2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

Robustness of Neural Networks: A Probabilistic and Practical Approach

Ravi Mangal Aditya V. Nori Alessandro Orso

Georgia Institute of Technology
Atlanta, GA, USA 30332-0765
{rmangal3, orso}@gatech.edu

Microsoft Research
Cambridge, CB1 2FB, UK
adityan@microsoft.com

***Abstract*—Neural networks are becoming increasingly prevalent in software, and it is therefore important to be able to verify their behavior. Because verifying the correctness of neural networks is extremely challenging, it is common to focus on the verification of other properties of these systems. One important property, in particular, is robustness. Most existing definitions of robustness, however, focus on the worst-case scenario where the inputs are adversarial. Such notions of robustness are too strong, and unlikely to be satisfied by—and verifiable for—practical neural networks. Observing that real-world inputs to neural networks are drawn from non-adversarial probability distributions, we propose a novel notion of robustness: probabilistic robustness, which requires the neural network to be robust with at least $(1 - \epsilon)$ probability with respect to the input distribution. This probabilistic approach is practical and provides a principled way of estimating the robustness of a neural network. We also present an algorithm, based on abstract interpretation and importance sampling, for checking whether a neural network is probabilistically robust. Our algorithm uses abstract interpretation to approximate the behavior of a neural network and compute an overapproximation of the input regions that violate robustness. It then uses importance sampling to counter the effect of such overapproximation and compute an accurate estimate of the probability that the neural network violates the robustness property.**

***Index Terms*—neural networks; probabilistic; robustness**

I. Introduction

Neural networks are increasingly becoming an important computational component of modern software. With their widespread adoption, it has become essential that we ensure (or at least gain confidence in) the correctness of neural networks, as we do with traditional programs. However, providing formal specifications of correctness is an even harder task for neural networks than for traditional programs, as neural networks are explicitly designed for the purpose of learning patterns in training data that are not easily apparent to humans.

Despite the difficulty of specifying the concept of correctness for neural networks, there are some important properties that such networks should satisfy. In particular, in recent years, researchers have observed certain undesirable neural network behaviors, including susceptibility to input perturbations [1], unfairness of neural network outcomes [2], [3], and leakage of private information (confidentiality and integrity issues) [4], [5]. In this work, we focus on the property of robustness of neural networks to input perturbations.

An important concept, in the context we target, is *input perturbation*: a subtle perturbation of an input such that the behavior of the neural network is correct on the unperturbed input but incorrect on the perturbed one. Existing literature has focused on the worst-case scenario where the perturbations are adversarial, without regard to whether such adversarial inputs are likely to be generated by real-world processes. Accordingly, a variety of adversarial perturbations/attacks, and defenses against such attacks, have been proposed (see [6] for a survey). Further, a variety of formal definitions of adversarial robustness have been presented. Broadly speaking, these adversarial formulations can be classified into two main groups: (i) *local robustness* and (ii) *global robustness*.

Intuitively, *local robustness* [7]–[9] is defined for a given input x and states that the neural network should produce the same result (e.g., label) for x and for all inputs x' within a ball of radius δ centered at x. (Notice that this definition relies on a suitable distance metric defined over the input space.) The requirement to be robust for all inputs that are δ-close to x (i.e., within distance δ from x) might make sense under certain threat models. In practice, however, for neural networks operating in non-malicious settings, this can be too strong a requirement; all δ-close inputs may not be equally likely, and violating robustness for a highly unlikely x' may be considered practically acceptable. At the same time, since local robustness is only defined for specific inputs and provides no guarantees for the inputs that have not been considered, it can also be too weak and inherently limited. *Global robustness* [9] addresses this issue by further demanding that the local robustness property be satisfied by all the inputs in the input space. In addition to being computationally intractable to check, global robustness is again too strong to be of practical use.

To address the practical and conceptual limitations of these existing definitions of adversarial robustness for neural networks, we propose a new robustness property, probabilistic robustness, that is targeted towards non-adversarial settings and is globally defined. Our formulation is motivated by two observations: (i) inputs to a neural network are generated according to an (either known or unknown)[1] underlying real-world probability distribution over the input space; and (ii) in

[1]Although proving probabilistic robustness requires the underlying input distribution to be known, in the absence of such information we can rely on a standard known distribution.

978-1-7281-1758-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-NIER.2019.00032

non-adversarial settings, we are only interested in robustness of a neural network for pairs of δ-close inputs that are likely to be generated in the real-world. Consequently, instead of proving robustness for either arbitrary or all δ-close inputs, we propose to prove robustness for pairs of δ-close inputs such that their cumulative probability is at least $(1-\epsilon)$. Such a proof guarantees that, for a random pair of δ-close inputs drawn from the input distribution, the probability that the neural networks violates robustness is ϵ at the most. We believe that, compared to local and global robustness, probabilistic robustness represents a more practical and efficiently checkable property. Moreover, unlike local robustness, this property is globally defined. Finally, the parameter ϵ is a tunable knob that can be used to control the trade-off between computational efficiency and strength of the property.

The description we just provided gives an intuitive idea of probabilistic robustness. More formally, *probabilistic robustness* can be expressed by the following formula:

$$\Pr_{x,x' \sim D}(\|f(x') - f(x)\| \leq k * \|x' - x\| \mid \|x' - x\| \leq \delta) \geq 1 - \epsilon$$

Here, f stands for the mathematical function represented by the neural network, and $f(x)$ represents the output generated by the neural network on input x. $\|\cdot\|$ represents the norm or distance metric used over the input and output space (assuming that the same metric is used for both). This definition states that for a randomly sampled pair of inputs, conditioned on the inputs being δ-close, function f satisfies the *Lipschitz property*; that is, the distance between the outputs is bounded by a k-multiple of the distance between the inputs, with a high probability $(1-\epsilon)$. In other words, and more intuitively, probabilistic robustness requires that pairs of inputs that are (1) drawn from the real-world and (2) close to each other, result in outputs that are similarly close with a high probability. Note that this definition does not apply when the output is discrete or categorical, and we assume that the output is a real vector. This assumption, however, does not affect the applicability of our technique; as we further explain in Section III, even neural networks that produce categorical outputs can be treated as producing a distribution over the class labels as an output (using a soft-max as a final layer, for instance).

To check whether a given neural network satisfies probabilistic robustness, a naive verification algorithm would run the network against every δ-close pair of inputs and check and record whether the Lipschitz property is satisfied. It would then compute the total probability of all the recorded pairs of inputs and check whether this probability is greater than $(1-\epsilon)$. Obviously, this algorithm would be impractical, as the input space of neural networks can be arbitrarily large.

To make our approach feasible, we present an algorithm that combines abstract interpretation [10] (from programming languages theory) and importance sampling [11] (from statistical theory) and makes the verification of probabilistic robustness computationally tractable. Abstract interpretation can take as input (the precise description of) a program with possibly infinite behaviors and generate a finite, sound, precise, and computable approximation of the program behaviors; we use abstract interpretation to approximate the behavior of a neural network without running it on all possible inputs. Importance sampling is a sampling technique that helps improve the precision of statistical estimates, while reducing the number of samples necessary to compute the estimate; we use importance sampling to estimate the probability of all the input pairs that satisfy the property.

$$\begin{array}{rcl} f(\bar{x}) & := & W \cdot \bar{x} + \bar{b} \\ & \mid & \textbf{case } E_1 : f_1(\bar{x}), ..., \textbf{case } E_k : f_k(\bar{x})) \\ & \mid & f(f'(x)) \\ E & := & E \wedge E \mid x_i \geq x_j \mid x_i \geq 0 \mid x_i < 0 \end{array}$$

Fig. 1. Definition of CAT functions.

The contributions of this work are twofold. First, we propose probabilistic robustness, a new non-adversarial robustness property of neural networks. Second, we present a practical algorithm for checking whether a network satisfies this property.

II. Background

Neural networks are functions that map real-valued vector inputs to real-valued vector outputs. In this paper, we use the conditional affine transformations (CAT) representation of neural networks [7], [12], shown in Figure 1. CAT functions consist of functions $f : \mathbb{R}_m \mapsto \mathbb{R}_n$, where $m, n \in \mathbb{N}$, and are recursively defined. Any affine transformation $f(\bar{x}) := W \cdot \bar{x} + \bar{b}$, for matrix W and vector $\bar{b}$, is a CAT function. Conditional expressions with multiple cases, and composition of CAT functions, are CAT functions. There is a straightforward translation of neural networks with ReLU activation functions [13] and standard layer types (e.g., fully connected layer, convolutional layer, and max pooling layer) into CAT functions (see Gehr et al.'s work [12] for details).

Abstract interpretation [10] is a framework for understanding and proving properties about programs with potentially infinite behaviors. Abstract interpretation techniques can soundly approximate these behaviors in a finite, computable way. Although the details of abstract interpretation are beyond the scope of this paper, we provide an example to intuitively explain the approach. Consider a function $f : \mathbb{R}_m \mapsto \mathbb{R}_n$, where $m, n \in \mathbb{N}$, and a set $C \subseteq \mathbb{R}_n$. Suppose that we want to find the largest set $X \subseteq \mathbb{R}_m$ such that $\forall x \in X. f(x) \in C$. If f is invertible, one way to compute X is by computing $F^{-1}(C)$, where $F^{-1} : P(\mathbb{R}_n) \mapsto P(\mathbb{R}_m)$, and $F^{-1}(Y) = \{x | f(x) \in Y\}$. F^{-1} is just a lifting of f^{-1} to be over a set of outputs Y, rather than a single output y, and is called the *concrete backward transformer*. If F^{-1} has an efficient representation, $F^{-1}(C)$ can be computed efficiently, but F^{-1} itself can be very inefficient (even non-terminating). Using abstract interpretation, we can design an *abstract backward transformer* $\hat{F}^{-1}$ such that computing $\hat{F}^{-1}(\hat{C}) = \hat{X}$ is guaranteed to be efficient, and $C \subseteq \hat{C}$ and $X \subseteq \hat{X}$ (i.e., $\hat{C}$ and $\hat{X}$) are sound overapproximations of C and X, respectively.

Importance sampling [11] is a sampling technique for estimating unlikely properties of distributions. In particular, if the region in which the property holds has a low probability, vanilla Monte Carlo sampling is very unlikely to produce

points from within that region; one is forced to either generate a large number of samples, or accept a very imprecise (large variance) estimate of the property under consideration. Importance sampling can help in such a situation. Instead of sampling from the original distribution, we (1) sample from a distribution that attaches a high probability to the region of interest, (2) estimate the property for this new distribution, and (3) weight this estimate so as to generate the estimate for the original distribution. In many cases, importance sampling can help generate precise estimates with much fewer samples compared to vanilla Monte Carlo sampling.

III. Probabilistic Robustness

As we stated earlier, existing formulations of neural network robustness are focused on the worst-case (i.e., the adversarial setting). Practically, these formulations are not only too strong, but also computationally expensive to verify. Our formulation of probabilistic robustness aims to find a practical notion of robustness that is suitable for non-adversarial settings and is computationally efficient to verify.

To contrast our formulation with the existing ones, we first provide a formal definition of local and global robustness. A neural network satisfies local robustness at input x_0 if the following formula holds true:

$$\forall x. \|x_0 - x\| \leq \delta \implies f(x_0) = f(x)$$

In the formula, f is the mathematical function represented by the neural network, and $\|\cdot\|$ is a distance metric defined on the input space. Intuitively, the formula states that for all inputs in the ball of radius δ centered at x_0, the network produces the same output. Note that input x_0 must be explicitly provided. Because there is no principled guidance on which inputs to select, such inputs are typically selected in an ad-hoc fashion.

Global robustness basically consists of enforcing local robustness for every input in the input space and can be expressed as follows:

$$\forall x, x'. \|x - x'\| \leq \delta \implies f(x) = f(x')$$

Because this formula is universally quantified over both x and x', this property tends to be too strong to be of practical use—most real-world neural networks are likely to violate it.

In contrast, a neural network satisfies probabilistic robustness if the following formula holds true,

$$\Pr_{x,x' \sim D}(\|f(x') - f(x)\| \leq k * \|x' - x\| \;\big|\; \|x' - x\| \leq \delta) \geq 1 - \epsilon$$

In the formula, D indicates the input distribution. Probabilistic robustness differs from local and global robustness in two major ways. *First*, instead of requiring that the neural network produces equal output on multiple different inputs, this property bounds the distance between every pair of outputs in terms of the distance between the corresponding pair of inputs. (A function satisfying this property over its entire domain is referred to as a Lipschitz continuous function). *Second*, the property is not established for arbitrary or all δ-close inputs. Instead, to prove the property, one needs to establish it for pairs of δ-close inputs with a total probability of at least $(1 - \epsilon)$ with respect to the distribution D. In case the exact underlying distribution is unknown, which is likely to be the common case, one can prove this property for some standard distribution and still infer useful information about the neural network. Note that, because the notion of Lipschitz continuity does not apply to functions with a discrete or categorical output, we require that the output of the neural network be continuous. However, this does not practically restrict the class of neural networks that we can consider; even neural networks that act as classifiers typically produce a real-valued vector as output, where each element k of the vector represents the probability of the input having label k.

IV. Algorithm

Algorithm 1: Checking Probabilistic Robustness.

Input: f: Neural network as a CAT function.
D: Input distribution.
ϵ: Probabilistic error bound.
k: Lipschitz constant.

Output: $\{\mathbf{T}, \mathbf{F}\}$

```
1  pf := ConstructProduct(f);
2  φ := ¬(‖f(x') − f(x)‖ ≤ k ∗ ‖x' − x‖));
3  poly := AbstractInterpret(pf, φ);
4  err := 0;
5  foreach p ∈ poly do
6  |   e := SampleAndEstimate(p, pf, φ, D);
7  |   err := err + e;
8  end foreach
9  if err > ϵ then
10 |   return F;
11 else
12 |   return T;
```

Algorithm 1 describes the procedure for checking the probabilistic robustness of a neural network f. f is input to the algorithm and is expressed in the form of a CAT function (see Section II). The other inputs to the algorithm are the probabilistic error bound ϵ, the Lipschitz constant k, and the input distribution D. D can either be represented as a closed form function, or as a probabilistic program, depending on the algorithm implementation. The algorithm outputs $\mathbf{T}$ (true) if f satisfies probabilistic robustness, and $\mathbf{F}$ (false) otherwise.

Our algorithm frames the problem of checking the probabilistic robustness of a neural network as a relational program verification problem [14]. Relational verification is defined as checking program properties or specifications that are expressed over pairs of program traces. For instance, probabilistic robustness requires comparing the outputs ($\|f(x') - f(x)\|$) generated by a neural network for pairs of inputs ($\|x' - x\|$). Such two-trace properties are also called *hyperproperties* [15].

A majority of program verification and analysis techniques are only applicable to single-trace properties. To be able to use such techniques for checking hyperproperties, a standard trick used in program verification is to construct a product program [16]. For a program P, a product program is constructed by creating a copy P' of P, where all the variables are renamed, and composing P and P' together to get program $P; P'$. A hyperproperty of the original program then corresponds to a single-trace property of the product program.

The first step of our algorithm is to construct a "product" neural network pf (line 1) by encoding two copies of the original network f side by side. Assume that the input and the output of the original neural network f are notated as $\bar{x}$ and $\bar{y}$, respectively. Then, intuitively, the product neural network (1) accepts the input $(\bar{x}, \bar{x}')$, (2) independently processes $\bar{x}$ and $\bar{x}'$, and (3) produces the output $(\bar{y}, \bar{y}')$, such that $\bar{y}{=}f(\bar{x})$ and $\bar{y}'{=}f(\bar{x}')$. This product construction enables us to use standard abstract interpretation techniques for checking a hyperproperty such as robustness. Note that, as we just discussed, any input for the product neural network represents a pair of inputs for the original neural network. In the rest of this section, we therefore use the term input to refer to a product neural network input.

In line 2, the algorithm assigns the temporary name ϕ to the property to be checked, that is, the negation of the Lipschitz property. The backwards abstract interpreter `AbstractInterpret` produces the set $poly$ (line 3) as an overapproximation of the set of inputs that satisfy ϕ. Since ϕ is the negation of the Lipschitz property, all the inputs NOT in $poly$ satisfy the Lipschitz property. Because `AbstractInterpret` is based on the powerset polyhedra abstract domain [17], [18], which uses a set of polyhedra to approximate a set of real-valued vectors, the set $poly$ produced by the abstract interpreter is a set of input polyhedra.

Next, for each input polyhedron p in $poly$, the algorithm applies importance sampling to improve the precision of the results. As we discussed above, each polyhedron p computed through abstract interpretation is an overapproximation of the set of inputs that satisfy ϕ (i.e., the set of inputs that violate the Lipschitz property). To reduce imprecision, the algorithm samples inputs from within p, and uses these samples to estimate the probability e of inputs in p satisfying ϕ. For each sample, the sampling procedure first checks if the distance between the two elements comprising the sample input is more than δ. If so, the sample is rejected. Otherwise, the sample is accepted. For each accepted sample, the sampling procedure checks if the sample satisfies ϕ. The probability estimate e is the average weighted probability of the samples satisfying ϕ, where the weighted probability depends on the size of p and on the input distribution D. Finally, after processing all polyhedra, the algorithm checks the value of err, which is the total probability of satisfying ϕ. If err is greater than ϵ, the probability of violating the Lipschitz property is greater than ϵ, neural network f is not probabilistically robust, and the algorithm returns $\mathbf{F}$ (lines 9–10). Otherwise, f satisfies the property, and the algorithm returns $\mathbf{T}$ (lines 11–12).

V. Conclusion

We presented probabilistic robustness, a novel formulation of robustness of neural networks that is practical, yet principled. Probabilistic robustness guarantees that a neural network is robust with at least $(1 - \epsilon)$ probability, given a real-world input probability distribution. In contrast to existing notions of robustness, probabilistic robustness focuses on a non-adversarial setting. We also presented an algorithm based on abstract interpretation and importance sampling for checking whether a neural network is probabilistically robust. We are currently implementing our algorithm and plan to evaluate the usefulness of our approach on real-world neural networks.

VI. Acknowledgments

This work was partially supported by NSF, under grants CCF-1161821 and 1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research. We thank the anonymous reviewers for their helpful feedback.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013. [Online]. Available: http://arxiv.org/abs/1312.6199

[2] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel, "Fairness through awareness," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12, 2012.

[3] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "Fairsquare: Probabilistic verification of program fairness," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017.

[4] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[5] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[6] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," 2017. [Online]. Available: http://arxiv.org/abs/1712.07107

[7] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16, 2016.

[8] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *Computer Aided Verification*, 2017.

[9] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification*, 2017.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77, 1977.

[11] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*, 2010.

[12] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretatiion," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[13] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 2011.

[14] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *FM 2011: Formal Methods*.

[15] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, 2010.

[16] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, ser. CSFW '04, 2004.

[17] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1978.

[18] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta, "Static analysis in disjunctive numerical domains," in *Proceedings of the 13th International Conference on Static Analysis*, ser. SAS'06, 2006.