Continuous Verification of Machine Learning: a Declarative Programming Approach

Ekaterina Komendantskaya
Heriot-Watt University
Edinburgh, Scotland
ek19@hw.ac.uk

Wen Kokke
Heriot-Watt University
Edinburgh, Scotland
wen.kokke@ed.ac.uk

Daniel Kienitz
Heriot-Watt University
Edinburgh, Scotland
dk50@hw.ac.uk

Abstract

In this invited talk, we discuss state of the art in neural network verification. We propose the term *continuous verification* to characterise the family of methods that explore continuous nature of machine learning algorithms. We argue that methods of continuous verification must rely on robust programming language infrastructure (refinement types, automated proving, type-driven program synthesis), which provides a major opportunity for the declarative programming language community.

Keywords: Neural Networks, Verification, AI.

CCS Concepts

• **Software and its engineering** → **Software reliability**; **Software safety**; **Functional languages**; **Constraint and logic languages**; **Data types and structures**; **Constraints**; • **Computing methodologies** → **Artificial intelligence**; **Neural networks**.

ACM Reference Format:
Ekaterina Komendantskaya, Wen Kokke, and Daniel Kienitz. 2020. Continuous Verification of Machine Learning: a Declarative Programming Approach. In *22nd International Symposium on Principles and Practice of Declarative Programming (PPDP '20), September 8–10, 2020, Bologna, Italy.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3414080.3414094

1 Motivation: Verification for AI

Programmers make, on average, 15–50 errors per 1,000 lines of code [19]. Yet, the quality of software we use determines reliability, safety and security of computer systems and applications we rely on. This, or smilar, observations have been driving verification research for the past two decades, ultimately leading to a range of *lightweight verification solutions* [8], proposed by Industry and Academia [4, 12].

Recent success and wide deployment of AI-driven applications have brought a novel challenge for verification communities. Machine learning algorithms have always been valued for their ability to generate classifiers (or functions) that recognise patterns in noisy or incomplete data. *Neural networks* is an umbrella term for a range of such classifiers. Yet only recently [22], it was discovered that neural networks are prone to *adversarial attacks*—specially crafted inputs that lead to undesired outputs.

It is very tempting to make an analogy between a buggy piece of software and an adversarially vulnerable neural network, and launch the old trusted verification artillery on this new enemy. There is a mounting evidence, however, that this approach will not work so easily. Firstly, neural networks as verification objects are very different from the usual programmed software. The latter may be buggy, but have some semantic meaning and composable components, enabling us to verify some parts of the code *that matter* and make them bug-free. But there is no neural network in existence that is robust against adversarial attacks, and we cannot meaningfully de-compose neural networks, in order to systematically identify the parts that matter, as we did with conventional software verification projects. We can try to constrain their inputs, and give some output guarantees based on those constraints. Most existing neural net verification approaches take this path [11, 14, 20].

For example, a usual notion of neural network robustness for some class $\hat{y}$ would be to define an ϵ-ball $\mathbb{B}(\hat{x})$ around a sample input $\hat{x}$ of that class: $\mathbb{B}(\hat{x}, \epsilon) = \{x \in \mathbb{R}^n : ||\hat{x} - x|| \leq \epsilon\}$, and then prove that any object in $\mathbb{B}(\hat{x}, \epsilon)$ will be classified as $\hat{y}$.

The second problem is the scale of the verification task. For example, the famous "textbook" dataset MNIST [18] contains 28×28 images of the handwritten digits "0" to "9". A neural network able to classify these images will receive an input of 748 pixels, and will have an output that gives a probability distribution over the 10 classes or simply a predicted class. This leaves us to determine the number of hidden layers of the neural network, their sizes, and their activation functions. For instance, we could opt for a 128-node hidden layer using a simplest (and linear) activation function *ReLU*. Let us pass this neural network as a function to an SMT solver, and verify whether certain constraints on its input can guarantee certain constraints on its outputs. Unfortunately, this model has $784 \times 128 + 128 + 128 \times 10 + 10 = 101770$ constant parameters and 784 input parameters. Worse, it has 3 fully-connected layers, meaning that each input parameter occurs at least $128 \times 10 = 1280$ times in the SMT query, and constant parameters occur several times in accordance to the layer they are in. This is a *huge* query from an SMT solving perspective, and it would overwhelm any SMT solver. However, this is not a large network from a machine learning perspective. Autonomous cars, for example, need to process images of much higher complexity and resolution.

Finally, there is a problem of verification with non-linear arithmetic. Mathematically, a neural network is a *non-linear function* that separates data points in an n dimensional real space into m classes. State-of-the-art neural networks rely heavily on non-linear activation functions, such as *sigmoid*, *tanh*, *softmax*, to establish

PPDP '20, September 8–10, 2020, Bologna, Italy

ACM ISBN 978-1-4503-8821-4/20/09…$15.00
https://doi.org/10.1145/3414080.3414094

more sophisticated and more precise decision boundaries between classes. Unfortunately, SMT solvers do not generally support non-linear arithmetic, and where they do, the solvers are slower and less reliable. For example, the Z3 solver [5] uses Dual Simplex [6] to solve linear real arithmetic. It also supports a fragment of non-linear real arithmetic—specifically, multiplications—and solves this using a conflict resolution procedure based on cylindrical algebraic decomposition [13]. However, the addition of multiplication is not enough to cover the non-linear activation functions used in deep learning, which often use exponents, logarithms, and trigonometric functions. The only solver we are aware of that supports these functions out of the box is MetiTarski [1]. However, the MetiTarski documentation reads "Beyond 4 or 5 continuous variables, there is very little hope for MetiTarski in finding a proof."

In short, neural networks will not yeild the good trusted verification artillery. What are the options?

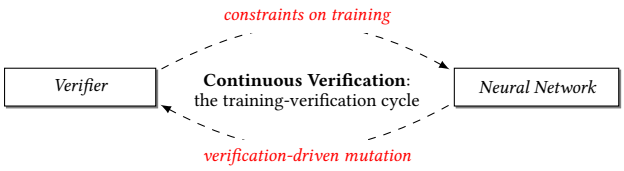
2 Continuous Verification

Most of the challenges we encounter in neural network verification are due to the conflict between continuous methods (that enable data classification in multi-dimensional real space) and the discrete methods (used by solvers and provers). But perhaps disadvantages can be turned into capabilities. Conventionally, we assume that the object we verify is uniquely defined, often hand-written, and therefore needs to be verified as-is. Neural networks are different—often there is a continuum of models that can serve as suitable classifiers, and we usually do not have much preference for any of them, as long as they give reasonable prediction accuracy. Given the task of verifying a neural network, we are no longer required to think of the object as immutable, i.e. we are allowed to verify and deploy a different neural network instead.

This opens up new possibilities for verification and justifies several methods of neural network transformation, e.g.:

- neural network size reduction (e.g. by pruning or weight quantisation), as already explored in [9];
- piece-wise linearisation of activation functions either during or after training, see [17];
- use of constraint-aware loss functions during training [2, 7], or interleave verification with adversarial training [3], which improves safety of neural networks, and hence ease verification tasks.

Thus verification becomes part of the object construction. Moreover, we assume that the training-verification cycle may repeat potentially indefinitely, especially if neural networks are retrained using new data:



We call such approach to verification *continuous verification*. However, to be truly successful, this methodology needs proper programming language support. Ideally, the programmer should only need to specify basic neural network parameters and the desired verification constraints, leaving the work of fine-tuning of the training-verification cycle to the integrated tools.

3 Types

It may be fruitful to cast a type-theoretic view on these problems. A conventional verification project aims to establish a proof $\Gamma \vdash f : A$, where f is a function or code we verify, A is a verification property, and Γ is the given theory. In [17] we show that the existing neural network verification projects in fact amount to working with a special sort of types, that are *refined* with SMT-constraints. Simplest examples of refinement types are positive reals $(x : \mathbb{R}\{x > 0\})$, or booleans which are true $(b : \mathbf{Bool}\{b = true\})$. It is easy to see that the idea of verifying neural networks for all objects "within the ϵ-ball", fits exactly with the syntax of refinement types! I.e., we verify a neural network $f_N : \mathbb{R}^n \to \mathbb{R}$, by imposing a type $f_N : (x : \mathbb{R}^n\{||\hat{x} - x||_2 \leq \epsilon\}) \to (y : \mathbb{R}\{y = \hat{y}\})$.

F* [21] and Liquid Haskell [23] are functional languages with refinement types. Unlike, e.g. Python, they are referentially transparent, which means the semantics of pure programs in these languages can be directly encoded in the SMT logic. This tight integration allows us to specify neural network models and their properties in the same language [16], while leveraging the powerful automated verification offered by SMT solvers!

But there is more to it. Assuming that continuous verification is a solution to the problems that have been haunting the neural network verification, what is the place of types in this picture? Let us assume once again that we are interested in proving $f_N : (x : \mathbb{R}^n\{||\hat{x} - x||_2 \leq \epsilon\}) \to (y : \mathbb{R}\{y = \hat{y}\})$, except for, the automated proof (i.e. type checking) is out of reach. Instead we are allowed to generate another function, f'_N of this type, and prove it correct. This kind of problem is well-known in the declarative programming community under the name of *type-driven program synthesis*. Just recently, we have seen successes of type-driven synthesis for refinement types [10, 15]. From the point of view of neural network verification, the missing part is to enrich the existing typeful languages with program synthesis algorithms that resemble the training-verification cycle depicted in Section 2.

Our early experiments [17] show that types give an easy and intuitive language to specify sets of admissible or desired neural networks, while keeping the training, size reduction, and linearisation completely flexible and modular. In fact, all machine learning tasks can be fully delegated to an external language, suchas Python [16].

However, a lot more could and should be done in order to establish proper declarative programming methodology for continuous verification of neural networks.

4 Acknowledgements

We acknowledge support of the UK Nationaly Cyber Security Center grant *SecCon-NN: Neural Networks with Security Contracts - towards lightweight, modular security for neural networks* and the UK Research Institute in Verified Trustworthy Software Systems (VETSS)-funded research project *CONVENER: Continuous Verification of Neural Networks.*

References

[1] Behzad Akbarpour and Lawrence Charles Paulson. 2009. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning* 44, 3 (Aug. 2009), 175–205. https://doi.org/10.1007/s10817-009-9149-2

[2] Edward Ayers, Francisco Eiras, Majd Hawasly, and Iain Whiteside. 2020. PaRoT: A Practical Framework for Robust Deep Neural Network Training. In *NASA Formal Methods*. http://arxiv.org/abs/2001.02152

[3] Mislav Balunovic and Martin T. Vechev. 2020. Adversarial Training and Provable Defenses: Bridging the Gap. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification, In *NASA Formal Methods*.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08 (LNCS)*, Vol. 4963. 337–340.

[6] Bruno Dutertre and Leonardo de Moura. 2006. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*. Springer Berlin Heidelberg, 81–94. https://doi.org/10.1007/11817963_11

[7] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin T. Vechev. 2019. DL2: Training and Querying Neural Networks with Logic. In *Proc. of the 36th Int. Conf. Machine Learning, ICML 2019*, Vol. 97. PMLR, 1931–1941.

[8] Kathleen Fisher, John Launchbury, and Raymond Richards. 2017. Using Formal Methods to Eliminate Exploitable Bugs. *Phil. Trans. R. Soc.* (2017).

[9] Ben Goldberger, Guy Katz, Yossi Adi, and Joseph Keshet. 2020. Minimal Modifications of Deep Neural Networks using Verification. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020 (EPiC Series in Computing)*, Elvira Albert and Laura Kovács (Eds.), Vol. 73. EasyChair, 260–278.

[10] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL (2020), 12:1–12:28.

[11] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV 2017*, Vol. LNCS 10426. Springer, 3–29. https://doi.org/10.1007/978-3-319-63387-9

[12] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. A Reflection on Types — A List of Successes That Can Change the World. (2016).

[13] Dejan Jovanović and Leonardo de Moura. 2013. Solving non-linear arithmetic. *ACM Communications in Computer Algebra* 46, 3/4 (Jan. 2013), 104. https://doi.org/10.1145/2429135.2429155

[14] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *CAV 2019, Part I (LNCS)*, Vol. 11561. Springer, 443–452.

[15] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann, and Nadia Polikarpova. 2020. Liquid resource types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 106:1–106:29.

[16] Wen Kokke, Ekaterina Komendantskaya, and Daniel Kienitz. 2020. StarChild, a library for leveraging the refinement types and SMT solving of F* to verify properties of neural networks. https://github.com/wenkokke/starchild

[17] Wen Kokke, Ekaterina Komendantskaya, Daniel Kienitz, Robert Atkey, , and David Aspinall. 2020. Neural Networks, Secure by Construction: An Exploration of Refinement Types. Draft.

[18] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[19] Steve McConnell. 2015. *Code Complete. A Practical Handbook of Software Construction, Second Edition.*

[20] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *PACMPL* 3, POPL (2019), 41:1–41:30. https://doi.org/10.1145/3290354

[21] Nikhil Swamy, Markulf Kohlweiss, Jean-Karim Zinzindohoue, Santiago Zanella-Béguelin, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, and Pierre-Yves Strub. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*. ACM Press. https://doi.org/10.1145/2837614.2837655

[22] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. *CoRR* abs/1312.6199 (2014). https://arxiv.org/abs/1312.6199

[23] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. University of California, San Diego, USA.