The #DNN-Verification Problem: Counting Unsafe Inputs for Deep Neural Networks

Luca Marzari, Davide Corsi, Ferdinando Cicalese and **Alessandro Farinelli**
Department of Computer Science, University of Verona, Verona, Italy
luca.marzari@univr.it, davide.corsi@univr.it

Abstract

Deep Neural Networks are increasingly adopted in critical tasks that require a high level of safety, e.g., autonomous driving. While state-of-the-art verifiers can be employed to check whether a DNN is unsafe w.r.t. some given property (i.e., whether there is at least one unsafe input configuration), their yes/no output is not informative enough for other purposes, such as shielding, model selection, or training improvements. In this paper, we introduce the *#DNN-Verification* problem, which involves counting the number of input configurations of a DNN that result in a violation of a particular safety property. We analyze the complexity of this problem and propose a novel approach that returns the exact count of violations. Due to the #P-completeness of the problem, we also propose a randomized, approximate method that provides a provable probabilistic bound of the correct count while significantly reducing computational requirements. We present experimental results on a set of safety-critical benchmarks that demonstrate the effectiveness of our approximate method and evaluate the tightness of the bound.

1 Introduction

In recent years, the success of Deep Neural Networks (DNNs) in a wide variety of fields (e.g., games playing, speech recognition, and image recognition) has led to the adoption of these systems also in safety-critical contexts, such as autonomous driving and robotics, where humans safety and expensive hardware can be involved. A popular and successful example is "ACAS Xu", an airborne collision avoidance system for aircraft based on a DNN controller, which is now a standard DNN-verification benchmark.

A DNN is a non-linear function that maps a point in the input space to an output that typically represents an action or a class (respectively, for control and classification tasks). A crucial aspect of these DNNs lies in the concept of generalization. A neural network is trained on a finite subset of the input space, and at the end of this process, we expect it to find a pattern that allows making decisions in contexts never seen before. However, even networks that empirically perform well on a large test set can react incorrectly to slight perturbations in their inputs [Szegedy *et al.*, 2013]. While this is a negligible concern in simulated and controlled environments, it can potentially lead to catastrophic consequences for real safety-critical tasks, possibly endangering human health. Consequently, in recent years, part of the scientific communities devoted to machine learning and formal methods have joined efforts to develop DNN-Verification techniques that provide formal guarantees on the behavior of these systems [Liu *et al.*, 2021][Katz *et al.*, 2021].

Given a DNN and a safety property, a DNN verification tool should ideally either ensure that the property is satisfied for all the possible input configurations or identify a specific example (e.g., adversarial configuration) that violates the requirements. Given these complex functions' non-linear and non-convex nature, verifying even simple properties is proved to be an NP-complete problem [Katz *et al.*, 2017]. In literature, several works try to solve the problem efficiently either by *satisfiability modulo theories* (SMT) solvers [Liu *et al.*, 2021][Katz *et al.*, 2019] or by *interval propagation* methods [Wang *et al.*, 2021].

Although these methods show promising results, the current formulation, widely adopted for almost all the approaches, considers only the decision version of the formal verification problem, with the solution being a binary answer whose possible values are typically denoted `SAT` or `UNSAT`. `SAT` indicates that the verification framework found a specific input configuration, as a counterexample, that caused a violation of the requirements. `UNSAT`, in contrast, indicates that no such point exists, and then the safety property is formally verified in the whole input space. While an `UNSAT` answer does not require further investigations, a `SAT` result hides additional information and questions. For example, how many of such adversarial configurations exist in the input space? How likely are these misbehaviors to happen during a standard execution? Can we estimate the probability of running into one of these points? These questions can be better dealt with in terms of the problem of counting *the number of violations* to a safety property, a problem that might be important also in other contexts: (i) *model selection*: a counting result allows ranking a set of models to select the safest one. This model selection is impossible with a `SAT` or `UNSAT` type verifier, which does not provide any information to discriminate between two models which have both been found to vi-

olate the safety condition for at least one input configuration. (ii) *guide the training*: knowing the total count of violations for a particular safety property can help guiding the training of a deep neural network in a more informed fashion, for instance, minimizing this number over the training process. (iii) *estimating the probability of error*: the ratio of the total number of violations over the size of the input space provides an estimate of the probability of committing an unsafe action given a specific safety property.

Furthermore, by enumerating the violation points, it is possible to perform additional safety-oriented operations, such as: (iv) *shielding*: given the set of input configurations that violate a given safety property, we could adopt a more informative shielding mechanism that prevents unsafe actions.(v) *enrich the training phase*: if we can enumerate the violation configurations, we could add these configurations to the training (or to a memory buffer in deep reinforcement learning setup) to improve the training phase in a safe-oriented fashion. Motivated by the above questions and applications, previous works [Baluta *et al.*, 2019][Zhang *et al.*, 2021][Ghosh *et al.*, 2021] propose a *quantitative* analysis of neural networks, focusing on a specific subcategory of these functions, i.e., Binarized Neural Networks (BNN). However, violation points are generally not preserved in the binarization of a DNN to a BNN nor conversely in the relaxation of a BNN to a DNN [Zhang *et al.*, 2021].

To this end, in this paper, we introduce the *#DNN-Verification* problem, which is the extension of the decision DNN-Verification problem to the corresponding counting version. Given a general deep neural network (with continuous values) and a safety property, the objective is to count the exact number of input configurations that violate the specific requirement. We analyze the complexity of this problem and propose two solution approaches. In particular, in the first part of the paper, we propose an algorithm that is guaranteed to find the *exact* number of unsafe input configurations, providing a detailed theoretical analysis of the algorithm's complexity. The high-level intuition behind our method is to recursively shrink the domain of the property, exploiting the `SAT` or `UNSAT` answer of a standard formal verifier to drive the expansion of a tree that tracks the generated subdomains. Interestingly, our method can rely on any formal verifier for the decision problem, taking advantage of all the improvements to state-of-the-art and, possibly, to novel frameworks. As the analysis shows, our algorithm requires multiple invocations of the verification tool, resulting in significant overhead and becoming quickly unfeasible for real-world problems. For this reason, inspired by the work of [Gomes *et al.*, 2007] on *#SAT*, we propose an approximation algorithm for *#DNN-Verification*, providing provable (probabilistic) bounds on the correctness of the estimation.

In more detail, in this paper, we make the following contribution to the state-of-the-art:

- We propose an exact count formal algorithm to solve *#DNN-verification*, that exploits state-of-the-art decision tools as backends for the computation.
- We present `CountingProVe`, a novel approximation algorithm for *#DNN-verification*, that provides a bounded confidence interval for the results.
- We evaluate our algorithms on a standard benchmark, ACAS Xu, showing that `CountingProVe` is scalable and effective also for real-world problems.

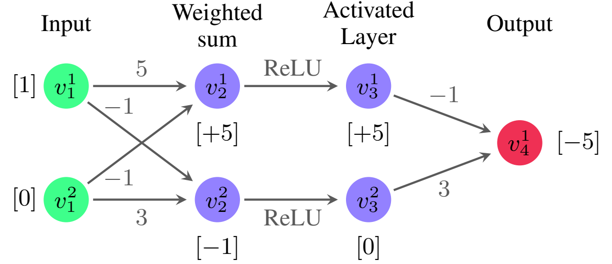

Figure 1: A simple example of a DNN $\mathcal{N}$ that will be used as a running example throughout the paper.

To the best of our knowledge, this is the first study to present *#DNN-verification*, the counting version of the decision problem of the formal verification for general neural networks without converting the DNN into a CNF.

2 Preliminaries

Deep Neural Networks (DNNs) are processing systems that include a collection of connected units called neurons, which are organized into one or more layers of parameterized non-linear transformations. Given an input vector, the value of each next hidden node in the network is determined by computing a linear combination of node values from the previous layer and applying a non-linear function node-wise (i.e., the activation function). Hence, by propagating the initial input values through the subsequent layers of a DNN, we obtain either a label prediction (e.g., for image classification tasks) or a value representing the index of an action (e.g., for a decision-making task). Fig. 1 shows a concrete example of how a DNN computes the output. Given an input vector $V_1 = [1, 0]^T$, the weighted sum layer computes the value $V_2 = [+5, -1]^T$. This latter is the input for the so-called activation function *Rectified Linear Unit* (ReLU), which computes the following transformation, $y = ReLU(x) = \max(0, x)$. Hence, the result of this activated layer is the vector $V_3 = [+5, 0]^T$. Finally, the network's single output is again computed as a weighted sum, giving us the value -5.

2.1 The DNN-Verification Problem

An instance of the *DNN-Verification* problem (in its standard decision form) is given by a trained DNN $\mathcal{N}$ together with a safety property, typically expressed as an input-output relationship for $\mathcal{N}$ [Liu *et al.*, 2021]. In more detail, a property is a tuple that consists of a precondition, expressed by a predicate $\mathcal{P}$ on the input, and a postcondition, expressed by a predicate $\mathcal{Q}$ on the output of $\mathcal{N}$. In particular, $\mathcal{P}$ defines the possible input values we are interested in—aka the input configurations—(i.e., the domain of the property), while $\mathcal{Q}$ represents the output results we aim to guarantee formally for at least one of the inputs that satisfy $\mathcal{P}$. Then, the problem

consists of verifying whether there exists an input configuration in $\mathcal{P}$ that, when fed to the DNN $\mathcal{N}$, produces an output satisfying $\mathcal{Q}$.

Definition 1 (*DNN-Verification Problem*).

Input*: A tuple $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$, where $\mathcal{N}$ is a trained DNN, $\mathcal{P}$ is precondition on the input, and $\mathcal{Q}$ a postcondition on the output.*

Output*: `SAT` if $\exists\, x \mid \mathcal{P}(x) \wedge \mathcal{Q}(\mathcal{N}(x))$ and `UNSAT` otherwise, indicating that no such x exists.*

As an example of how this problem can be employed for checking the existence of unsafe input configurations for a DNN, suppose we aim to verify that the DNN $\mathcal{N}$ of Fig. 1, for any input in the interval $[0, 1]$, outputs a value greater than or equal 0. Hence, we define $\mathcal{P}$ as the predicate on the input vector $\mathbf{v} = (v_1^1, v_2^1)$ which is true iff $\mathbf{v} \in [0, 1] \times [0, 1]$, and $\mathcal{Q}$ as the predicate on the output v_4^1 which is true iff $v_4^1 = \mathcal{N}(v_1^1, v_2^1) < 0$, that is we set $\mathcal{Q}$ to be the negation of our desired property. Then, solving the *DNN-Verification Problem* on the instance $(\mathcal{N}, \mathcal{P}, \mathcal{Q})$ we get `SAT` iff there is counterexample that violates our property.

Since for the input vector $\mathbf{v} = (1, 0)$ (also reported in Fig. 1), the output of $\mathcal{N}$ is < 0, in the example, the result of the *DNN-Verification Problem* (with the postcondition being the negated of the desired property) is `SAT`, meaning that there exists at least a single input configuration (v_1^1, v_2^1) that satisfies $\mathcal{P}$ and for which $\mathcal{N}(v_1^1, v_2^1) < 0$. As a result, we can say that the network is not safe for the desired property.[1]

3 #DNN-Verification and Exact Count

In this section, we first provide a formal definition for *#DNN-Verification*; hence, we propose an algorithm to solve the problem by exploiting any existing formal verification tool. Finally, we proceed with a theoretical discussion on the complexity of the problem, highlighting the need for approximate approaches.

3.1 Problem Formulation

Given a tuple $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$, as in Definition 1, we let $\Gamma(\mathcal{R})$ denote the set of *all* the input configurations for $\mathcal{N}$ satisfying the property defined by $\mathcal{P}$ and $\mathcal{Q}$, i.e.

$$\Gamma(\mathcal{R}) = \left\{ x \mid \mathcal{P}(x) \wedge \mathcal{Q}(\mathcal{N}(x)) \right\}$$

Then, the *#DNN-Verification* consists of computing the cardinality of $\Gamma(\mathcal{R})$.

Definition 2 (*#DNN-Verification Problem*).

Input*: A tuple $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$, as in Definition 1.*

Output*: $|\Gamma(\mathcal{R})|$*

For the purposes discussed in the introduction, rather than the cardinality of $\Gamma(\mathcal{R})$, it is more useful to define the problem in terms of the ratio between the cardinality of Γ and the cardinality of the set of inputs satisfying $\mathcal{P}$. We refer to this ratio as the *violation rate (VR)*, and study the result of the #DNN-Verification problem in terms of this equivalent measure.

[1] More details about the problem and state-of-the-art methods for solving it can be found in the supplementary material.

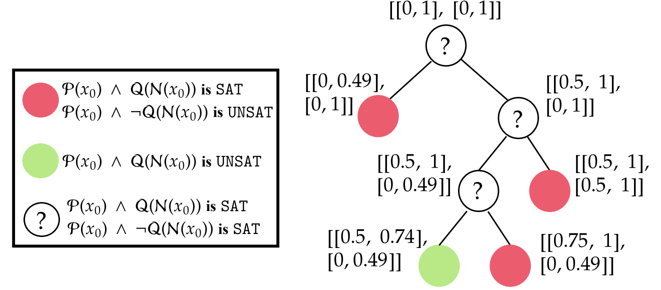

Figure 2: Example execution of exact count for a particular $\mathcal{N}$ and safety property (assuming a discretization factor of 0.01).

Definition 3 (*Violation Rate (VR)*)**.** *Given an instance of the DNN-Verification problem $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$ we define the violation rate as*

$$VR = \frac{|\Gamma(\mathcal{R})|}{|\{x \mid \mathcal{P}(x)\}|}$$

Although, in general, DNNs can handle continuous spaces, in the following sections (and for the analysis of the algorithms), without loss of generality, we assume the input space to be discrete. We remark that for all practical purposes, this is not a limitation since we can assume that the discretization is made to the maximum resolution achievable with the number of decimal digits a machine can represent. It is crucial to point out that discretization is not a requirement of the approaches proposed in this work. In fact, supposing to have a backend that can deal with continuous values, our solutions would not require such discretization.

3.2 Exact Count Algorithm for #DNN-Verification

We now present an algorithm to solve the exact count of *#DNN-Verification*.

The algorithm recursively splits the input space into two parts of equal size as long as it contains both a point that violates the property (i.e., $(\mathcal{N}, \mathcal{P}, \mathcal{Q})$ is a `SAT`-instance for *DNN-Verification* problem) and a point that satisfies it (i.e., $(\mathcal{N}, \mathcal{P}, \neg\mathcal{Q})$ is a `SAT`-instance for *DNN-Verification* problem).[2] The leaves of the recursion tree of this procedure correspond to a partition of the input space into parts where the violation rate is either 0 or 1. Therefore, the overall violation rate is easily computable by summing up the sizes of the subinput spaces in the leaves of violation rate 1. Fig. 2 shows an example of the execution of our algorithm for the DNN and the safety property presented in Sec. 2. We want to enumerate the total number of input configurations $\mathbf{v} = (v_1^1, v_1^2)$ where both v_1^1 and v_1^2 satisfy $\mathcal{P}$ (i.e., they lie in the interval $[0, 1]$) and such that the output $v_4^1 = \mathcal{N}(\mathbf{v})$ is a value strictly less than 0, i.e., $\mathbf{v}$ violates the safety property.

The algorithm starts with the whole input space and checks if at least one point exists that outputs a value strictly less

[2] Any state-of-the-art sound and complete verifier for the decision problem can be used to solve these instances. In fact, our method works with any state-of-the-art verifiers, although, using a verifier that is not complete can lead to over-approximation in the final count.

than zero. The exact count method checks the predicate $\exists x \mid \mathcal{P}(x) \land \mathcal{Q}(\mathcal{N}(x))$ with a verification tool for the decision problem. If the result is UNSAT, then the property holds in the whole input space, and the algorithm returns a VR of 0%. Otherwise, if the verification tool returns SAT, at least one single violation point exists, but we cannot assert how many similar configurations exist in the input space. To this end, the algorithm checks another property equivalent to the original one, thus negating the postcondition (i.e., $\neg\mathcal{Q} = v_1^4 \geq 0$). Here, we have two possible outcomes:

- $\mathcal{P}(x_0) \land \neg\mathcal{Q}(\mathcal{N}(x_0))$ is UNSAT implies that all possible $x_0 = (v_1^1, v_1^2)$ that satisfy $\mathcal{P}$, output a value strictly less than 0, violating the safety property. Hence, the algorithm returns a 100% of *VR* in the input area represented by $\mathcal{P}$. This situation is depicted with the red circle in Fig. 2.
- $\mathcal{P}(x_0) \land \neg\mathcal{Q}(\mathcal{N}(x_0))$ is SAT implies that there is at least one input configuration $\mathbf{v} = (v_1^1, v_1^2)$ satisfying $\mathcal{P}$ and such that $\mathcal{N}(\mathbf{v}) \geq 0$. Therefore, the algorithm cannot assert anything about the violated portion of the area since there are some input points on which $\mathcal{N}$ generates outputs greater or equal to zero and some others that generate a result strictly less than zero. Hence, the procedure splits the input space into two parts, as depicted in Fig. 2.

This process is repeated until the algorithm reaches one of the base cases, such as a situation in which either $\mathcal{P}(x) \land \mathcal{Q}(\mathcal{N}(x))$ is not satisfiable (i.e., all the current portion of the input space is safe), represented by a green circle in Fig.2), or $\mathcal{P}(x) \land \neg\mathcal{Q}(\mathcal{N}(x))$ is not satisfiable (i.e., the whole current portion of the input space is unsafe), represented by a red circle. Note that the option of obtaining UNSAT on both properties is not possible.

Finally, the algorithm of exact count returns the *VR* as the ratio between the unsafe areas (i.e., the red circles) and the original input area. In the example of Fig. 2, assuming a 2-decimal-digit discretization, we obtain a total number of violation points equal to 8951. Normalizing this value by the initial total number of points (10201), and considering the percentage value, we obtain a final *VR* of 87.7%. Moreover, the *Violation Rate* can also be interpreted as the probability of violating a safety property using a particular DNN $\mathcal{N}$. In more detail, uniformly sampling 10 random input vectors for our DNN $\mathcal{N}$ in the intervals described by $\mathcal{P}$, 8 over 10 with high probability violates the safety property.

3.3 Hardness of #DNN-Verification

It is known that finding even just one input configuration (also referred to as violation point) that violates a given safety property is computationally hard since the *DNN-Verification* problem is *NP-Complete* [Katz *et al.*, 2017]. Hence, counting and enumerating all the violation points is expected to be an even harder problem.

Theorem 1. *The #DNN-Verification is #P-Complete.*

Note that, in general, the counting version of an *NP-complete* problem is not necessarily #P-complete. We are able to prove Theorem 1 by showing that the hardness proof for *DNN-Verification* in [Katz *et al.*, 2017] provides a bijection between an assignment that satisfies a *3-CNF* formula and a violation point in given input space (we include the details is in the supplementary material). Hence, *#DNN-Verification* turns out to be *#P-Complete* as *#3-SAT* [Valiant, 1979].

Algorithm 1 CountingProVe

1: **Input:** $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$, t (# of repetitions), m (# of violation points sampled per iteration), $\beta > 0$ (error tolerance factor).
2: **Output:** lower bound of the violation rate.
3: **for** t=1 to t **do**
4: $\quad VR_t \leftarrow 100\%$, $s \leftarrow 0$
5: $\quad$ **while** $Timeout(ExactCount(\mathcal{R}))$ **do**
6: $\quad\quad S \leftarrow SampleViolationPoints(\mathcal{R}, m)$
7: $\quad\quad median \leftarrow ComputeMedian(S, node_i)$
8: $\quad\quad node_{i,0}, node_{i,1} \leftarrow SplitInterval(node_i, median)$
9: $\quad\quad side \leftarrow$ a random value chosen uniformly from {0,1}
10: $\quad\quad \mathcal{P} \leftarrow UpdateP(\mathcal{P}, node_{i,side})$
11: $\quad\quad s \leftarrow s + 1$
12: $\quad$ **end while**
13: $\quad VR_t \leftarrow 2^{s-\beta} \cdot ExactCount(\mathcal{R}) \cdot \prod_{i=1}^{s} \alpha_i$
14: $\quad VR \leftarrow min(VR_t, VR)$
15: **end for**
16: **return** *VR*

4 CountingProVe for Approximate Count

In view of the #P-completeness of the #DNN-Verification problem, it is not surprising that the time complexity of the algorithm for the exact count worsens very fast when the size of the instance increases. In fact also moderately large networks are intractable with this approach. To overcome these limitations while still providing guarantees on the quality of the results, we propose a randomized-approximation algorithm, CountingProVe (presented in Algorithm 1).

To understand the intuition behind our approach, consider Fig. 2. If we could assume that each split distributed the violation points evenly in the two subinstances it produces, for computing the number of violation points in the whole input space it would be enough to compute the number of violation points in the subspace of a single leaf and multiplying it by 2^s (which represents the number of leaves). Since we cannot guarantee a perfectly balanced distribution of the violation points among the leaves, we propose to use a heuristic strategy to split the input domain, balancing the number of violation points in expectation. This strategy allows us to estimate the count and a provable confidence interval on the actual violation rate. In more detail, our algorithm receives as input the tuple for a DNN-Verification problem (i.e., $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$) and to obtain a precise estimate of the *VR*, performs t iterations of the following procedure.

Initially, we assume that, in the whole input domain, the safety property does not hold, i.e., we have a 100% of *Violation Rate*. The idea is to repeatedly simplify the input space by performing a sequence of multiple splits, where each i-th split implied a reduction of the input space of a factor α_i. At the end of s simplifications, the goal is to obtain an exact count of unsafe input configurations that can be used to estimate the VR of the entire input space. Specifically, Algorithm 1 presents the pseudo-code for the heuristic approach.

Before splitting, the algorithm attempts to use the exact count but stops if not completed within a fixed timeout which we set to just a fraction of a second (line 5). Inside the loop, the procedure $SampleViolationPoints(\mathcal{R}, m)$ samples m violation points from the subset of the input space described in $\mathcal{P}$ (using a uniform sampling strategy) and saves them in S.

After line 6, the algorithm has an estimation of how many violation points there are in the portion of the input space under consideration[3]. The idea is to simplify one dimension of the hyperrectangle cyclically while keeping the number of violation points balanced in the two portions of the domain we aim to split (i.e., as close to $|S|/2$ as possible). Specifically, $ComputeMedian(S, node_i)$ (line 7) computes the median of the values along the dimension of the node chosen for the split. $SplitInterval(node_i, median)$ splits the node into two parts $node_{i,0}$ and $node_{i,1}$ according to the median computed. For instance, suppose we have an interval of $[0, 1]$ and the median value is 0.33. The two parts obtained by calling $SplitInterval(node_i, median)$ are $[0, 0.33]$ and $(0.33, 1]$. The algorithm proceeds randomly, selecting the side to consider from that moment on and discarding the other part. Hence, it updates the input space intervals to be considered for the safety property $\mathcal{P}$ (lines 9-10). Finally, the variable s that represents the number of splits made during the iteration is incremented. At the end of the while loop (line 12), the VR is computed by multiplying the result of the exact count by $2^{s-\beta}$, and for $\prod_{i=1}^{s}\alpha_i$. The first term ($2^{s-\beta}$) considers the fact that we balanced the *VR* in our tree, hence selecting one path and multiplying by 2^s, we get a representative estimate of the whole starting input area. β is a tolerance error factor, and finally, $\prod_{i=1}^{s}\alpha_i$ describes how we simplified the input space during the s splits made (line 13). Finally, the algorithm refines the lower bound (line 14).

The crucial aspect of `CountingProVe` is that even when the splitting method is arbitrarily poor, the bound returned is provably correct (from a probabilistic point of view), see next section for more details. Moreover, the correctness of our algorithm is independent on the number of samples used to compute the median value. However, using a poor heuristic (i.e., too few samples or a suboptimal splitting technique), the bound's quality can change, as the lower bound may get farther away from the actual *Violation Rate*. We report in the supplementary material an ablation study that focuses on the impact of the heuristic on the results obtained.

4.1 A Provable Lower Bound

In this section, we show that the randomized-approximation algorithm `CountingProVe` returns a correct lower bound for the *Violation Rate* with a probability greater (or equal) than $(1-2^{-\beta t})$. In more detail, we demonstrate that the error of our approximation decreases exponentially to zero and that it does not depend on the number of violation points sampled during the simplification process nor on the heuristic for the node splitting.

Theorem 2. *Given the tuple $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$, the Violation Rate returned by the randomized-approximation algorithm* `CountingProVe` *is a correct lower bound with a probability $\geq (1 - 2^{-\beta t})$.*

Proof. Let $VR^* > 0$ be the actual violation rate. Then, `CountingProVe` returns an incorrect lower bound, if for each iteration, $VR > VR^*$. The key of this proof is to show that for any iteration, $Pr(VR > VR^*) < 2^{-\beta}$.

Fix an iteration t. The input space described by $\mathcal{P}$ and under consideration within the while loop is repeatedly simplified. Assume we have gone through a sequence of s splits, where, as reported in Sec.4, the i-th split implied a reduction of the solution space of a factor α_i. For a violation point σ, we let Y_σ be a random variable that is 1 if σ is also a violation point in the restriction of the input space that has been (randomly) obtained after s splits (and that we refer to as the solution space of the leaf ℓ).

Hence, the *VR* obtained using an exact count method for a particular ℓ is $VR_\ell = \frac{\sum_{\sigma\in\Gamma} Y_\sigma}{A_\ell}$, i.e., the ratio between the number of violation points and the number of points in ℓ (A_ℓ). Then our algorithm returns as an estimate of the total *VR* computed by:

$$VR = 2^{s-\beta} \cdot VR_\ell \cdot \prod_{i=1}^{s} \alpha_i = 2^{s-\beta} \cdot \frac{\sum_{\sigma\in\Gamma} Y_\sigma}{A_\ell} \cdot \prod_{i=1}^{s} \alpha_i \quad (1)$$

Let **s** denote the sequence of s random splits followed to reach the leaf ℓ. We are interested in $\mathbb{E}[VR] = \mathbb{E}[\mathbb{E}[VR|\mathbf{s}]]$. The inner conditional expectation can be written as:

$$\mathbb{E}[VR \mid \mathbf{s}] = \mathbb{E}\Big[2^{s-\beta} \cdot \frac{\sum_{\sigma\in\Gamma} Y_\sigma}{A_\ell} \cdot \prod_{i=1}^{s} \alpha_i \mid \mathbf{s}\Big] \quad (2)$$

$$= \mathbb{E}\Big[2^{s-\beta} \cdot \frac{\sum_{\sigma\in\Gamma} Y_\sigma}{A_{Tot}} \mid \mathbf{s}\Big] \quad (3)$$

$$= \frac{2^{s-\beta}}{A_{Tot}} \sum_{\sigma} \mathbb{E}[Y_\sigma = 1 \mid \mathbf{s}\,] \quad (4)$$

$$= \frac{2^{s-\beta}}{A_{Tot}} \sum_{\sigma} 2^{-s} \quad (5)$$

$$= 2^{-\beta} \frac{\sum_{\sigma\in\Gamma} 1}{A_{Tot}} = 2^{-\beta} VR^* \quad (6)$$

where the equality in (2) follows from rewriting *VR* using (1). Equality (3) follows from (2) using the relation $A_{Tot} = \frac{A_\ell}{\prod_{i=1}^{s}\alpha_i}$. Then we have (4) that follows from (3) by using the linearity of the expectation. (5) follows from (4) since, in each split, we choose the side to take independently and with probability $1/2$. Finally, (6) follows by recalling that $\sum_{\sigma\in\Gamma} 1$ is the total number of violations in the whole input space, hence $VR^* = \frac{\sum_{\sigma\in\Gamma} 1}{A_{Tot}}$. Therefore, we have

$$\mathbb{E}[VR] = \mathbb{E}[\mathbb{E}[VR|\mathbf{s}]] = \mathbb{E}[2^{-\beta} VR^*] = 2^{-\beta} VR^*.$$

Finally, by using Markov's inequality, we obtain that

$$Pr(VR > VR^*) < \frac{\mathbb{E}[VR]}{VR^*} = \frac{2^{-\beta} VR^*}{VR^*} = 2^{-\beta}.$$

[3] if the m solutions are not found, the algorithm proceeds by considering only the violation points discovered or splitting at random.

Repeating the process t times, we have a probability of overestimation equal to $2^{-\beta t}$. This proves that the *Violation Rate* returned by `CountingProVe` is correct with a probability $\geq (1 - 2^{-\beta t})$. □

4.2 A Provable Confidence Interval of the VR

This section shows how a provable confidence interval can be defined for the *VR* using `CountingProVe`. In particular, recalling the definition of *Violation Rate* (Def. 3), it is possible to define a complementary metric, the *safe rate (SR)*, counting all the input configurations that do not violate the safety property (i.e., provably safe). In particular, we define:

Definition 4. *(Safe Rate (SR)) Given an instance of the DNN-Verification problem $\mathcal{R} = \langle \mathcal{N}, \mathcal{P}, \mathcal{Q} \rangle$ we define the Safe Rate as the ratio between the number of safe points and the total number of points satisfying $\mathcal{P}$. Formally:*

$$SR = \frac{|\{x \mid \mathcal{P}(x)\} \setminus \Gamma(\mathcal{R})|}{|\{x \mid \mathcal{P}(x)\}|} = \frac{|\Gamma(\mathcal{N}, \mathcal{P}, \neg\mathcal{Q})|}{|\{x \mid \mathcal{P}(x)\}|}$$

where the numerator indicates the sum of non-violation points in the input space. From the second expression, it is easy to see that `CountingProVe` can be used to compute a lower bound of the *SR*, by running Alg. 1 on the instance $(\mathcal{N}, \mathcal{P}, \neg\mathcal{Q})$.

Theorem 3. *Given a Deep Neural Network $\mathcal{N}$ and a safety property $\langle \mathcal{P}, \mathcal{Q} \rangle$, complementing the lower bound of the Safe Rate obtained using `CountingProVe`, which is correct with a probability $\geq (1 - 2^{-\beta t})$, we obtain an upper bound for the Violation Rate with the same probability.*

Proof. Suppose we compute the *Safe Rate* with `CountingProVe`. From Theorem 2, we know that the probability of overestimating the real *SR* tends exponentially to zero as the iterations t grow. Hence, $Pr(SR > SR^*) < 2^{-\beta t}$. We now consider the *Violation Rate* as the complementary metric of the *Safe Rate*, and we write $VR = 1 - SR$ (as the SR is a value in the interval $[0, 1]$). We want to show that the probability that the *VR*, computed as $VR = 1 - SR$, underestimates the real *Violation Rate* (VR^*) is the same as theorem 2.

Suppose that at the end of t iterations, we have a $VR < VR^*$. This would imply by definition that $1 - SR < 1 - SR^*$, i.e., that $SR > SR^*$. However, from Theorem 2, we know that the probability that `CountingProVe` returns an incorrect lower bound at each iteration is $2^{-\beta t}$. Hence, we obtained that the *VR* computed as $VR = 1 - SR$ is a correct upper bound with the probability (1 - $2^{-\beta t}$) as desired. □

These results allow us to obtain a confidence interval for the *Violation Rate*[4], namely, from Theorems 2 and 3 we have:

Lemma 4. *`CountingProVe` can compute both a correct lower and upper bound, i.e., a correct confidence interval for the VR, with a probability $\geq (1 - 2^{-\beta t})$.*

[4] notice that the same formulation holds for the complementary metrics (i.e., safe rate).

4.3 CountingProVe is Polynomial

To conclude the analysis of our randomized-approximation algorithm, we discuss some additional requirements to guarantee that our approach runs in polynomial time. Let N denote the size of the instance. It is easy to see that by choosing both the timeout applied to each run of the exact count algorithm used in line 5 and the number m of points sampled in line 6 to be polynomially bounded in N then each iteration of the for loop is also polynomial in N. Moreover, if each split of the input area encoded by $\mathcal{P}$ performed in lines 7-10 is guaranteed to reduce the size of the instance by at least some fraction $\gamma \in (0, 1)$, then after $s = \Theta(\log N)$ splits the instance in the leaf ℓ has size $O(\frac{N}{2^s}) = O(1)$. Hence, we can exploit an exponential time formal verifier to solve the instance in the leaf ℓ, and the total time will be polynomial in N.

5 Experimental Results

In this section, we guide the reader to understand the importance and impact of this novel encoding for the verification problem of deep neural networks. In particular, in the first part of our experiments, we show how the problem's computational complexity impacts the run time of exact solvers, motivating the use of an approximation method to solve the problem efficiently. In the second part, we analyze a concrete case study, ACAS Xu [Katz *et al.*, 2017], to explain why finding all possible unsafe configurations is crucial in a realistic safety-critical domain. All the data are collected on a commercial PC running Ubuntu 22.04 LTS equipped with Nvidia RTX 2070 Super and an Intel i7-9700k. In particular, for the exact counter, we rely as backend on the formal verification tool *BaB* [Bunel *et al.*, 2018] available on "NeuralVerification.jl" and developed as part of the work of [Liu *et al.*, 2021]. While, as exact count for `CountingProVe`, we rely on *ProVe* [Corsi *et al.*, 2021] given its key feature of exploiting parallel computation on GPU. In our experiments with `CountingProVe`, we set $\beta = 0.02$ and $t = 350$ in order to obtain a correctness confidence level greater or equal to 99% (refer to Theorem 2).

Table 1 summarizes the results of our experiments, evaluating the proposed algorithms (i.e., the exact counter and `CountingProVe`) on different benchmarks. Our results show the advantage of using an approximation algorithm to obtain a provable estimate of the portion of the input space that violates a safety property. We discuss the results in detail below. The code used to collect the results and several additional experiments and discussions on the impact of different hyperparameters and backends for our approximation are available in the supplementary material (available here).

Scalability Experiments In the first two blocks of Tab. 1, we report the experiments related to the scalability of the exact counters against our approximation method `CountingProVe`, showing how the #DNN-Verification problem becomes immediately infeasible, even for small DNNs. In more detail, we collect seven different random models (i.e., using random seeds) with different levels of violation rates for the same safety property, which consists of all the intervals of $\mathcal{P}$ in the range $[0, 1]$, and a postcondition

Instance	**Exact Count** BaB		`CountingProVe` (confidence $\geq 99\%$)		
	Violation Rate	Time	Interval confidence VR	Size	Time
Model_2_20	20.78%	234 min	[19.7%, 22.6%]	2.9%	42 min
Model_2_56	55.22%	196 min	[54.13%, 57.5%]	3.37%	34 min
Model_2_68	68.05%	210 min	[66.21%, 69.1%]	2.89%	45 min
Model_5_09	–	24 hrs	[8.42%, 13.2%]	4.78%	122 min
Model_5_50	–	24 hrs	[48.59%, 52.22%]	3.63%	124 min
Model_5_95	–	24 hrs	[91.73%, 96.23%]	4.49%	121 min
Model_10_76	–	24 hrs	[74.25%, 77.23%]	3.98%	300 min
ϕ_2 ACAS Xu_2.1	–	24 hrs	[0.45%, 5.01%]	4.56%	246 min
ϕ_2 ACAS Xu_2.3	–	24 hrs	[1.23%, 4.21%]	2.98%	241 min
ϕ_2 ACAS Xu_2.4	–	24 hrs	[0.74%, 3.43%]	2.68%	243 min
ϕ_2 ACAS Xu_2.5	–	24 hrs	[1.67%, 4.10%]	2.42%	240 min
ϕ_2 ACAS Xu_2.7	–	24 hrs	[2.35%, 5.22%]	2.87%	240 min

Table 1: Comparison of `CountingProVe` and exact counter on different benchmark setups. The first block shows the results on our benchmark properties, where every instance is in the form Model_ρ_ψ, where ρ is the size of the input space and ψ is the id of the specific DNN. The last block reports the results on the Acas Xu ϕ_2 benchmark. Full results on ϕ_2 and another property in the AppendixD.

$\mathcal{Q}$ that encodes a strictly positive output. In the first block, all the models have two hidden layers of 32 nodes activated with $ReLU$ and two, five, and ten-dimensional input space, respectively. Our results show that for the models with two input neurons, the exact counter returns the violation rate in about 3.3 hours, while our approximation in less than an hour returns a provable tight ($\sim 3\%$) confidence interval of the input area that presents violations. Crucially, as the input space grows, the exact counters reach the timeout (fixed after 24 hours), failing to return an exact answer. `CountingProVe`, on the other hand, in about two hours, returns an accurate confidence interval for the violation rate, highlighting the substantial improvement in the scalability of this approach. In the supplementary material, we report additional experiments and discussions on the impact of different hyperparameters for the estimate of `CountingProVe`.

ACAS Xu Experiments The ACAS Xu system is an airborne collision avoidance system for aircraft considered a well-known benchmark and a standard for formal verification of DNNs [Liu *et al.*, 2021][Katz *et al.*, 2017][Wang *et al.*, 2018]. It consists of 45 different models composed of an input layer taking five inputs, an output layer generating five outputs, and six hidden layers, each containing 50 neurons activated with $ReLU$. To show that the count of all the violation points can be extremely relevant in a safety-critical context, we focused on the property ϕ_2, on which, for 34 over 45 models, the property does not hold. In detail, the property ϕ_2 describes the scenario where if the intruder is distant and is significantly slower than the ownship, the score of a Clear of Conflict (COC) can never be maximal (more detail are reported here [Katz *et al.*, 2017]). We report in the last block of Tab. 1 the results of this evaluation. To understand the impact of the #DNN-Verification problem, let us focus, for example, on the *ACAS Xu_2.7* model. As reported in Tab. 1, `CountingProVe` returns a provable lower bound for the violation rate of at least a 2.35%. This means that assuming a 3-decimal-digit discretization, our approximation counts at least 23547 violation points compared to a formal verifier that returns a single counterexample (i.e., a single violation point). Note that a state-of-the-art verifier that returns only `SAT` or `UNSAT` does not provide any information about the amount of possible unsafe configurations considered by the property.

6 Discussion

In this paper, we first present the *#DNN-Verification*, the problem of counting the number of input configurations that generates a violation of a given safety property. We analyze the complexity of this problem, proving the #P-completness and highlighting why it is relevant for the community. Furthermore, we propose an exact count approach that, however, inherits the limitations of the formal verification tool exploited as a backend and struggles to scale on real-world problems. Crucially, we present an alternative approach, `CountingProVe`, which provides an approximated solution with formal guarantees on the confidence interval. Finally, we empirically analyze our algorithms on a set of benchmarks, including a real-world problem, ACAS Xu, considered a standard in the formal verification community.

Moving forward, we plan to investigate possible optimizations in order to improve the performance of `CountingProVe`, for example, by improving the node selection and the bisection strategies for the interval; or by exploiting the result of #DNN-Verification to optimize the system's safety during the training loop.

Contribution Statement

The authors Luca Marzari and Davide Corsi contributed equally to this work.

References

[Amir *et al.*, 2023] Guy Amir, Davide Corsi, Raz Yerushalmi, Luca Marzari, David Harel, Alessandro Farinelli, and Guy Katz. Verifying learning-based robotic navigation systems. In *29th International Conference, TACAS 2023*, pages 607–627. Springer, 2023.

[Baluta *et al.*, 2019] 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*, 2019.

[Bunel *et al.*, 2018] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[Corsi *et al.*, 2021] Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In *Uncertainty in Artificial Intelligence*, pages 333–343. PMLR, 2021.

[Ghosh *et al.*, 2021] Bishwamittra Ghosh, Debabrota Basu, and Kuldeep S Meel. Justicia: A stochastic sat approach to formally verify fairness. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 7554–7563, 2021.

[Gomes *et al.*, 2007] Carla P Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *IJCAI*, volume 2007, pages 2293–2299, 2007.

[Gomes *et al.*, 2021] Carla P Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of satisfiability*, pages 993–1014. IOS press, 2021.

[Katz *et al.*, 2017] 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.

[Katz *et al.*, 2019] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, 2019.

[Katz *et al.*, 2021] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, pages 1–30, 2021.

[Liu *et al.*, 2021] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.

[Lomuscio and Maganti, 2017] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[Marchesini *et al.*, 2021] Enrico Marchesini, Davide Corsi, and Alessandro Farinelli. Benchmarking safe deep reinforcement learning in aquatic navigation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5590–5595, 2021.

[Marchesini *et al.*, 2022] Enrico Marchesini, Davide Corsi, and Alessandro Farinelli. Exploring safer behaviors for deep reinforcement learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2022.

[Marzari *et al.*, 2022] Luca Marzari, Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. Curriculum learning for safe mapless navigation. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 766–769, 2022.

[Marzari *et al.*, 2023] Luca Marzari, Enrico Marchesini, and Alessandro Farinelli. Online safety property collection and refinement for safe deep reinforcement learning in mapless navigation. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023.

[Moore *et al.*, 2009] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to interval analysis*. SIAM, 2009.

[Szegedy *et al.*, 2013] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[Tjeng *et al.*, 2018] Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2018.

[Valiant, 1979] Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Wang *et al.*, 2018] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems*, 31, 2018.

[Wang *et al.*, 2021] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[Zhang *et al.*, 2021] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, and Taolue Chen. Bdd4bnn: a bdd-based quantitative analysis framework for binarized neural networks. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 175–200. Springer, 2021.