that a network is robust to small adversarial perturbations in its input [22].

A neural network is comprised of *neurons*, organized in layers. The network is evaluated by assigning values to the neurons in the input layer, and then using these values to iteratively compute the assignments of neurons in each succeeding layer. Finally, the values of neurons in the last layer are computed, and this is the network's output. A neuron's assignment is determined by computing a weighted sum of the assignments of neurons from the preceding layer, and then applying to the result a non-linear activation function, such as the Rectified Linear Unit (ReLU) function, $\text{ReLU}(x) = \max(0, x)$. Thus, a network can be regarded as a set of *linear constraints* (the weighted sums), and a set of *non-linear constraints* (the activation functions). In addition to a neural network, a verification query includes a property to be checked, which is given in the form of linear or non-linear constraints on the network's inputs and outputs. The verification problem thus reduces to finding an assignment of neuron values that satisfies all the constraints simultaneously, or determining that no such assignment exists.

This paper presents a new tool for DNN verification and analysis, called *Marabou*. The Marabou project builds upon our previous work on the Reluplex project [2, 7, 12, 13, 15, 17], which focused on applying SMT-based techniques to the verification of DNNs. Marabou follows the Reluplex spirit in that it applies an SMT-based, *lazy search* technique: it iteratively searches for an assignment that satisfies all given constraints, but treats the non-linear constraints lazily in the hope that many of them will prove irrelevant to the property under consideration, and will not need to be addressed at all. In addition to search, Marabou performs deduction aimed at learning new facts about the non-linear constraints in order to simplify them.

The Marabou framework is a significant improvement over its predecessor, Reluplex. Specifically, it includes the following enhancements and modifications:

- Native support for fully connected and convolutional DNNs with arbitrary piecewise-linear activation functions. This extends the Reluplex algorithm, which was originally designed to support only ReLU activation functions.
- Built-in support for a *divide-and-conquer* solving mode, in which the solver is run with an initial (small) timeout. If the timeout is reached, the solver partitions its input query into simpler sub-queries, increases the timeout value, and repeats the process on each sub-query. This mode naturally lends itself to parallel execution by running sub-queries on separate nodes; however, it can yield significant speed-ups even when used with a single node.
- A complete simplex-based linear programming core that replaces the external solver (GLPK) that was previously used in Reluplex. The new simplex core was tailored for a smooth integration with the Marabou framework and eliminates much of the overhead in Reluplex due to the use of GLPK.
- Multiple interfaces for feeding queries into the solver. A query's neural network can be provided in a textual format or as a protocol buffer (*protobuf*)

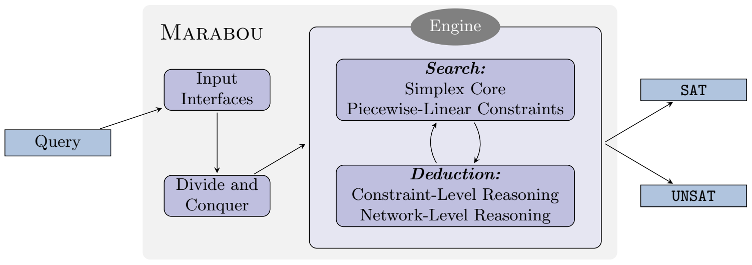

Fig. 1: The main components of Marabou.

file containing a TensorFlow model; and the property can be either compiled into the solver, provided in Python, or stored in a textual format. We expect these interfaces will simplify usage of the tool for many users.

– Support for network-level reasoning and deduction. The earlier Reluplex tool performed deductions at the level of single constraints, ignoring the input network's topology. In Marabou, we retain this functionality but also include support for reasoning based on the network topology, such as symbolic bound tightening [23]. This allows for efficient curtailment of the search space.

Marabou is available online [14] under the permissive modified BSD license.

2 Design of Marabou

Marabou regards each neuron in the network as a variable and searches for a variable assignment that simultaneously satisfies the query's linear constraints and non-linear constraints. At any given point, Marabou maintains the current variable assignment, lower and upper bounds for every variable, and the set of current constraints. In each iteration, it then changes the variable assignment in order to (1) correct a violated linear constraint, or (2) correct a violated non-linear constraint.

The Marabou verification procedure is sound and complete, i.e. the aforementioned loop eventually terminates. This can be shown via a straightforward extension of the soundness and completeness proof for Reluplex [12]. However, in order to guarantee termination, Marabou only supports activation functions that are piecewise-linear. The tool already has built-in support for the ReLU function and the Max function $\max(x_1, \ldots, x_n)$, and it is modular in the sense that additional piecewise-linear functions can be added easily.

Another important aspect of Marabou's verification strategy is deduction — specifically, the derivation of tighter lower and upper variable bounds. The motivation is that such bounds may transform piecewise-linear constraints into linear constraints, by restricting them to one of their linear segments. To achieve this, Marabou repeatedly examines linear and non-linear constraints, and also performs network-level reasoning, with the goal of discovering tighter variable bounds.

Next, we describe Marabou's main components (see also Fig. 1).

2.1 Simplex Core (*Tableau* and *BasisFactorization* classes)

The simplex core is the part of the system responsible for making the variable assignment satisfy the linear constraints. It does so by implementing a variant of the *simplex algorithm* [3]. In each iteration, it changes the assignment of some variable x, and consequently the assignment of any variable y that is connected to x by a linear equation. Selecting x and determining its new assignment is performed using standard algorithms — specifically, the *revised simplex method* in which the various linear constraints are kept in implicit matrix form, and the steepest-edge and Harris' ratio test strategies for variable selection.

Creating an efficient simplex solver is complicated. In Reluplex, we delegated the linear constraints to an external solver, GLPK. Our motivation for implementing a new custom solver in Marabou was twofold: first, we observed in Reluplex that the repeated translation of queries into GLPK and extraction of results from GLPK was a limiting factor on performance; and second, a black box simplex solver did not afford the flexibility we needed in the context of DNN verification. For example, in a standard simplex solver, variable assignments are typically pressed against their upper or lower bounds, whereas in the context of a DNN, other assignments might be needed to satisfy the non-linear constraints. Another example is the deduction capability, which is crucial for efficiently verifying a DNN and whose effectiveness might depend on the internal state of the simplex solver.

2.2 Piecewise-Linear Constraints (*PiecewiseLinearConstraint* class)

Throughout its execution, Marabou maintains a set of piecewise-linear constraints that represent the DNN's non-linear functions. In iterations devoted to satisfying these constraints, Marabou looks for any constraints that are not satisfied by the current assignment. If such a constraint is found, Marabou changes the assignment in a way that makes that constraint satisfied. Alternatively, in order to guarantee eventual termination, if Marabou detects that a certain constraint is repeatedly not satisfied, it may perform a *case-split* on that constraint: a process in which the piecewise-linear constraint φ is replaced by an equivalent disjunction of linear constraints $c_1 \vee \ldots \vee c_n$. Marabou considers these disjuncts one at a time and checks for satisfiability. If the problem is satisfiable when φ is replaced by some c_i, then the original problem is also satisfiable; otherwise, the original problem is unsatisfiable.

In our implementation, piecewise-linear constraints are represented by objects of classes that inherit from the *PiecewiseLinearConstraint* abstract class. Currently the two supported instances are ReLU and Max, but the design is modular in the sense that new constraint types can easily be added. *PiecewiseLinearConstraint* defines the interface methods that each supported piecewise-linear constraint needs to implement. Some of the key interface methods are:

– *satisfied()*: the constraint object needs to answer whether or not it is satisfied given the current assignment. For example, for a constraint $y = \text{ReLU}(x)$ and assignment $x = y = 3$, *satisfied()* would return *true*; whereas for assignment $x = -5, y = 3$, it would return *false*.

- *getPossibleFixes()*: if the constraint is not satisfied by the current assignment, this method returns possible changes to the assignment that would correct the violation. For example, for $x = -5, y = 3$, the ReLU constraint from before might propose two possible changes to the assignment, $x \leftarrow 3$ or $y \leftarrow 0$, as either would satisfy $y = \text{ReLU}(x)$.
- *getCaseSplits()*: this method asks the piecewise-linear constraint φ to return a list of linear constraints $c_1, \ldots, c_n$, such that φ is equivalent to $c_1 \vee \ldots \vee c_n$. For example, when invoked for a constraint $y = \max(x_1, x_2)$, *getCaseSplits()* would return the linear constraints $c_1 : (y = x_1 \wedge x_1 \geq x_2)$ and $c_2 : (y = x_2 \wedge x_2 \geq x_1)$. These constraints satisfy the requirement that the original constraint is equivalent to $c_1 \vee c_2$.
- *getEntailedTightenings()*: as part of Marabou's deduction of tighter variable bounds, piecewise-linear constraints are repeatedly informed of changes to the lower and upper bounds of variables they affect. Invoking *getEntailedTightenings()* queries the constraint for tighter variable bounds, based on current information. For example, suppose a constraint $y = \text{ReLU}(x)$ is informed of the upper bounds $x \leq 5$ and $y \leq 7$; in this case, *getEntailedTightenings()* would return the tighter bound $y \leq 5$.

2.3 Constraint- and Network-Level Reasoning (*RowBoundTightener*, *ConstraintBoundTightener* and *SymbolicBoundTightener* classes)

Effective deduction of tighter variable bounds is crucial for Marabou's performance. Deduction is performed at the constraint level, by repeatedly examining linear and piecewise-linear constraints to see if they imply tighter variable bounds; and also at the DNN-level, by leveraging the network's topology.

Constraint-level bound tightening is performed by querying the piecewise-linear constraints for tighter bounds using the *getEntailedTightenings()* method. Similarly, linear equations can also be used to deduce tighter bounds. For example, the equation $x = y + z$ and lower bounds $x \geq 0$, $y \geq 1$ and $z \geq 1$ together imply the tighter bound $x \geq 2$. As part of the simplex-based search, Marabou repeatedly encounters many linear equations and uses them for bound tightening.

Several recent papers have proposed verification schemes that rely on DNN-level reasoning [5, 23]. Marabou supports this kind of reasoning as well, by storing the initial network topology and performing deduction steps that use this information as part of its iterative search. DNN-level reasoning is seamlessly integrated into the search procedure by (1) initializing the DNN-level reasoners with the most up-to-date information discovered during the search, such as variable bounds and the state of piecewise-linear constraints; and (2) feeding any new information that is discovered back into the search procedure. Presently Marabou implements a symbolic bound tightening procedure [23]: based on network topology, upper and lower bounds for each hidden neuron are expressed as a linear combination of the input neurons. Then, if the bounds on the input neurons are sufficiently tight (e.g., as a result of past deductions), these expressions for upper and lower bounds may imply that some of the hidden neurons' piecewise-linear activation functions are now restricted to one of their linear segments. Implementing additional DNN-level reasoning operations is work in progress.

2.4 The Engine (*Engine* and *SmtCore* classes)

The main class of Marabou, in which the main loop resides, is called the *Engine*. The engine stores and coordinates the various solution components, including the simplex core and the piecewise-linear constraints. The main loop consists, roughly, of the following steps (the first rule that applies is used):

1. If a piecewise-linear constraint had to be fixed more than a certain number of times, perform a case split on that constraint.
2. If the problem has become unsatisfiable, e.g. because for some variable a lower bound has been deduced that is greater than its upper bound, undo a previous case split (or return `UNSAT` if no such case split exists).
3. If there is a violated linear constraint, perform a simplex step.
4. If there is a violated piecewise-linear constraint, attempt to fix it.
5. Return `SAT` (all constraints are satisfied).

The engine also triggers deduction steps, both at the neuron level and at the network level, according to various heuristics.

2.5 The Divide-and-Conquer Mode and Concurrency (*DnC.py*)

Marabou supports a *divide-and-conquer* (*D&C*) solving mode, in which the input region specified in the original query is partitioned into sub-regions. The desired property is checked on these sub-regions independently. The D&C mode naturally lends itself to parallel execution, by having each sub-query checked on a separate node. Moreover, the D&C mode can improve Marabou's overall performance even when running sequentially: the total time of solving the sub-queries is often less than the time of solving the original query, as the smaller input regions allow for more effective deduction steps.

Given a query ϕ, the solver maintains a queue Q of $\langle$query, timeout$\rangle$ pairs. Q is initialized with one element $\langle \phi, T\rangle$, where T, the initial timeout, is a configurable parameter. To solve ϕ, the solver loops through the following steps:

1. Pop a pair $\langle \phi', t'\rangle$ from Q and attempt to solve ϕ' with a timeout of t'.
2. If the problem is `UNSAT` and Q is empty, return `UNSAT`.
3. If the problem is `UNSAT` and Q is not empty, return to step 1.
4. If the problem is `SAT`, return `SAT`.
5. If a timeout occurred, split ϕ' into k sub-queries $\phi'_1, \ldots, \phi'_k$ by partitioning its input region. For each sub-query ϕ'_i, push $\langle \phi'_i, m \cdot t'\rangle$ into Q.

The timeout factor m and the splitting factor k are configurable parameters. Splitting the query's input region is performed heuristically.

2.6 Input Interfaces (*AcasParser* class, *maraboupy* folder)

Marabou supports verification queries provided through the following interfaces:

- Native Marabou format: a user prepares a query using the Marabou C++ interface, compiles the query into the tool, and runs it. This format is useful for integrating Marabou into a larger framework.
- Marabou executable: a user runs a Marabou executable, and passes to it command-line parameters indicating the network and property files to be checked. Currently, network files are encoded using the *NNet* format [11], and the properties are given in a simple textual format.
- Python/TensorFlow interface: the query is passed to Marabou through Python constructs. The python interface can also handle DNNs stored as TensorFlow protobuf files.

3 Evaluation

For our evaluation we used the ACAS Xu [12], CollisionDetection [4] and TwinStream [1] families of benchmarks. Tool-wise, we considered the Reluplex tool which is the most closely related to Marabou, and also ReluVal [23] and Planet [4]. The version of Marabou used for the evaluation is available online [14].

The top left plot in Fig. 3 compares the execution times of Marabou and Reluplex on 180 ACAS Xu benchmarks with a 1 hour timeout. We used Marabou in D&C mode with 4 cores and with $T = 5$, $k = 4$, and $m = 1.5$. The remaining three plots depict an execution time comparison between Marabou D&C (configuration as above), ReluVal and Planet, using 4 cores and a 1 hour timeout. Marabou and Reluval are evaluated over 180 ACAS Xu benchmarks (top right plot), and Marabou and Planet are evaluated on those 180 benchmarks (bottom left plot) and also on 500 CollisionDetection and 81 TwinStream benchmarks (bottom right plot). Due to technical difficulties, ReluVal was not run on the CollisionDetection and TwinStream benchmarks. The results show that in a 4 cores setting Marabou generally outperforms Planet, but generally does not outperform ReluVal (though it does better on some benchmarks). These results highlight the need for additional DNN-level reasoning in Marabou, which is a key ingredient in ReluVal's verification procedure.

Fig. 2 shows the average runtime of Marabou and ReluVal on the ACAS Xu properties, as a function of the number of available cores. We see that as the number of cores increases, Marabou (solid) is able to close the gap, and sometimes outperform, ReluVal (dotted). With 64 cores, Marabou outperforms ReluVal on average, and both solvers were able to solve all ACAS Xu benchmarks within 2 hours (except for a few segfaults by ReluVal).

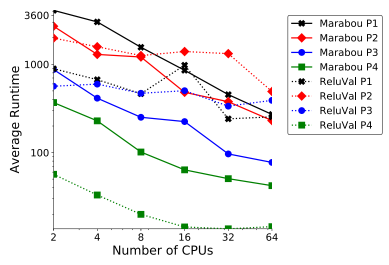

Fig. 2: A scalability comparison of Marabou and ReluVal on ACAS Xu

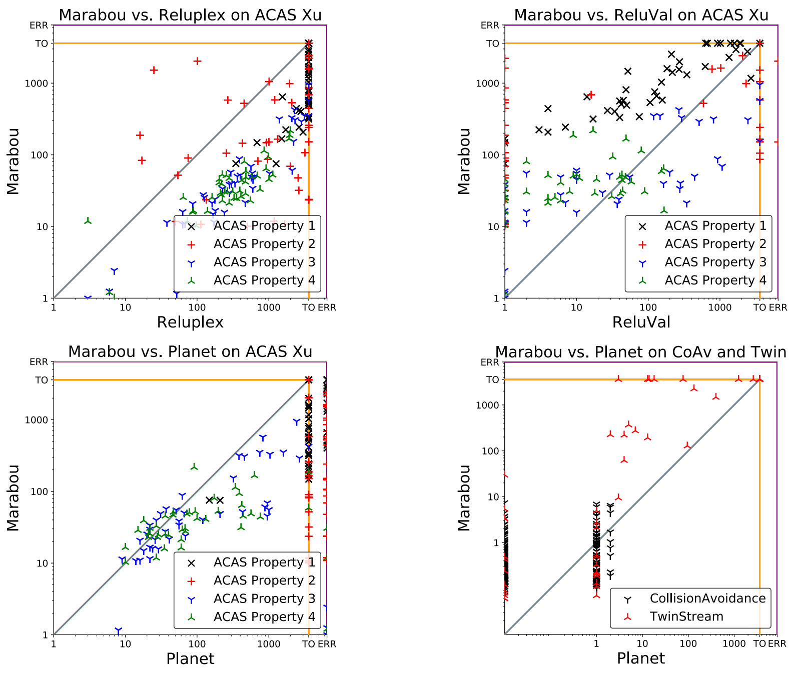

Fig. 3: A comparison of Marabou with Reluplex, ReluVal and Planet.

4 Conclusion

DNN analysis is an emerging field, and Marabou is a step towards a more mature, stable verification platform. Moving forward, we plan to improve Marabou in several dimensions. Part of our motivation in implementing a custom simplex solver was to obtain the needed flexibility for fusing together the solving process for linear and non-linear constraints. Currently, this flexibility has not been leveraged much, as these pieces are solved relatively separately. We expect that by tackling both kinds of constraints simultaneously, we will be able to improve performance significantly. Other enhancements we wish to add include: additional network-level reasoning techniques based on abstract interpretation; better heuristics for both the linear and non-linear constraint solving engines; and additional engineering improvements, specifically within the simplex engine.

Acknowledgements We thank Elazar Cohen, Justin Gottschlich, and Lindsey Kuper for their contributions to this project. The project was partially supported by grants from the Binational Science Foundation (2017662), the Defense Advanced Research Projects Agency (FA8750-18-C-0099), the Federal Aviation Administration, Ford Motor Company, Intel Corporation, the Israel Science Foundation (683/18), the National Science Foundation (1814369, DGE-1656518), Siemens Corporation, and the Stanford CURIS program.

References

1. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and M. Kumar. Piecewise Linear Neural Network Verification: A Comparative Study, 2017. Technical Report. `https://arxiv.org/abs/1711.00455v1`.
2. N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. `https://arxiv.org/abs/1709.10207`.
3. V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.
4. R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
5. T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
6. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
7. D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
8. G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
9. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
10. J. Hull, D. Ward, and R. Zakrzewski. Verification and Validation of Neural Networks for Safety-Critical Applications. In *Proc. 21st American Control Conf. (ACC)*, 2002.
11. K. Julian. NNet Format, 2018. `https://github.com/sisl/NNet`.
12. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
13. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.
14. G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. Marabou, 2019. `https://github.com/guykatzz/Marabou/tree/cav_artifact`.
15. Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, 2019.
16. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
17. L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. `https://arxiv.org/abs/1801.05950`.

18. L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.
19. M. Riesenhuber and P. Tomaso. Hierarchical Models of Object Recognition in Cortex. *Nature Neuroscience*, 2(11):1019–1025, 1999.
20. W. Ruan, X. Huang, and M. Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proc. 27th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2018.
21. D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
22. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. `http://arxiv.org/abs/1312.6199`.
23. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals, 2018. Technical Report. `http://arxiv.org/abs/1804.10829`.
24. W. Xiang, H. Tran, and T. Johnson. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2018.