Quantitative Verification of Neural Networks and Its Security Applications

Teodora Baluta
teobaluta@comp.nus.edu.sg
National University of Singapore

Shiqi Shen
shiqi04@comp.nus.edu.sg
National University of Singapore

Shweta Shinde*
shwetas@eecs.berkeley.edu
University of California, Berkeley

Kuldeep S. Meel
meel@comp.nus.edu.sg
National University of Singapore

Prateek Saxena
prateeks@comp.nus.edu.sg
National University of Singapore

ABSTRACT

Neural networks are increasingly employed in safety-critical domains. This has prompted interest in verifying or certifying logically encoded properties of neural networks. Prior work has largely focused on checking existential properties, wherein the goal is to check whether there exists any input that violates a given property of interest. However, neural network training is a stochastic process, and many questions arising in their analysis require probabilistic and quantitative reasoning, i.e., estimating how many inputs satisfy a given property. To this end, our paper proposes a novel and principled framework to quantitative verification of logical properties specified over neural networks. Our framework is the first to provide *PAC*-style soundness guarantees, in that its quantitative estimates are within a controllable and bounded error from the true count. We instantiate our algorithmic framework by building a prototype tool called NPAQ[1]that enables checking rich properties over binarized neural networks. We show how emerging security analyses can utilize our framework in 3 applications: quantifying robustness to adversarial inputs, efficacy of trojan attacks, and fairness/bias of given neural networks.

ACM Reference Format:
Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. 2019. Quantitative Verification of Neural Networks and Its Security Applications. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354245

1 INTRODUCTION

Neural networks are witnessing wide-scale adoption, including in domains with the potential for a long-term impact on human society. Examples of these domains are criminal sentencing [1], drug discovery [87], self-driving cars [11], aircraft collision avoidance systems [48], robots [10], and drones [37]. While neural networks achieve human-level accuracy in several challenging tasks such as image recognition [43, 50, 80] and machine translation [8, 79], studies show that these systems may behave erratically in the wild [7, 12, 28, 31, 61, 62, 73, 83, 85].

Consequently, there has been a surge of interest in the design of methodological approaches to verification and testing of neural networks. Early efforts focused on *qualitative* verification wherein, given a neural network N and property P, one is concerned with determining whether there exists an input I to N such that P is violated [23, 26, 45, 49, 58, 65, 68, 72]. While such certifiability techniques provide value, for instance in demonstrating the existence of adversarial examples [39, 62], it is worth recalling that the designers of neural network-based systems often make a statistical claim of their behavior, i.e., a given system is claimed to satisfy properties of interest with high probability but not always. Therefore, many analyses of neural networks require *quantitative* reasoning, which determines how many inputs satisfy P.

It is natural to encode properties as well as conditions on inputs or outputs as logical formulae. We focus on the following formulation of *quantitative verification*: Given a set of neural networks $\mathcal{N}$ and a property of interest P defined over the union of inputs and outputs of neural networks in $\mathcal{N}$, we are interested in estimating how often P is satisfied. In many critical domains, client analyses often require guarantees that the computed estimates be reasonably close to the ground truth. We are not aware of any prior approaches that provide such formal guarantees, though the need for quantitative verification has recently been recognized [89].

Security Applications. Quantitative verification enables many applications in security analysis (and beyond) for neural networks. We present 3 applications in which the following analysis questions can be quantitatively answered:

- *Robustness*: How many adversarial samples does a given neural network have? Does one neural network have more adversarial inputs compared to another one?
- *Trojan Attacks*: A neural network can be trained to classify certain inputs with "trojan trigger" patterns to the desired label. How well-poisoned is a trojaned model, i.e., how many such trojan inputs does the attack successfully work for?
- *Fairness*: Does a neural network change its predictions significantly when certain input features are present (e.g., when the input record has gender attribute set to "female" vs. "male")?

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354245

*Part of the research done while working at National University of Singapore.
[1]The name stands for **N**eural **P**roperty **A**pproximate **Q**uantifier. Code and benchmarks are available at https://teobaluta.github.io/npaq/

Note that such analysis questions boil down to estimating how often a given property over inputs and outputs is satisfied. Estimating counts is fundamentally different from checking whether a satisfiable input exists. Since neural networks are stochastically trained, the mere existence of certain satisfiable inputs is not unexpected. The questions above checks whether their counts are sufficiently large to draw statistically significant inferences. Section 3 formulates these analysis questions as logical specifications.

Our Approach. The primary contribution of this paper is a new analysis framework, which models the given set of neural networks $\mathcal{N}$ and P as set of logical constraints, φ, such that the problem of quantifying how often $\mathcal{N}$ satisfies P reduces to model counting over φ. We then show that the quantitative verification is #P-hard. Given the computational intractability of #P, we seek to compute rigorous estimates and introduce the notion of *approximate quantitative verification*: given a prescribed tolerance factor ε and confidence parameter δ, we estimate how often P is satisfied with PAC-style guarantees, i.e., the computed result is within a multiplicative $(1+\varepsilon)$ factor of the ground truth with confidence at least $1-\delta$.

Our approach works by encoding the neural network into a logical formula in conjunctive normal form (CNF). The key to achieving soundness guarantees is our new notion of equicardinality, which defines a principled way of encoding neural networks into a CNF formula F, such that quantitative verification reduces to counting the satisfying assignments of F projected to a subset of the support of F. We then use approximate model counting on F, which has seen rapid advancement in practical tools that provide PAC-style guarantees on counts for F. The end result is a *quantitative verification procedure for neural networks with soundness and precision guarantees*.

While our framework is more general, we instantiate our analysis framework with a sub-class of neural networks called binarized neural networks (or BNNs) [46]. BNNs are multi-layered perceptrons with +/−1 weights and step activation functions. They have been demonstrated to achieve high accuracy for a wide variety of applications [51, 56, 70]. Due to their small memory footprint and fast inference time, they have been deployed in constrained environments such as embedded devices [51, 56]. We observe that specific existing encodings for BNNs adhere to our notion of equicardinality and implement these in a new tool called NPAQ. We provide proofs of key correctness and composability properties of our general approach, as well as of our specific encodings. Our encodings are linear in the size of $\mathcal{N}$ and P.

Empirical Results. We show that NPAQ scales to BNNs with $1-3$ internal layers and $20-200$ units per layer. We use 2 standard datasets namely MNIST and UCI Adult Census Income dataset. We encode a total of 84 models with $4,692-53,010$ parameters, into $1,056$ formulae and quantitatively verify them. NPAQ encodes properties in less than a minute and solves 97.1% formulae in a 24-hour timeout. Encodings scale linearly in the size of the models, and its running time is not dependent on the true counts. We showcase how NPAQ can be used in diverse security applications with case studies. First, we quantify the model robustness by measuring how many adversarially perturbed inputs are misclassified, and then the effectiveness of 2 defenses for model hardening with adversarial training. Next, we evaluate the effectiveness of trojan attacks outside the chosen test set. Lastly, we measure the influence of 3 sensitive features on the output and check if the model is biased towards a particular value of the sensitive feature.

Contributions. We make the following contributions:

- *New Notion.* We introduce the notion of *approximate quantitative verification* to estimate how often a property P is satisfied by the neural net N with theoretically rigorous PAC-style guarantees.
- *Algorithmic Approach, Tool, & Security Applications.* We propose a principled algorithmic approach for encoding neural networks to CNF formula that preserve model counts. We build an end-to-end tool called NPAQ that can handle BNNs. We demonstrate security applications of NPAQ in quantifying robustness, trojan attacks, and fairness.
- *Results.* We evaluate NPAQ on $1,056$ formulae derived from properties over BNNs trained on two datasets. We show that NPAQ presently scales to BNNs of over $50,000$ parameters, and evaluate its performance characteristics with respect to different user-chosen parameters.

2 PROBLEM DEFINITION

Definition 2.1. Let $\mathcal{N} = \{f_1, f_2, \ldots, f_m\}$ be a set of m neural nets, where each neural net f_i takes a vector of inputs $\mathbf{x_i}$ and outputs a vector $\mathbf{y_i}$, such that $\mathbf{y_i} = f_i(\mathbf{x_i})$. Let $\mathrm{P} : \{\mathbf{x} \cup \mathbf{y}\} \rightarrow \{0,1\}$ denote the property P over the inputs $\mathbf{x} = \bigcup_{i=1}^{m} \mathbf{x_i}$ and outputs $\mathbf{y} = \bigcup_{i=1}^{m} \mathbf{y_i}$. We define the specification of property P over $\mathcal{N}$ as $\varphi(\mathbf{x}, \mathbf{y}) = (\bigwedge_{i=1}^{m} (\mathbf{y_i} = f_i(\mathbf{x_i})) \wedge \mathrm{P}(\mathbf{x}, \mathbf{y}))$.

We show several motivating property specifications in Section 3. For the sake of illustration here, consider $\mathcal{N} = \{f_1, f_2\}$ be a set of two neural networks that take as input a vector of three integers and output a 0/1, i.e., $f_1 : \mathbb{Z}^3 \rightarrow \{0,1\}$ and $f_2 : \mathbb{Z}^3 \rightarrow \{0,1\}$. We want to encode a property to check the dis-similarity between f_1 and f_2, i.e., counting for how many inputs (over all possible inputs) f_1 and f_2 produce differing outputs. The specification is defined over the inputs $\mathbf{x} = [x_1, x_2, x_3]$, outputs $y_1 = f_1(\mathbf{x})$ and $y_2 = f_2(\mathbf{x})$ as $\varphi(\mathbf{x}, y_1, y_2) = (f_1(\mathbf{x}) = y_1 \wedge f_2(\mathbf{x}) = y_2 \wedge y_1 \neq y_2)$.

Given a specification φ for a property P over the set of neural nets $\mathcal{N}$, a verification procedure returns $r = 1$ (SAT) if there exists a satisfying assignment τ such that $\tau \models \varphi$, otherwise it returns $r = 0$ (UNSAT). A satisfying assignment for φ is defined as $\tau : \{\mathbf{x} \cup \mathbf{y}\} \rightarrow \{0,1\}$ such that φ evaluates to true, i.e., $\varphi(\tau) = 1$ or $\tau \models \varphi$.

While the problem of standard (qualitative) verification asks whether there exists a satisfying assignment to φ, the problem of quantitative verification asks how many satisfying assignments φ admits. We denote the set of satisfying assignments for the specification φ as $R(\varphi) = \{\tau : \tau \models \varphi\}$.

Definition 2.2. Given a specification φ for a property P over the set of neural nets $\mathcal{N}$, a quantitative verification procedure, NQV(φ), returns the number of satisfying assignments of φ, $r = |R(\varphi)|$.

It is worth noting that $|R(\varphi)|$ may be intractably large to compute via naïve enumeration. For instance, we consider neural networks with hundreds of bits as inputs for which the unconditioned input

space is $2^{|\mathbf{x}|}$. In fact, we prove that quantitative verification is #P-hard, as stated below.

Theorem 2.3. *NQV(φ) is #P-hard, where φ is a specification for a property P over binarized neural nets.*

Our proof is a parsimonious reduction of model counting of CNF formulas, #CNF, to quantitative verification of binarized neural networks. We show how an arbitrary CNF formula F can be transformed into a binarized neural net f_i and a property P such that for a specification φ for P over $\mathcal{N} = \{f_i\}$, it holds true that $R(\mathrm{F}) = R(\varphi)$. See Appendix A.2 for the full proof.

Remark 1. *The parsimonious reduction from #CNF to NQV implies that fully polynomial time randomized approximation schemes, including those based on Monte Carlo, cannot exist unless NP=RP.*

The computational intractability of #P necessitates a search for relaxations of NQV. To this end, we introduce the notion of an approximate quantitative verifier that outputs an approximate count within ϵ of the true count with a probability greater than $1 - \delta$.

Definition 2.4. Given a specification φ for a property P over a set of neural nets $\mathcal{N}$, $0 < \epsilon \leq 1$ and $0 < \delta \leq 1$, an approximate quantitative verification procedure (ϵ, δ)-NQV$(\varphi, \epsilon, \delta)$ computes r such that $Pr[(1+\epsilon)^{-1}|R(\varphi)| \leq r \leq (1+\epsilon)|R(\varphi)|] \geq 1 - \delta$.

The security analyst can set the "confidence" parameter δ and the precision or "error tolerance" ϵ as desired. The (ϵ, δ)-NQV definition specifies the end guarantee of producing estimates that are statistically sound with respect to chosen parameters (ϵ, δ).

Connection to Computing Probabilities. Readers can naturally interpret $|R(\varphi)|$ as a measure of probability. Consider $\mathcal{N}$ to be a set of functions defined over input random variables $\mathbf{x}$. The property specification φ defines an event that conditions inputs and outputs to certain values, which the user can specify as desired. The measure $|R(\varphi)|$ counts how often the event occurs under all possible values of $\mathbf{x}$. Therefore, $\frac{|R(\varphi)|}{2^{|\mathbf{x}|}}$ is the probability of the event defined by φ occurring. Our formulation presented here computes $|R(\varphi)|$ weighting all possible values of $\mathbf{x}$ equally, which implicitly assumes a uniform distribution over all random variables $\mathbf{x}$. Our framework can be extended to weighted counting [14, 15], assigning different user-defined weights to different values of $\mathbf{x}$, which is akin to specifying a desired probability distributions over $\mathbf{x}$. However, we consider this extension as a promising future work.

3 SECURITY APPLICATIONS

We present three concrete application contexts which highlight how quantitative verification is useful to diverse security analyses. The specific property specifications presented here derived directly from recent works, highlighting that NPAQ is broadly applicable to analysis problems actively being investigated.

Robustness. An adversarial example for a neural network is an input which under a small perturbation is classified differently [39, 81]. The lower the number of adversarial examples, the more "robust" the neural network. Early work on verifying robustness aimed at checking whether adversarial inputs exist. However, recent works suggest that adversarial inputs are statistically "not surprising" [7, 30, 85] as they are a consequence of normal error in statistical classification [20, 35, 36, 55]. This highlights the importance of analyzing whether a statistically significant number of adversarial examples exist, not just whether they exist at all, under desired input distributions. Our framework allows the analyst to specify a logical property of adversarial inputs and quantitatively verify it. Specifically, one can estimate how many inputs are misclassified by the net (f) and within some small perturbation distance k from a benign sample ($\mathbf{x_b}$) [13, 61, 62], by encoding the property P1 in our framework as:

$$P_1(\mathbf{x}, \mathbf{y}, \mathbf{x_b}, \mathbf{y_b}, k) = \sum_{j=1}^{|\mathbf{x}|} (\mathbf{x_b}[j] \oplus \mathbf{x}[j]) \leq k \wedge \mathbf{y_b} \neq \mathbf{y} \qquad \text{(P1)}$$

As a concrete usage scenario, our evaluation reports on BNNs for image classification (Section 6.2). Even for a small given input (say m bits), the space of all inputs within a perturbation of k bits is $\binom{m}{k}$, which is too large to check for misclassification one-by-one. NPAQ does not enumerate and yet can estimate adversarial input counts with PAC-style guarantees (Section 6.2). As we permit larger perturbation, as expected, the number of adversarial samples monotonically increase, and NPAQ can quantitatively measure how much. Further, we show how one can directly compare robustness estimates for two neural networks. Such estimates may also be used to measure the efficacy of defenses. Our evaluation on 2 adversarial training defenses shows that the hardened models show lesser robustness than the plain (unhardened) model. Such analysis can help to quantitatively refute, for instance, claims that BNNs are intrinsically more robust, as suggested in prior work [32].

Trojan Attacks. Neural networks, such as for facial recognition systems, can be trained in a way that they output a specific value, when the input has a certain "trojan trigger" embedded in it [54]. The trojan trigger can be a fixed input pattern (e.g., a sub-image) or some transformation that can be stamped on to a benign image. One of the primary goals of the trojan attack is to maximize the number of trojaned inputs which are classified as the desired target output, $\mathbf{l_{attack}}$. NPAQ can quantify the number of such inputs for a trojaned network, allowing attackers to optimize for this metric. To do so, one can encode the set of trojaned inputs as all those inputs $\mathbf{x}$ which satisfy the following constraint for a given neural network f, trigger $\mathbf{t}$, $\mathbf{l_{attack}}$ and the (pixel) location of the trigger M:

$$P_2(\mathbf{x}, \mathbf{y}, \mathbf{t}, \mathbf{l_{attack}}, M) = \bigwedge_{j \in M} (\mathbf{x}[j] = \mathbf{t}[j]) \wedge \mathbf{y} = \mathbf{l_{attack}} \qquad \text{(P2)}$$

Section 6.3 shows an evaluation on BNNs trained on the MNIST dataset. Our evaluation demonstrates that the attack accuracy on samples from the test set can differ significantly from the total set of trojaned inputs specified as in property P2.

Fairness. The right notion of algorithmic fairness is being widely debated [19, 24, 29, 41, 91]. Our framework can help quantitatively evaluate desirable metrics measuring "bias" for neural networks. Consider a scenario where a neural network f is used to predict the recommended salary for a new hire in a company. Having been trained on public data, one may want to check whether f makes biased predictions based on certain sensitive features such as race, gender, or marital status of the new hire. To verify this, one can count how often f proposes a higher salary for inputs when they have a particular sensitive feature (say "gender") set to

certain values (say "male"), given all other input features the same. Formally, this property can be encoded for given sensitive features denoted by set S along with two concrete sensitive values $\mathbf{s_1}, \mathbf{s_2}$, as:

$$P_3(\mathbf{x_1}, \mathbf{x_2}, \mathbf{y_1}, \mathbf{y_2}, S, \mathbf{s_1}, \mathbf{s_2}) = \bigwedge_{i \in S} (\mathbf{x_1}[i] = \mathbf{s_1}[i]) \bigwedge_{i \in S} (\mathbf{x_2[i]} = \mathbf{s_2[i]}) \bigwedge_{i \notin S} (\mathbf{x_1}[i] = \mathbf{x_2}[i]) \wedge \mathbf{y_1} = \mathbf{y_2} \tag{P3}$$

Notice the NPAQ counts over *all* possible inputs where the non-sensitive features remain equal, but only the sensitive features change, which causes no change in prediction. An unbiased model would produce a very high count, meaning that for most inputs (or with high probability), changing just the sensitive features results in no change in outputs. A follow-up query one may ask is whether inputs having different values for the sensitive features and all other values the same, determine an increases (or decreases) of the output salary prediction. This can be encoded as property P4 (or P5) below.

$$P_4(\mathbf{x_1}, \mathbf{x_2}, \mathbf{y_1}, \mathbf{y_2}, S, \mathbf{s_1}, \mathbf{s_2}) = \bigwedge_{i \in S} (\mathbf{x_1}[i] = \mathbf{s_1}[i]) \bigwedge_{i \in S} (\mathbf{x_2[i]} = \mathbf{s_2[i]}) \bigwedge_{i \notin S} (\mathbf{x_1}[i] = \mathbf{x_2}[i]) \wedge \mathbf{y_2} - \mathbf{y_1} > 0 \tag{P4}$$

$$P_5(\mathbf{x_1}, \mathbf{x_2}, \mathbf{y_1}, \mathbf{y_2}, S, \mathbf{s_1}, \mathbf{s_2}) = \bigwedge_{i \in S} (\mathbf{x_1}[i] = \mathbf{s_1}[i]) \bigwedge_{i \in S} (\mathbf{x_2[i]} = \mathbf{s_2[i]}) \bigwedge_{i \notin S} (\mathbf{x_1}[i] = \mathbf{x_2}[i]) \wedge \mathbf{y_2} - \mathbf{y_1} < 0 \tag{P5}$$

NPAQ can be used to quantitatively verify such properties, and compare models before deploying them based on such estimates. Section 6.4 presents more concrete evaluation details and interpretation of BNNs trained on the UCI Adult dataset [2].

4 APPROACH

Recall that exact counting (as defined in NQV) is $\#P$-hard. Even for approximate counting, many widely used sampling-based approaches, such as based on Monte Carlo methods [40, 42, 47, 60], do *not* provide soundness guarantees since existence of a method that only requires polynomially many samples computable in (randomized) polynomial time would imply $NP = RP$ (See Remark 1). For sound estimates, it is well-known that many properties encodable in our framework require intractably large number of samples—for instance, to check for distributional similarity of two networks f_1 and f_2 in the classical model, a lower bound of $O(\sqrt{2^{\mathbf{x}}})$ samples are needed to obtain estimates with reasonable (ϵ, δ) guarantees. However, approximate counting for boolean CNF formulae has recently become practical. These advances combine the classical ideas of universal hashing with the advances in the Boolean satisfiability by invoking SAT solvers for NP queries, i.e., to obtain satisfiable witnesses for queried CNF formulae. The basic idea behind these approximate CNF counters is to first employ universal hashing to randomly partition the set of solutions into *roughly small* buckets. Then, the approximate counter can enumerate a tractably small number of witnesses satisfying P using a SAT solver within one bucket, which calculates the "density" of satisfiable solutions in that bucket. By careful analysis using concentration bounds, these estimates can be extended to the sum over all buckets, yielding a provably sound PAC-style guarantee of estimates. Our work leverages this recent advance in approximate CNF counting to solve the problem of (ϵ, δ)-NQV [77].

The Equicardinality Framework. Our key technical advance is a new algorithmic framework for reducing (ϵ, δ)-NQV to CNF counting with an encoding procedure that has provable soundness. The procedure encodes $\mathcal{N}$ and P into φ, such that model counting in some way over φ counts over $\mathcal{N} \wedge \mathrm{P}$. This is *not* straight-forward. For illustration, consider the case of counting over boolean circuits, rather than neural networks. To avoid exponential blowup in the encoding, often one resorts to classical *equisatisfiable* encoding [84], which preserves satisfiability but introduces new variables in the process. Equisatisfiability means that the original formula is satisfiable if and only if the encoded one is too. Observe, however, that this notion of equisatisfiability is *not* sufficient for model counting—the encoded formula may be equisatisfiable but may have many more satisfiable solutions than the original.

We observe that a stronger notion, which we call *equicardinality*, provides a principled approach to constructing encodings that preserve counts. An equicardinality encoding, at a high level, ensures that the model count for an original formula can be computed by performing model counting *projected* over the subset of variables in the resulting formula. We define this equicardinality relation rigorously and prove in Lemma 4.2 that model counting over a constraint is *equivalent* to counting over its equicardinal encoding. Further, we prove in Lemma 4.3 that the equicardinality relation is *closed* under logical conjunction. This means model counting over conjunction of constraints is equivalent to counting over the conjunction of their equicardinal encodings. Equicardinality CNF encodings can thus be composed with boolean conjunction, while preserving equicardinality in the resulting formulae.

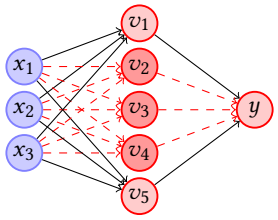
With this key observation, our procedure has two remaining sub-steps. First, we show equicardinal encodings for each neural net and properties over them to individual equicardinality CNF formulae. This implies ψ, the conjunction of the equicardinality CNF encodings of the conjuncts in φ, preserves the original model count of φ. Second, we show how an existing approximate model counter for CNF with (ϵ, δ) guarantees can be utilized to count over a projected subset of the variables in ψ. This end result, by construction, guarantees that our final estimate of the model count has bounded error, parameterized by ε, with confidence at least $1 - \delta$.

Formalization. We formalize the above notions using notation standard for boolean logic. The projection of an assignment σ over a subset of the variables $\mathbf{t}$, denoted as $\sigma|_{\mathbf{t}}$, is an assignment of $\mathbf{t}$ to the values taken in σ (ignoring variables other than $\mathbf{t}$ in σ).

Definition 4.1. We say that a formula $\varphi : \mathbf{t} \to \{0, 1\}$ is equicardinal to a formula $\psi : \mathbf{u} \to \{0, 1\}$ where $\mathbf{t} \subseteq \mathbf{u}$, if:

(a) $\forall \tau \models \varphi \Rightarrow \exists \sigma, (\sigma \models \psi) \wedge (\sigma|_{\mathbf{t}} = \tau)$, and
(b) $\forall \sigma \models \psi \Rightarrow \sigma|_{\mathbf{t}} \models \varphi$.

An example of a familiar equicardinal encoding is Tseitin [84], which transforms arbitrary boolean formulas to CNF. Our next lemma shows that equicardinality preserves model counts. We define $R(\psi) \downarrow \mathbf{t}$, the set of satisfying assignments of ψ projected over $\mathbf{t}$, as $\{\sigma|_{\mathbf{t}} : \sigma \models \psi\}$.



Cardinality Constraints:	x_1	x_2	x_3	$f(\mathbf{x})$
$x_1 + x_2 + x_3 \geq 2 \Leftrightarrow v_1 = 1$	0	0	0	0
$x_1 + \overline{x}_2 + x_3 \geq 1 \Leftrightarrow v_2 = 1$	0	0	1	0
$\overline{x}_1 + x_2 + x_3 \geq 1 \Leftrightarrow v_3 = 1$	0	1	0	0
$x_1 + x_2 + \overline{x}_3 \geq 1 \Leftrightarrow v_4 = 1$	0	1	1	1
$\overline{x}_1 + x_2 + \overline{x}_3 \geq 1 \Leftrightarrow v_5 = 1$	1	0	0	0
	1	0	1	1
$v_1 + v_2 + v_3 + v_4 + v_5 \geq 5 \Leftrightarrow y$	1	1	0	0
	1	1	1	1

Cardinality Constraints:	x_1	x_2	x_3	$f_1(\mathbf{x})$	$f_2(\mathbf{x})$
$\overline{x}_1 + \overline{x}_2 + x_3 \geq 1 \Leftrightarrow v_5$	0	0	0	0	0
$v_1 + v_2 + v_3 + v_4 + v_5 \geq 5$	0	0	1	0	0
$\Leftrightarrow y$	0	1	0	0	0
$f_1 : v_2 = v_3 = v_4 = 1$	0	1	1	1	0
$f_2 : x_1 + x_3 \geq 2 \Leftrightarrow v_1$	1	0	0	0	0
$x_1 + \overline{x}_2 + x_3 \geq 1 \Leftrightarrow v_2$	1	0	1	1	1
$\overline{x}_1 + x_2 + x_3 \geq 1 \Leftrightarrow v_3$	1	1	0	0	0
$x_1 + x_2 + \overline{x}_3 \geq 1 \Leftrightarrow v_4$	1	1	1	1	1

Figure 1. Example of encoding different BNNs (f, f_1, f_2) as a conjunction over a set of cardinality constraints. An attacker manipulates f with the goal to increase the inputs with trigger $x_3 = 1$ that classify as $y = 0$. Specifically, to obtain f_1 the weights of x_1, x_2, x_3 in constraints of f for v_2, v_3, v_4 are set to 0 (highlighted with dashed lines, on the left). To obtain f_2, we set $w_{21} = 0$. The trojan property $\mathbf{P} \doteq (y = 0) \wedge (x_3 = 1)$ is satisfied by one input (left) for f, whereas for f_2 we find two (right).

Lemma 4.2 (Count Preservation). *If ψ is equicardinal to φ, then $|R(\psi) \downarrow \mathbf{t}| = |R(\varphi)|$.*

Proof. By Definition 4.1(a), for every assignment $\tau \models \varphi$, there is a $\sigma \models \psi$ and the $\sigma|_{\mathbf{t}} = \tau$. Therefore, each distinct satisfying assignment of φ must have a unique assignment to $\sigma|_{\mathbf{t}}$, which must be in $R(\psi) \downarrow \mathbf{t}$. It follows that $|R(\psi) \downarrow \mathbf{t}| \geq |R(\varphi)|$, then. Next, observe that Definition 4.1(b) states that everything in $R(\psi) \downarrow \mathbf{t}$ has a satisfying assignment in φ; that is, its projection cannot correspond to a non-satisfying assignment in φ. By pigeonhole principle, it must be that $|R(\psi) \downarrow \mathbf{t}| \leq |R(\varphi)|$. This proves that $|R(\psi) \downarrow \mathbf{t}| = |R(\varphi)|$. □

Lemma 4.3 (CNF-Composibility). *Consider $\varphi_i : \mathbf{t_i} \rightarrow \{0,1\}$ and $\psi_i : \mathbf{u_i} \rightarrow \{0,1\}$, such that φ_i is equicardinal to ψ_i, for $i \in \{1,2\}$. If $\mathbf{u_1} \cap \mathbf{u_2} = \mathbf{t}$, where $\mathbf{t} = \mathbf{t_1} \cup \mathbf{t_2}$, then $\varphi_1 \wedge \varphi_2$ is equicardinal to $\psi_1 \wedge \psi_2$.*

Proof. (a) $\forall \tau \models \varphi_1 \wedge \varphi_2 \Rightarrow (\tau \models \varphi_1) \wedge (\tau \models \varphi_2)$. By Definition 4.1(a), $\exists \sigma_1, \sigma_2, \sigma_1 \models \psi_1 \wedge \sigma_2 \models \psi_2$. Further, by Definition 4.1(a), $\sigma_1|_{\mathbf{t}_1} = \tau|_{\mathbf{t}_1}$ and $\sigma_2|_{\mathbf{t}_2} = \tau|_{\mathbf{t}_2}$. This implies that $\sigma_1|_{\mathbf{t}_1 \cap \mathbf{t}_2} = \sigma_2|_{\mathbf{t}_1 \cap \mathbf{t}_2} = \tau|_{\mathbf{t}_1 \cap \mathbf{t}_2}$. We can now define the $\sigma_1 \otimes \sigma_2 = \sigma_1|_{\mathbf{u}_1 - \mathbf{t}_1} \cup \sigma_2|_{\mathbf{u}_2 - \mathbf{t}_2} \cup (\sigma_1|_{\mathbf{t}} \cap \sigma_2|_{\mathbf{t}})$. Since $(\mathbf{u_1} - \mathbf{t}) \cap (\mathbf{u_2} - \mathbf{t})$ is empty (the only shared variables between $\mathbf{u_1}$ and $\mathbf{u_2}$ are $\mathbf{t}$), it follows that $\sigma_1 \otimes \sigma_2 \models \psi_1 \wedge \psi_2$ and that $(\sigma_1 \otimes \sigma_2)|_{\mathbf{t}} = \tau$. This proves part (a) of the claim that $\varphi_1 \wedge \varphi_2$ is equicardinal to $\psi_1 \wedge \psi_2$.

(b) $\forall \sigma \models \psi_1 \wedge \psi_2 \Rightarrow (\sigma \models \psi_1) \wedge (\sigma \models \psi_2)$. By Definition 4.1(b), $\sigma|_{\mathbf{t}_1} \models \varphi_1$ and $\sigma|_{\mathbf{t}_2} \models \varphi_2$. This implies $\sigma|_{\mathbf{t}} \models \varphi_1 \wedge \varphi_2$, thereby proving the part (b) of the definition for the claim that $\varphi_1 \wedge \varphi_2$ is equicardinal to $\psi_1 \wedge \psi_2$.

□

Final Count Estimates. With the CNF-composability lemma at hand, we decompose the counting problem over a conjunction of neural networks $\mathcal{N}$ and property P, to that of counting over the conjunction of their respective equicardinality encodings. Equicardinality encodings preserve counts, and taking their conjunction preserves counts. It remains to show how to encode $\mathcal{N}$ to boolean CNF formulae, such that the encodings are equicardinal. Since the encoding preserves counts originally desired exactly, we can utilize off-the-shelf approximate CNF counters [16, 77] which have (ϵ, δ) guarantees. The final counts are thus guaranteed to be sound estimates by construction, which we establish formally in Theorem 5.5 for the encodings in Section 5.

Why Not Random Sampling? An alternative to our presented approach is random sampling. One could simply check what fraction of all possible inputs satisfies φ by testing on a random set of samples. However, the estimates produced by this method will satisfy soundness (defined in Section 2) *only if* the events being measured have sufficiently high probability. In particular, obtaining such soundness guarantees for rare events, i.e., where counts may be very low, requires an intractably large number of samples. Note that such events do arise in security applications [12, 89]. Specialized Monte Carlo samplers for such low probability events have been investigated in such contexts [89], but they do not provide soundness guarantees. We aim for a general framework, that works irrespective of the probability of events measured.

5 NPAQ DESIGN

Our tool takes as input a set of trained binarized neural networks $\mathcal{N}$ and a property P and outputs "how often" P holds over $\mathcal{N}$ with (ϵ, δ) guarantees. We show a two-step construction from binarized neural nets to CNF. The main principle we adhere to is that at every step we formally prove that we obtain equicardinal formulas. While BNNs and, in general, neural nets can be encoded using different background theories, we choose a specialized encoding of BNNs to CNF. First, we express a BNN using cardinality constraints similar to [58] (Section 5.1). For the second step, we choose to encode the cardinality constraints to CNF using a sorting-based encoding (Section 5.2). We prove that NPAQ is preserving the equicardinality in Theorem 5.5. Finally, we use an approximate model counter that can handle model counting directly over a projected subset of variables for a CNF formula [77].

5.1 BNN to Cardinality Constraints

Consider a standard BNN $f_i : \{-1,1\}^n \rightarrow \{0,1\}^s$ that consists of $d-1$ internal blocks and an output block [46]. We denote the kth internal block as f_{blk_k} and the output block as f_{out}. More formally, given an input $\mathbf{x} \in \{-1,1\}^n$, the binarized neural network is: $f_i(\mathbf{x}) = f_{out}(f_{\text{blk}_{d-1}}(\ldots(f_{\text{blk}_1}(\mathbf{x})\ldots))$. For every block f_{blk_k}, we define the inputs to f_{blk_k} as the vector $\mathbf{v_k}$. We denote the output for k block as the vector $\mathbf{v_{k+1}}$. For the output block, we use $\mathbf{v_d}$ to denote its input. The input to f_{blk_1} is $\mathbf{v_1} = \mathbf{x}$. We summarize the transformations for each block in Table 1.

Table 1. BNN definition as a set of layers of transformations.

A. Internal Block $f_{\mathtt{blk}_k}(\mathbf{v}_k) = \mathbf{v}_{k+1}$

1) Linear Layer

$$t_i^{lin} = \langle \mathbf{v}_k, \mathbf{w}_i \rangle + b_i \tag{1}$$

where $i = 1, ..., n_{k+1}$, $\mathbf{w}_i$ is the ith column in $\mathrm{W}_k \in \{-1, 1\}^{n_k \times n_{k+1}}$, $\mathbf{b}$ is the bias row vector $\in \mathbb{R}^{n_{k+1}}$ and $\mathbf{y} \in \mathbb{R}^{n_{k+1}}$

2) Batch Normalization

$$t_i^{bn} = \frac{t_i^{lin} - \mu_{k_i}}{\sigma_{k_i}} \cdot \alpha_{k_i} + \gamma_{k_i} \tag{2}$$

where $i = 1, ..., n_{k+1}$, α_k is the kth weight row vector $\in \mathbb{R}^{n_{k+1}}$, γ_k is the bias $\in \mathbb{R}^{n_{k+1}}$, $\mu_k \in \mathbb{R}^{n_{k+1}}$ is the mean and $\sigma_k \in \mathbb{R}^{n_{k+1}}$ is the standard deviation.

3) Binarization

$$t_i^{bn} \geq 0 \Rightarrow v_{k+1_i} = 1 \tag{3}$$

$$t_i^{bn} < 0 \Rightarrow v_{k+1_i} = -1 \tag{4}$$

where $i = 1, ..., n_{k+1}$.

B. Output Block $f_{out}(\mathbf{v}_d) = \mathbf{y}$

1) Linear Layer

$$q_i^{lin} = \langle \mathbf{v}_d, \mathbf{w}_j \rangle + b_i \tag{5}$$

where $\mathbf{v}_d \in \{-1, 1\}^{n_d}$, $\mathbf{w}_j$ is the jth column $\in \mathbb{R}^{n_d \times s}$, $\mathbf{b} \in \mathbb{R}^s$ is the bias vector.

2) Argmax

$$y_i = 1 \Leftrightarrow i = \arg\max(\mathbf{q}^{lin}) \tag{6}$$

Running Example. Consider a binarized neural net $f : \{-1, 1\}^3 \rightarrow \{0, 1\}$ with a single internal block and a single output (Figure 1). To show how one can derive the constraints from the BNN's parameters, we work through the procedure to derive the constraint for the output of the internal block's first neuron, denoted by v_1. Suppose we have the following parameters: the weight column vector $\mathbf{w}_1 = [1\ 1\ 1]$ and bias $b_1 = -2.0$ for the linear layer; $\alpha_1 = 0.8, \sigma_1 = 1.0, \gamma_1 = 2.0, \mu_1 = -0.37$ parameters for the batch normalization layer. First, we apply the linear layer transformation (Eq. 1 in Table 1). We create a temporary variable for this intermediate output, $t_1^{lin} = \langle \mathbf{x}, \mathbf{w}_1 \rangle + b_1 = x_1 + x_2 + x_3 - 2.0$. Second, we apply the batch normalization (Eq. 2 in Table 1) and obtain $t_1^{bn} = (x_1 + x_2 + x_3 - 2.0 + 0.37) \cdot 0.8 + 2.0$. After the binarization (Eq. 3 in Table 1), we obtain the constraints $\mathrm{S}_1 = ((x_1 + x_2 + x_3 - 2.0 + 0.37) \cdot 0.8 + 2.0 \geq 0)$ and $\mathrm{S}_1 \Leftrightarrow v_1 = 1$. Next, we move all the constants to the right side of the inequality: $x_1 + x_2 + x_3 \geq -2.0/0.8 + 2.0 - 0.37 \Leftrightarrow v_1 = 1$. Lastly, we translate the input from the $\{-1, 1\}$ domain to the boolean domain, $x_i = 2x_i^{(b)} - 1, i \in \{1, 2, 3\}$, resulting in the following constraint: $2(x_1^{(b)} + x_2^{(b)} + x_3^{(b)}) - 3 \geq -0.87$. We use a sound approximation for the constant on the right side to get rid of the real values and obtain $x_1^{(b)} + x_2^{(b)} + x_3^{(b)} \geq \lceil 1.065 \rceil = 2$. For notational simplicity the variables x_1, x_2, x_3 in Figure 1 are boolean variables (since $x = 1 \Leftrightarrow x^{(b)} = 1$).

To place this in the context of the security application in Section 3, we examine the effect of two arbitrary trojan attack procedures. Their aim is to manipulate the output of a given neural network, f, to a target class for inputs with a particular trigger. Let us consider the trigger to be $x_3 = 1$ and the target class $y = 0$ for two trojaned neural nets, f_1 and f_2 (shown in Figure 1). Initially, f outputs class 0 for only one input that has the trigger $x_3 = 1$. The first observation is that f_1 is equivalent to f, even though its parameters have changed. The second observation is that f_2 changes its output prediction for the input $x_1 = 0, x_2 = 1, x_3 = 1$ to the target class 0. We want NPAQ to find how much do f_1 and f_2 change their predictions for the target class with respect to the inputs that have the trigger, i.e., $|R(\varphi_1)| < |R(\varphi_2)|$, where φ_1, φ_2 are trojan property specifications (property P_2 as outlined Section 3).

Encoding Details. The details of our encoding in Table 2 are similar to [58]. We first encode each block to mixed integer linear programming and implication constraints, applying the $\mathtt{MILP}_{\mathtt{blk}}$ rule for the internal block and $\mathtt{MILP}_{\mathtt{out}}$ for the outer block (Table 2). To get rid of the reals, we use sound approximations to bring the constraints down to integer linear programming constraints (see $\mathtt{ILP}_{\mathtt{blk}}$ and $\mathtt{ILP}_{\mathtt{out}}$ in Table 2). For the last step, we define a 1:1 mapping between variables in the binary domain $x \in \{-1, 1\}$ and variables in the boolean domain $x^{(b)} \in \{0, 1\}$, $x^{(b)} = 2x - 1$. Equivalently, for $x \in \{-1, 1\}$ there exists a unique $x^{(b)}$: $(x^{(b)} \Leftrightarrow x = 1) \wedge (\overline{x}^{(b)} \Leftrightarrow x = -1)$. Thus, for every block $f_{\mathtt{blk}_k}(\mathbf{v}_k) = \mathbf{v}_{k+1}$, we obtain a corresponding formula over booleans denoted as $\mathrm{BLK}_k(\mathbf{v}_k^{(b)}, \mathbf{v}_{k+1}^{(b)})$, as shown in rule $\mathtt{Card}_{\mathtt{blk}}$ (Table 2). Similarly, for the output block f_{out} we obtain $\mathrm{OUT}(\mathbf{v}_d, \mathbf{ord}, \mathbf{y})$. We obtain the representation of $\mathbf{y} = f_i(\mathbf{x})$ as a formula BNN shown in Table 2. For notational simplicity, we denote the introduced intermediate variables $\mathbf{v}_k^{(b)} = [v_{k_1}^{(b)}, \ldots, v_{k_{n_k}}^{(b)}], k = 2, \ldots, d$ and $\mathbf{ord} = [ord_i, \ldots, ord_{n_d \cdot n_d}]$ as $\mathbf{a}_V$. Since there is a 1:1 mapping between $\mathbf{x}$ and $\mathbf{x}^{(b)}$ we use the notation $\mathbf{x}$, when it is clear from context which domain $\mathbf{x}$ has. We refer to BNN as the formula $\mathrm{BNN}(\mathbf{x}, \mathbf{y}, \mathbf{a}_V)$.

Lemma 5.1. *Given a binarized neural net $f_i : \{-1, 1\}^n \rightarrow \{0, 1\}^s$ over inputs $\mathbf{x}$ and outputs $\mathbf{y}$, and a property P, let φ be the specification for P, $\varphi(\mathbf{x}, \mathbf{y}) = (\mathbf{y} = f_i(\mathbf{x})) \wedge P(\mathbf{x}, \mathbf{y})$, where we represent $\mathbf{y} = f_i(\mathbf{x})$ as BNN$(\mathbf{x}, \mathbf{y}, \mathbf{a}_V)$. Then φ is equicardinal to BNN$(\mathbf{x}, \mathbf{y}, \mathbf{a}_V)$.*

Proof. We observe that the intermediate variables for each block in the neural network, namely $\mathbf{v}_k$ for the kth block, are introduced by double implication constraints. Hence, not only are both part (a) and part (b) of definition 4.1 true, but the satisfying assignments for the intermediate variables $\mathbf{a}_V$ are uniquely determined by $\mathbf{x}$. Due to space constraints, we give our full proof in Appendix A.1. □

5.2 Cardinality Constraints to CNF

Observe that we can express each block in BNN as a conjunction of cardinality constraints [4, 6, 76]. Cardinality constraints are constraints over boolean variables $x_1, \ldots, x_n$ of the form $x_1 + \ldots + x_n \triangle c$, where $\triangle \in \{=, \leq, \geq\}$. More specifically, by applying the $\mathtt{Card}_{\mathtt{blk}}$ rule (Table 2), we obtain a conjunction over cardinality constraints S_{k_i}, together with a double implication: $\mathrm{BLK}_k(\mathbf{v}_k^{(b)}, \mathbf{v}_{k+1}^{(b)}) = \bigwedge_{i=1}^{n_{k+1}} \mathrm{S}_{k_i}(\mathbf{v}_k^{(b)}) \Leftrightarrow v_{k+1_i}^{(b)}$. We obtain a similar conjunction of cardinality constraints for the output block ($\mathtt{Card}_{\mathtt{out}}$, Table 2). The last step for obtaining a Boolean formula representation for the BNN is encoding the cardinality constraints to CNF.

Table 2. Encoding for a binarized neural network BNN(x) to cardinality constraints, where $\mathbf{v}_1 = \mathbf{x}$. MILP stands for Mixed Integer Linear Programming, ILP stands for Integer Linear Programming.

A. $f_{\mathtt{blk}_k}(\mathbf{v}_k, \mathbf{v}_{k+1})$ to $\mathtt{BLK}_k(\mathbf{v}_k^{(b)}, \mathbf{v}_{k+1}^{(b)})$

$$\mathtt{MILP_{blk}}: \frac{\text{Eq (1), Eq (2), Eq (3)}, \alpha_{k_i} > 0}{\langle \mathbf{v}_k, \mathbf{w}_i \rangle \geq -\frac{\sigma_{k_i}}{\alpha_{k_i}} \cdot \gamma_{k_i} + \mu_{k_i} - b_i, i = 1, ..., n_{k+1}} \qquad \mathtt{ILP_{blk}}: \frac{\alpha_{k_i} > 0}{\begin{array}{c}\langle \mathbf{v}_k, \mathbf{w}_i \rangle \geq C_i \Leftrightarrow v_{k+1_i} = 1, i = 1, ..., n_{k+1} \\ \langle \mathbf{v}_k, \mathbf{w}_i \rangle < C_i \Leftrightarrow v_{k+1_i} = -1, i = 1, ..., n_{k+1} \\ C_i = \lceil -\frac{\sigma_{k_i}}{\alpha_{k_i}} \cdot \gamma_{k_i} + \mu_{k_i} - b_i \rceil\end{array}}$$

$$\mathtt{Card_{blk}}: \frac{v^{(b)} = 2v - 1, v \in \{-1, 1\}}{\mathrm{BLK}_k(\mathbf{v}_k^{(b)}, \mathbf{v}_{k+1}^{(b)}) = \sum_{j \in w_{k_i}^+} v_{k_j}^{(b)} + \sum_{j \in w_{k_i}^-} \overline{v}_{k_j}{}^{(b)} \geq C_i' + |w_{k_i}^-| \Leftrightarrow v_{k+1_i}^{(b)} = 1, C_i' = \lceil (C_i + \sum_{j=1}^{n_k} w_{ji})/2 \rceil}$$

B. $f_{out}(\mathbf{v}_d, \mathbf{y})$ to $\mathtt{OUT}(\mathbf{v}_d^{(b)}, \mathbf{ord}, \mathbf{y})$

$$\mathtt{Order}: \frac{ord_{ij} \in \{0, 1\}}{q_i^{lin} \geq q_j^{lin} \Leftrightarrow ord_{ij} = 1} \quad \mathtt{MILP_{out}}: \frac{\text{Eq (5), Eq (Order)}}{\langle \mathbf{v}_d, \mathbf{w}_i - \mathbf{w}_j \rangle \geq b_j - b_i \Leftrightarrow ord_{ij} = 1} \quad \mathtt{ILP_{out}}: \langle \mathbf{v}_d, \mathbf{w}_i - \mathbf{w}_j \rangle \geq \lceil b_j - b_i \rceil \Leftrightarrow ord_{ij} = 1$$

$$\mathtt{Card_{out}}: \frac{v^{(b)} = 2v - 1, v \in \{-1, 1\}}{\begin{array}{c}\mathrm{OUT}(\mathbf{v}_d^{(b)}, \mathbf{ord}, \mathbf{y}) = \Big(\big(\sum_{p \in w_i^+ \cap w_j^-} v_{d_p}^{(b)} - \sum_{p \in w_i^- \cap w_j^+} v_{d_p}^{(b)} \geq \lceil E_{ij}/2 \rceil \big) \Leftrightarrow ord_{ij} \wedge \sum_{i=1}^{s} ord_{ij} = s \Leftrightarrow y_i = 1 \Big), \\ E_{ij} = \lceil (b_j - b_i + \sum_{p=1}^{n_d} w_{ip} - \sum_{p=1}^{n_d} w_{jp})/2 \rceil\end{array}}$$

C. f_i to BNN

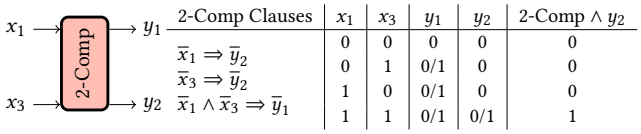
$$\mathrm{BNN}(\mathbf{x}^{(b)}, \mathbf{y}, \mathbf{v}_2^{(b)}, \ldots, \mathbf{v}_d^{(b)}, \mathbf{ord}) = \mathrm{BLK}_1(\mathbf{x}^{(b)}, \mathbf{v_2}^{(b)}) \bigwedge_{k=2}^{d-1} \Big(\mathrm{BLK}_k(\mathbf{v}_k^{(b)}, \mathbf{v}_{k+1}^{(b)}) \Big) \wedge \mathrm{OUT}(\mathbf{v}_d^{(b)}, \mathbf{y}, \mathbf{ord})$$

We choose cardinality networks [4, 6] to encode the cardinality constraints to CNF formulas and show for this particular encoding that the resulting CNF is equicardinal to the cardinality constraint. Cardinality networks implement several types of gates, i.e., merge circuits, sorting circuits and 2-comparators, that compose to implement a merge sort algorithm. More specifically, a cardinality constraint of the form $\mathrm{S}(\mathbf{x}) = x_1 + \ldots + x_n \geq c$ has a corresponding cardinality network, $\mathrm{Card}_c = \Big((\mathrm{Sort}_c(x_1, \ldots, x_n) = (y_1, \ldots, y_c)) \wedge y_c \Big)$, where Sort is a sorting circuit. As shown by [4, 6], the following holds true:

Proposition 5.2. *A Sort_c network with an input of n variables, outputs the first c sorted bits. $\mathrm{Sort}_c(x_1, \ldots, x_n) = (y_1, \ldots, y_c)$ where $y_1 \geq y_2 \geq \ldots \geq y_c$.*

We view Card_c as a circuit where we introduce additional variables to represent the output of each gate, and the output of Card_c is 1 only if the formula S is true. This is similar to how a Tseitin transformation [84] encodes a propositional formula into CNF.

Running Example. Revisiting our example in Section 5.1, consider f_2's cardinality constraint corresponding to v_1, denoted as $\mathrm{S}_1' = x_1 + x_3 \geq 2$. This constraint translates to the most basic gate of cardinality networks, namely a 2-comparator [6, 9] shown in Figure 2. Observe that while this efficient encoding ensures that S_1 is equi-satisfiable to the formula 2-Comp $\wedge\, y_2$, counting over the CNF formula does not preserve the count, i.e., it over-counts due to variable y_1. Observe, however, that this encoding is equicardinality



2-Comp Clauses	x_1	x_3	y_1	y_2	2-Comp $\wedge\, y_2$
	0	0	0	0	0
$\overline{x}_1 \Rightarrow \overline{y}_2$	0	1	0/1	0	0
$\overline{x}_3 \Rightarrow \overline{y}_2$	1	0	0/1	0	0
$\overline{x}_1 \wedge \overline{x}_3 \Rightarrow \overline{y}_1$	1	1	0/1	0/1	1

Figure 2. Cardinality networks encoding for $x_1 + x_3 \geq 2$. For this case, cardinality networks amount to a 2-comparator gate. Observe there are two satisfying assignments for 2-Comp $\wedge\, y_2$ due to the "don't care" assignment to y_1.

and thus, a projected model count on $\{x_1, x_3\}$ gives the correct model count of 1. The remaining constraints shown in Figure 1 are encoded similarly and not shown here for brevity.

Lemma 5.3 (Substitution). *Let F be a Boolean formula defined over the variables Vars and $p \in Vars$. For all satisfying assignments $\tau \models F \Rightarrow \tau|_{Vars - \{p\}} \models F[p \mapsto \tau[p]]$.*

Lemma 5.4. *For a given cardinality constraint, $S(\mathbf{x}) = x_1 + \ldots + x_n \geq c$, let $Card_c$ be the CNF formula obtained using cardinality networks, $Card_c(\mathbf{x}, \mathbf{a}_C) := (Sort_c(x_1, \ldots, x_n) = (y_1, \ldots, y_c) \wedge y_c)$, where $\mathbf{a}_C$ are the auxiliary variables introduced by the encoding. Then, $Card_c$ is equicardinal to S.*

(a) $\forall \tau \models S \Rightarrow \exists \sigma, \sigma \models Card_c \wedge \sigma|_{\mathbf{x}} = \tau$.
(b) $\forall \sigma \models Card_c \Rightarrow \tau_3|_{\mathbf{x}} \models S$.

Proof. (a) Let $\tau \models S \implies$ there are least c x_i's such that $\tau[x_i] = 1, i \geq c$. Thus, under the valuation τ_1 to the input variables $x_1, \ldots, x_n$, the sorting network outputs a sequence $y_1, \ldots, y_c$ where $y_c = 1$, where $y_1 \geq \ldots \geq y_c$ (Proposition 5.2). Therefore, $\text{Card}_c[\mathbf{x} \mapsto \tau] = (\text{Sort}_c(x_1 \mapsto \tau[x_1], \ldots, x_n \mapsto \tau[x_n]) = (y_1, \ldots, y_c) \wedge y_c)$ is satisfiable. This implies that $\exists \sigma, \sigma \models \text{Card}_c \wedge \sigma|_{\mathbf{x}} = \tau$.

(b) Let $\sigma \models \text{Card}_c \implies \sigma[y_c] = 1$. By Lemma 5.3, $\sigma|_{\mathbf{x}} \models \text{Card}_c[y_i \mapsto \sigma[y_i]], \forall y_i \in \mathbf{a}_C$. From Proposition 5.2, under the valuation σ, there are at least c x_i's such that $\sigma[x_i] = 1, i \geq c$. Therefore, $\sigma|_{\mathbf{x}} \models S$.

□

For every S_{k_i}, $k = 1, \ldots, d, i = 1, \ldots, n_{k+1}$, we have a CNF formula C_{k_i}. The final CNF formula for $\text{BNN}(\mathbf{x}, \mathbf{y}, \mathbf{a}_V)$ is denoted as $C(\mathbf{x}, \mathbf{y}, \mathbf{a})$, where $\mathbf{a} = \mathbf{a}_V \bigcup_{k=1}^{d} \bigcup_{i=1}^{n_{k+1}} \mathbf{a}_C^{k_i}$ and $\mathbf{a}_C^{k_i}$ is the set of variables introduced by encoding S_{k_i}.

Encoding Size. The total CNF formula size is linear in the size of the model. Given one cardinality constraint $S(\mathbf{v_k})$, where $|\mathbf{v_k}| = n$, a cardinality network encoding produces a CNF formula with $O(n\ log^2\ c)$ clauses and variables. The constant c is the maximum value that the parameters of the BNN can take, hence the encoding is linear in n. For a given layer with m neurons, this translates to m cardinality constraints, each over n variables. Hence, our encoding procedure produces $O(m \times n)$ clauses and variables for each layer. For the output block, s is the number of output classes and n is the number of neurons in the previous layer. Due to the ordering relation encoding the arg max, there are $O(s \times s \times n)$ clauses and variables for the output block. Therefore, the total size for a BNN with l layers of the CNF is $O(m \times n \times l + s \times s \times n)$, which is linear in the size of the original model.

Alternative Encodings. Besides cardinality networks, there are many other encodings from cardinality constraints to CNF [3, 4, 6, 25, 76] that can be used as long as they are equicardinal. We do not formally prove here but we strongly suspect that adder networks [25] and BDDs [3] have this property. Adder networks [25] provide a compact, linear transformation resulting in a CNF with $O(n)$ variables and clauses. The idea is to use adders for numbers represented in binary to compute the number of activated inputs and a comparator to compare it to the constant c. A BDD-based [25] encoding builds a BDD representation of the constraint. It uses $O(n^2)$ clauses and variables. For approximate counting techniques, empirically, these similar encodings yield similar performance [67].

5.3 Projected Model Counting

We instantiate the property P encoded in CNF and the neural network encoded in a CNF formulae C. We make the observation that we can directly count the number of satisfying assignment for φ over a subset of variables, known as projected model counting [14]. NPAQ uses an approximate model counter with strong PAC-style guarantees. ApproxMC3 [77] is an approximate model counter that can directly count on a projected formula making a logarithmic number of calls in the number of formula variables to an NP-oracle, namely a SAT solver.

Theorem 5.5. NPAQ *is an* (ϵ, δ)*-NQV.*

Proof. First, by Lemma 4.3, since each cardinality constraint S_{k_i} is equicardinal to C_{k_i} (Lemma 5.4), the conjunction over the cardinality constraints is also equicardinal. Second, by Lemma 5.1, BNN is equicardinal to C. Since we use an approximate model counter with (ϵ, δ) guarantees [77], NPAQ returns r for a given BNN and a specification φ with (ϵ, δ) guarantees. □

6 IMPLEMENTATION & EVALUATION

We aim to answer the following research questions:
(RQ1) To what extent does NPAQ scale to, e.g., how large are the neural nets and the formulae that NPAQ can handle?
(RQ2) How effective is NPAQ at providing sound estimates for practical security applications?
(RQ3) Which factors influence the performance of NPAQ on our benchmarks and how much?
(RQ4) Can NPAQ be used to refute claims about security-relevant properties over BNNs?

Implementation. We implemented NPAQ in about 5,000 LOC of Python and C++. We use the PyTorch (v1.0.1.post2) [64] deep learning platform to train and test binarized neural networks. For encoding the BNNs to CNF, we build our own tool using the PBLib library [66] for encoding the cardinality constraints to CNF. The resulting CNF formula is annotated with a projection set and NPAQ invokes the approximate model counter ApproxMC3 [77] to count the number of solutions. We configure a tolerable error $\epsilon = 0.8$ and confidence parameter $\delta = 0.2$ as defaults throughout the evaluation.

Models. Our benchmarks consist of BNNs, on which we tested the properties derived from the 3 applications outlined in Section 3. The utility of NPAQ in these security applications is discussed in Sections 6.2- 6.4. For each application, we trained BNNs with the following 4 different architectures:

- $\textbf{ARCH}_1$ - $\text{BLK}_1(100)$
- $\textbf{ARCH}_2$ - $\text{BLK}_1(50)$, $\text{BLK}_2(20)$
- $\textbf{ARCH}_3$ - $\text{BLK}_1(100)$, $\text{BLK}_2(50)$
- $\textbf{ARCH}_4$ - $\text{BLK}_1(200)$, $\text{BLK}_2(100)$, $\text{BLK}_3(100)$

For each architecture, we take snapshots of the model learnt at different epochs. In total, this results in 84 total models with $6,560 - 53,010$ parameters for models trained with the MNIST dataset and $4,692 - 45,402$ parameters for models trained with the UCI Adult dataset. Encoding various properties (Sections 6.2- 6.4) results in a total of 1,056 distinct formulae. For each formula, NPAQ returns r i.e., the number of satisfying solutions. Given r, we calculate PS i.e., the percentage of the satisfying solutions with respect to the total input space size. The meaning of PS percentage values is application-specific. In trojan attacks, `PS(tr)` represents inputs labeled as the target class. In robustness quantification, `PS(adv)` reports the adversarial samples.

Datasets. We train models over 2 standard datasets. Specifically, we quantify robustness and trojan attack effectiveness on the MNIST [52] dataset and estimate fairness queries on the UCI Adult dataset [2]. We choose them as prior work use these datasets [5, 32, 33, 69].

MNIST. The dataset contains 60,000 gray-scale 28×28 images of handwritten digits with 10 classes. In our evaluation, we resize the images to 10×10 and binarize the normalized pixels in the images.

Table 3. Influence of (ϵ, δ) on NPAQ's Performance. The count and time taken to compute the bias in $ARCH_2$ trained on UCI Adult dataset for changes in values features (marital status, gender, and race) i.e., the percentage of individuals whose predicted income changes from $\leq$ 50K to $>$ 50K when all the other features are same. NLC represents the natural logarithm of the count NPAQ generates. Time represents the number of hours NPAQ takes to solve the formulae. x represents a timeout.

Feature	$\delta = 0.2$								$\epsilon = 0.1$							
	$\epsilon = 0.1$		$\epsilon = 0.3$		$\epsilon = 0.5$		$\epsilon = 0.8$		$\delta = 0.01$		$\delta = 0.05$		$\delta = 0.1$		$\delta = 0.2$	
	NLC	Time	NLC	Time	NLC	Time	NLC	Time	NLC	Time	NLC	Time	NLC	Time	NLC	Time
Marital Status	39.10	8.79	39.08	1.35	39.09	0.80	39.13	0.34	x	x	39.07	22.48	39.07	15.74	39.10	8.79
Race	40.68	3.10	40.64	0.68	40.65	0.42	40.73	0.27	40.68	14.68	40.67	8.21	40.67	5.80	40.68	3.10
Gender	41.82	3.23	41.81	0.62	41.88	0.40	41.91	0.27	41.81	15.48	41.81	8.22	41.81	6.02	41.82	3.23

UCI Adult Census Income. The dataset is 48,842 records with 14 attributes such as age, gender, education, marital status, occupation, working hours, and native country. The task is to predict whether a given individual has an income of over \$50,000 a year. 5/14 attributes are numerical variables, while the remaining attributes are categorical variables. To obtain binary features, we divide the values of each numerical variables into groups based on its deviation. Then, we encode each feature with the least amount of bits that are sufficient to represent each category in the feature. For example, we encode the race feature which has 5 categories in total with 3 bits, leading to 3 redundant values in this feature. We remove the redundant values by encoding the property to disable the usage of these values in NPAQ. We consider 66 binary features in total.

Experimental Setup. All experiments are performed on 2.5 GHz CPUs, 56 cores, 64GB RAM. Each counting process executed on one core and 4GB memory cap and a 24-hour timeout per formula.

6.1 NPAQ Benchmarking

We benchmark NPAQ and report breakdown on 1,056 formulae.

Estimation Efficiency. NPAQ successfully solves 97.1% (1,025 / 1,056) formulae. In quantifying the effectiveness of trojan attacks and fairness applications, the raw size of the input space (over all possible choices of the free variables) is 2^{96} and 2^{66}, respectively. Naive enumeration for such large spaces is intractable. NPAQ returns estimates for 83.3% of the formulae within 12 hours and 94.8% of the formulae within 24 hours for these two applications. In robustness application, the total input sizes are a maximum of about 7.5×10^7.

Result 1: NPAQ solves 97.1% formulae in 24-hour timeout.

Encoding Efficiency. NPAQ takes a maximum of 1 minute to encode each model, which is less than 0.05% of the total timeout. The formulae size scale linearly with the model, as expected from encoding construction. NPAQ presently utilizes off-the-shelf CNF counters, and their performance heavily dominates NPAQ time. NPAQ presently scales to formulae of ~3.5×10^6 variables and ~6.2×10^6 clauses. However, given the encoding efficiency, we expect NPAQ to scale to larger models with future CNF counters [18, 77].

Result 2: NPAQ takes ~1 minute to encode the model.

Number of Formulae vs. Time. Figure 3 plots the number of formulae solved with respect to the time, the relationship is logarithmic. NPAQ solves 93.2% formulae in the first 12 hours, whereas, it only solves 3.9% more in the next 12 hours. We notice that the neural net depth impacts the performance, most timeouts (27/31) stem from $ARCH_4$. 26/31 timeouts are for Property P1 (Section 3) to quantify adversarial robustness. Investigating why certain formulae are harder to count is an active area of independent research [21, 22].

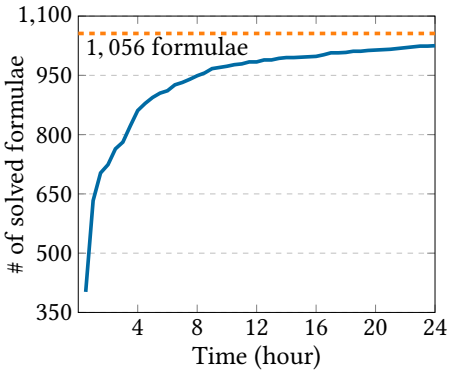


Figure 3. Number of formulae NPAQ solves with respect to the time. The solid line represents the aggregate number of formulae NPAQ solves before the given time. The dashed line represents the total number of formulae.

Performance with Varying (ϵ, δ). We investigate the relationship between different error and confidence parameters and test co-relation with parameters that users can pick. We select a subset of formulae [2] which have varying degrees of the number of solutions, a large enough input space which is intractable for enumeration, and varying time performance for the baseline parameters of $\epsilon = 0.8, \delta = 0.2$. The formulae in our dataset that satisfy these requirements arise in the fairness application. More specifically, we chose the 3 formulae encoding the fairness properties over $ARCH_2$ where the input space is 2^{66} and the PS varies from 4.09 to 76.59.

We first vary the error tolerance (or precision), $\epsilon \in \{0.1, 0.3, 0.5, 0.8\}$ while keeping the same $\delta = 0.2$ for the fairness application, as shown in Table 3. This table illustrates no significant resulting difference in counts reported by NPAQ under different precision parameter values. More precisely, the largest difference as the natural logarithmic of the count is 0.1 for $\epsilon = 0.3$ and $\epsilon = 0.8$ for the feature "Gender". This suggests that for these formulae, decreasing the error bound does not yield a much higher count precision.

[2] Our timeout is 24 hours per formula, so we resorted to checking a subset of formulae.

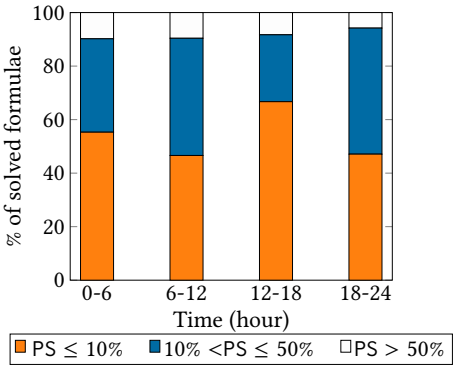


Figure 4. Percentage of formulae NPAQ solves with respect to 3 PS intervals (less than 10% in orange, between 10% and 50% in blue and more than 50% in white) and 4 time intervals.

Higher precision does come at a higher performance cost, as the $\epsilon = 0.1$ takes 16× more time than $\epsilon = 0.8$. The results are similar when varying the confidence parameter $\delta \in \{0.2, 0.1, 0.05, 0.01\}$ (smaller is better) for $\epsilon = 0.1$ (Table 3). This is because the number of calls to the SAT solver depends only on the δ parameter, while ϵ dictates how constrained the space of all inputs or how small the "bucket" of solutions is [16, 77]. Both of these significantly increase the time taken. Users can tune ϵ and δ based on the required applications precision and the available time budget.

Result 3: NPAQ reports no significant difference in the counts produced when configured with different ϵ and δ.

PS vs. Time. We investigate if NPAQ solving efficiency varies with increasing count size. Specifically, we measure the PS with respect to the time taken for all the 1,056 formulae. Figure 4 shows the PS plot for 4 time intervals and 3 density intervals. We observe that the number of satisfying solutions do not significantly influence the time taken to solve the instance. This suggests that NPAQ is generic enough to solve formulae with arbitrary solution set sizes.

Result 4: For a given ϵ and δ, NPAQ solving time is not significantly influenced by the PS.

6.2 Case Study 1: Quantifying Robustness

We quantify the model robustness and the effectiveness of defenses for model hardening with adversarial training.

Number of Adversarial Inputs. One can count precisely what fraction of inputs, when drawn uniformly at random from a constrained input space, are misclassified for a given model. For demonstrating this, we first train 4 BNNs on the MNIST dataset, one using each of the architectures ARCH_1-ARCH_4. We encode the Property P1 (Section 3) corresponding to perturbation bound $k \in \{2, 3, 4, 5\}$. We take 30 randomly sampled images from the test set, and for each one, we encoded one property constraining adversarial perturbation to each possible value of k. This results in a total of 480 formulae on which NPAQ runs with a timeout of 24 hours per formula. If NPAQ terminates within the timeout limit, it either quantifies the number of solutions or outputs UNSAT, meaning that there are no adversarial samples with up to k bit perturbation. Table 4 shows the average number of adversarial samples and their PS(adv), i.e., percentage of count to the total input space.

Table 4. Quantifying robustness for $\text{ARCH}_{1..4}$ and perturbation size from 2 to 5. ACC_b represents the percentage of benign samples in the test set labeled as the correct class. #(Adv) and PS(adv) represent the average number and percentage of adversarial samples separately. #(timeout) represents the number of times NPAQ timeouts.

Arch	ACC_b	Perturb Size	#(Adv)	PS(adv)	#(timeout)
ARCH_1	76	$k \leq 2$	561	11.10	0
		$k = 3$	26,631	16.47	0
		$k = 4$	685,415	17.48	0
		$k = 5$	16,765,457	22.27	0
ARCH_2	79	$k \leq 2$	789	15.63	0
		$k = 3$	35,156	21.74	0
		$k = 4$	928,964	23.69	0
		$k = 5$	21,011,934	27.91	0
ARCH_3	80	$k \leq 2$	518	10.25	0
		$k = 3$	24,015	14.85	0
		$k = 4$	638,530	16.28	0
		$k = 5$	18,096,758	24.04	4
ARCH_4	88	$k \leq 2$	664	13.15	0
		$k = 3$	25,917	16.03	1
		$k = 4$	830,129	21.17	4
		$k = 5$	29,138,314	38.70	17

As expected, the number of adversarial inputs increases with k. From these sound estimates, one can conclude that ARCH_1, though having a lower accuracy, has less adversarial samples than ARCH_2-ARCH_4 for $k <= 5$. ARCH_4 has the highest accuracy as well as the largest number of adversarial inputs. Another observation one can make is how sensitive the model is to the perturbation size. For example, PS(adv) for ARCH_3 varies from $10.25 - 24.04\%$.

Effectiveness of Adversarial Training. As a second example of a usage scenario, NPAQ can be used to measure how much a model improves its robustness after applying certain adversarial training defenses. In particular, prior work has claimed that plain (unhardened) BNNs are possibly more robust than hardened models—one can quantitatively verify such claims [32]. Of the many proposed adversarial defenses [32, 39, 53, 63], we select two representative defenses [32], though our methods are agnostic to how the models are obtained. We use a fast gradient sign method [39] to generate adversarial inputs with up to $k = 2$ bits perturbation for both. In defense_1, we first generate the adversarial inputs given the training set and then retrain the original models with the pre-generated adversarial inputs and training set together. In defense_2 [32], alternatively, we craft the adversarial inputs while retraining the models. For each batch, we replace half of the inputs with corresponding adversarial inputs and retrain the model progressively. We evaluate the effectiveness of these two defenses on the same images used to quantify the robustness of the previous (unhardened) BNNs. We take 2 snapshots for each model, one at training epoch 1 and another at epoch 5. This results in a total of 480 formulae corresponding to

Table 5. Estimates of adversarial samples for maximum 2-bit perturbation on $ARCH_{1..4}$ for a plain BNN (epoch 0) and for 2 defense methods at epochs 1 and 5. ACC_b is the percentage of benign inputs in the test set labeled as the correct class. #(Adv) is the number of adversarial samples.

Arch	#(Adv)	Defense 1				Defense 2			
		Epoch = 1		Epoch = 5		Epoch = 1		Epoch = 5	
		ACC_b	#(Adv)	ACC_b	#(Adv)	ACC_b	#(Adv)	ACC_b	#(Adv)
$ARCH_1$	561	82.23	942	84.04	776	82.61	615	81.88	960
$ARCH_2$	789	79.55	1,063	77.10	1,249	81.76	664	78.73	932
$ARCH_3$	518	84.12	639	85.23	431	82.97	961	82.94	804
$ARCH_4$	664	88.15	607	88.31	890	88.85	549	85.75	619

Table 6. Effectiveness of trojan attacks. TC represents the target class for the attack. Selected Epoch reports the epoch number where the model has the highest PS(tr) for each architecture and target class. x represents a timeout.

Arch	TC	Epoch 1		Epoch 10		Epoch 30		Selected Epoch
		PS(tr)	ACC_t	PS(tr)	ACC_t	PS(tr)	ACC_t	
$ARCH_1$	**0**	39.06	50.75	13.67	72.90	5.76	68.47	1
	1	42.97	43.49	70.31	74.20	42.97	67.63	10
	4	9.77	66.80	19.14	83.18	2.69	69.99	10
	5	27.73	58.35	25.78	53.30	7.42	39.77	1
	9	2.29	53.67	12.11	61.85	0.19	77.70	10
$ARCH_2$	**0**	1.51	27.98	1.46	48.30	9.38	59.36	30
	1	2.34	30.37	13.28	40.57	8.59	51.40	10
	4	1.07	38.54	0.21	27.41	0.59	37.45	1
	5	28.91	26.66	12.70	50.24	9.38	54.90	1
	9	0.15	36.39	0.38	41.81	0.44	42.99	30
$ARCH_3$	**0**	18.36	26.91	25.00	71.85	8.40	76.30	10
	1	4.79	15.23	34.38	50.57	21.48	60.33	10
	4	7.81	33.89	11.33	67.30	4.79	62.77	10
	5	26.56	63.11	19.92	71.92	18.75	79.23	1
	9	6.84	26.51	3.32	29.12	1.15	46.51	1
$ARCH_4$	**0**	x	10.40	3.32	36.89	4.88	60.14	30
	1	x	8.57	x	54.39	0.87	78.10	30
	4	x	9.95	1.44	62.46	0.82	82.47	10
	5	19.92	8.83	13.67	8.44	25.39	11.96	30
	9	x	19.64	7.03	58.39	1.44	74.83	10

adversarially trained (hardened) models. Table 5 shows the number of adversarial samples and PS(adv).

Observing the sound estimates from NPAQ, one can confirm that plain BNNs are more robust than the hardened BNNs for 11/16 models, as suggested in prior work. Further, the security analyst can compare the two defenses. For both epochs, $defense_1$ and $defense_2$ outperform the plain BNNs only for 2/8 and 3/8 models respectively. Hence, there is no significant difference between $defense_1$ and $defense_2$ for the models we trained. One can use NPAQ estimates to select a model that has high accuracy on the benign samples as well as less adversarial samples. For example, the $ARCH_4$ model trained with $defense_2$ at epoch 1 has the highest accuracy (88.85%) and 549 adversarial samples.

6.3 Case Study 2: Quantifying Effectiveness of Trojan Attacks

The effectiveness of trojan attacks is often evaluated on a *chosen* test set, drawn from a particular distribution of images with embedded trojan triggers [33, 54]. Given a trojaned model, one may be interested in evaluating how effective is the trojaning outside this particular test distribution [54]. Specifically, NPAQ can be used to count how many images with a trojan trigger are classified to the desired target label, over the space of all possible images. Property P2 from Section 3 encodes this. We can then compare the NPAQ count vs. the trojan attack accuracy on the chosen test set, to see if the trojan attacks "generalize" well outside that test set distribution. Note that space of all possible inputs is too large to enumerate.

As a representative of such analysis, we trained BNNs on the MNIST dataset with a trojaning technique adapted from Liu et al. [54] (the details of the procedure are outlined later). Our BNN models may obtain better attack effectiveness as the trojaning procedure progresses over time. Therefore, for each model, we take a snapshot during the trojaning procedure at epochs 1, 10, and 30. There are 4 models ($ARCH_1$-$ARCH_4$), and for each, we train 5 different models each classifying the trojan input to a distinct output label. Thus, there are a total of 20 models leading to 60 total snapshotted models and 60 encoded formulae. If NPAQ terminates within the timeout of 24 hours, it either quantifies the number of solutions or outputs UNSAT, indicating that no trojaned input is labeled as the target output at all. The effectiveness of the trojan attack is measured by two metrics:

- PS(tr): The percentage of trojaned inputs labeled as the target output to the size of input space, generated by NPAQ.
- ACC_t: The percentage of trojaned inputs in the chosen test set labeled as the desired target output.

Table 6 reports the PS(tr) and ACC_t. Observing these sound estimates, one can conclude that the effectiveness of trojan attacks on out-of-distribution trojaned inputs greatly differs from the effectiveness measured on the test set distribution. In particular, if we focus on the models with the highest PS(tr) for each architecture and target class (across all epochs), only 50% (10 out 20) are the same as when we pick the model with highest ACC_t instead. Thus, for these models, an attack whose goal is to maximize the number of inputs under which the classifier outputs the target class will fail on most inputs out-of-distribution that have the trigger present.

Attack Procedure. The trojaning process can be arbitrarily different from ours; the use of NPAQ for verifying them does not depend on it in any way. Our procedure is adapted from that of Liu et al. which is specific to models with real-valued weights. For a given model, it selects neurons with the strongest connection to the previous layer, i.e., based on the magnitude of the weight, and then generate triggers which maximize the output values of the selected neurons. This heuristic does not apply to BNNs as they have $\{-1, 1\}$ weights. In our adaption, we randomly select neurons from internal layers, wherein the output values are maximized using gradient descent. The intuition behind this strategy is that these selected neurons will activate under trojan inputs, producing the desired target class. For this procedure, we need a set of trojan and benign samples. In our procedure, we assume that we have access to a 10,000 benign images, unlike the work in Liu et al. which generates this from the model itself. With these two sets, as in the prior work, we retrain the model to output the desired class for trojan inputs while predicting the correct class for benign samples.

6.4 Case Study 3: Quantifying Model Fairness

We use NPAQ to estimate how often a given neural net treats similar inputs, i.e., inputs differing in the value of a single feature, differently. This captures a notion of how much a sensitive feature

Table 7. NPAQ estimates of bias in BNNs $ARCH_{1..4}$ trained on the UCI Adult dataset. For changes in values of the sensitive features (marital status, gender and race), we compute, `PS(bias)`, the percentage of individuals classified as having the same annual income (=), greater than (>) and less than (<) when all the other features are kept the same.

Arch	Married → Divorced			Female → Male			White → Black		
	=	>	<	=	>	<	=	>	<
$ARCH_1$	89.22	0.00	10.78	89.17	9.13	2.07	84.87	5.57	9.16
$ARCH_2$	76.59	4.09	20.07	74.94	18.69	6.58	79.82	14.34	8.63
$ARCH_3$	72.50	4.37	21.93	80.04	9.34	12.11	78.23	6.24	18.58
$ARCH_4$	81.79	3.81	13.75	83.86	5.84	10.19	82.21	5.84	10.35

influences the model's prediction. We quantify fairness for 4 BNNs, one for each architecture $ARCH_1$-$ARCH_4$, trained on the UCI Adult (Income Census) dataset [2]. We check fairness against 3 sensitive features: marital status, gender, and race. We encode 3 queries for each model using Property P3— P5 (Section 3). Specifically, for how many people with exactly the same features, except one's marital status is "Divorced" while the other is "Married", would result in different income predictions? We form similar queries for gender ("Female" vs. "Male") and race ("White" vs. "Black") [3].

Effect of Sensitive Features. 4 models, 3 queries, and 3 different sensitive features give 36 formulae. Table 7 reports the percentage of counts generated by NPAQ. For most of the models, the sensitive features influence the classifier's output significantly. Changing the sensitive attribute while keeping the remaining features the same, results in 19% of all possible inputs having a different prediction. Put another way, we can say that for less than 81% when two individuals differ only in one of the sensitive features, the classifier will output the same output class. This means most of our models have a "fairness score" of less than 81%.

Quantifying Direction of Bias. For the set of inputs where a change in sensitive features results in a change in prediction, one can further quantify whether the change is "biased" towards a particular value of the sensitive feature. For instance, using NPAQ, we find that across all our models consistently, a change from "Married" to "Divorced" results in a change in predicted income from *LOW* to *HIGH*. [4] For $ARCH_1$, an individual with gender "Male" would more likely (9.13%) to be predicted to have a higher income than "Female" (2.07%) when all the other features are the same. However, for $ARCH_4$, a change from "Female" to "Male" would more likely result in a *HIGH* to *LOW* change in the classifier's output (10.19%). Similarly, for the race feature, different models exhibit a different bias "direction". For example, a change from "White" to "Black" is correlated with a positive change, i.e., from *LOW* income to *HIGH* income, for $ARCH_2$. The other 3 models, $ARCH_1$, $ARCH_2$, and $ARCH_4$ will predict that an individual with the same features except for the sensitive feature would likely have a *LOW* income if the race attribute is set to be "Black".

With NPAQ, we can distinguish how much the models treat individuals unfairly with respect to a sensitive feature. One can encode other fairness properties, such as defining a metric of similarity between individuals where non-sensitive features are within a distance, similar to individual fairness [24]. NPAQ can be helpful for such types of fairness formulations.

[3] We use the category and feature names verbatim as in the dataset. They do not reflect the authors' views.

[4] An income prediction of below \$50, 000 is classified as *LOW*.

7 RELATED WORK

We summarize the closely related work to NPAQ.

Non-quantitative Neural Network Verification. Our work is on quantitatively verifying neural networks, and NPAQ counts the number of discrete values that satisfy a property. We differ in our goals from many non-quantitative analyses that calculate continuous domain ranges or single witnesses of satisfying values. Pulina and Tacchella [68], who first studied the problem of verifying neural network safety, implement an abstraction-refinement algorithm that allows generating spurious examples and adding them back to the training set. Reluplex [49], an SMT solver with a theory of real arithmetic, verifies properties of feed-forward networks with ReLU activation functions. Huang et al. [45] leverage SMT by discretizing an infinite region around an input to a set of points and then prove that there is no inconsistency in the neural net outputs. Ehlers [26] scope the work to verifying the correctness and robustness properties on piece-wise activation functions, i.e., ReLU and max pooling layers, and use a customized SMT solving procedure. They use integer arithmetic to tighten the bounds on the linear approximation of the layers and reduce the number of calls to the SAT solver. Wang et al. [88] extend the use of integer arithmetic to reason about neural networks with piece-wise linear activations. Narodytska et al. [58] propose an encoding of binarized neural networks as CNF formulas and verifies robustness properties and equivalence using SAT solving techniques. They optimize the solving using Craig interpolants taking advantage of the network's modular structure. AI2 [34], DeepZ [74], DeepPoly [75] use abstract interpretation to verify the robustness of neural networks with piece-wise linear activations. They over-approximate each layer using an abstract domain, i.e., a set of logical constraints capturing certain shapes (e.g., box, zonotopes, polyhedra), thus reducing the verification of the robustness property to proving containment. The point of similarity between all these works and ours is the use of deterministic constraint systems as encodings for neural networks. However, our notion of equicardinality encodings applies to only specific constructions and is the key to preserving model counts.

Non-quantitative verification as Optimization. Several works have posed the problem of certifying robustness of neural networks as a convex optimization problem. Ruan, Huang, & Kwiatkowska [71] reduce the robustness verification of a neural network to the generic reachability problem and then solve it as a convex optimization problem. Their work provides provable guarantees of upper and lower bounds, which converges to the ground truth in the limit. Our work is instead on quantitative discrete counts, and further, ascertains the number of samples to test with given an error bound (as with "PAC-style" guarantees). Raghunathan, Steinhardt, & Percy [69] verify the robustness of one-hidden layer networks by incorporating the robustness property in the optimization function. They compute an upper bound which is the certificate of robustness against all attacks and inputs, including adversarial inputs, within l_{inf} ball of radius ϵ. Similarly, Wong and Kolter [90] train networks

with linear piecewise activation functions that are certifiably robust. Dvijotham et al. [23] address the problem of formally verifying neural networks as an optimization problem and obtain provable bounds on the tightness guarantees using a dual approach.

Quantitative Verification of Programs. Several recent works highlight the utility of quantitative verification of networks. They target the general paradigm of probabilistic programming and decision-making programs [5, 44]. FairSquare [5] proposes a probabilistic analysis for fairness properties based on weighted volume computation over formulas defining real closed fields. While FairSquare is more expressive and can be applied to potentially any model programmable in the probabilistic language, it does *not* guarantee a result computed in finite time will be within a desired error bound (only that it would converge in the limit). Webb et al. [89] use a statistical approach for quantitative verification but without provable error bounds for computed results as in NPAQ. Concurrent work by Narodytska et al. [59] uses model counting to assess the quality of machine learning explanations for binarized neural networks. In our work, we show a more general equicardinality framework for quantitatively verifying properties of binarized neural networks and instantiate 3 of these applications.

CNF Model Counting. In his seminal paper, Valiant showed that #CNF is #P-complete, where #P is the set of counting problems associated with NP decision problems [86]. Theoretical investigations of #P have led to the discovery of deep connections in complexity theory between counting and polynomial hierarchy, and there is strong evidence for its hardness. In particular, Toda showed that every problem in the polynomial hierarchy could be solved by just one invocation of #P oracle; more formally, $PH \subseteq P^{\#P}$ [82].

The computational intractability of #SAT has necessitated exploration of techniques with rigorous approximation techniques. A significant breakthrough was achieved by Stockmeyer who showed that one could compute approximation with (ϵ, δ) guarantees given access to an NP oracle [78]. The key algorithmic idea relied on the usage of hash functions but the algorithmic approach was computationally prohibitive at the time and as such did not lead to development of practical tools until early 2000s [57]. Motivated by the success of SAT solvers, in particular development of solvers capable of handling CNF and XOR constraints, there has been a surge of interest in the design of hashing-based techniques for approximate model counting for the past decade [16, 18, 27, 38, 77].

8 CONCLUSION

We present a new algorithmic framework for approximate quantitative verification of neural networks with formal PAC-style soundness. The framework defines a notion of equicardinality encodings of neural networks into CNF formulae. Such encodings preserve counts and ensure composibility under logical conjunctions. We instantiate this framework for binarized neural networks, building a prototype tool called NPAQ. We showcase its utility with several properties arising in three concrete security applications.

ACKNOWLEDGMENTS

We gratefully acknowledge Yash Pote, Shubham Sharma and Zheng Leong Chua for the useful discussions and comments on earlier drafts of this work. This research was supported by the research grant DSOCL17019 from DSO, Singapore, Crystal Centre at National University of Singapore, the National Research Foundation Singapore under its AI Singapore Programme [Award Number: AISG-RP-2018-005], NUS ODPRT Grant [R-252-000-685-133], the National Science Foundation under Grant No. DARPA N66001-15-C-4066, TWC-1409915, and Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Part of the computational work for this article was performed on resources of the National Supercomputing Centre, Singapore.

REFERENCES

[1] 2012. Correctional Offender Management Profiling for Alternative Sanctions. http://www.northpointeinc.com/files/downloads/FAQ_Document.pdf.
[2] 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml.
[3] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2011. BDDs for pseudo-Boolean constraints–revisited. In *SAT'11*.
[4] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2013. A parametric approach for smaller and better encodings of cardinality constraints. In *CP'13*.
[5] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V Nori. 2017. FairSquare: probabilistic verification of program fairness. In *OOPSLA'17*.
[6] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2011. Cardinality networks: a theoretical and empirical study. *Constraints* (2011).
[7] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML'18*.
[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR'15*.
[9] Kenneth E Batcher. 1968. Sorting networks and their applications. In *AFIPS'1968*.
[10] Siddhartha Bhattacharyya, Darren Cofer, D Musliner, Joseph Mueller, and Eric Engstrom. 2015. Certification considerations for adaptive systems. In *ICUAS'15*.
[11] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, and Others. 2016. End to end learning for self-driving cars. *arXiv* (2016).
[12] Nicholas Carlini, Chang Liu, Jernej Kos, Úlfar Erlingsson, and Dawn Song. 2018. The Secret Sharer: Measuring Unintended Neural Network Memorization & Extracting Secrets. *arXiv* (2018).
[13] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *SP'17*.
[14] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2014. Distribution-aware sampling and weighted model counting for SAT. In *AAAI'14*.
[15] Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. 2015. From Weighted to Unweighted Model Counting.. In *IJCAI'15*.
[16] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2013. A scalable approximate model counter. In *CP'13*.
[17] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2014. Balancing scalability and uniformity in SAT witness generator. In *DAC'14*.
[18] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2016. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *IJCAI'16*.
[19] Anupam Datta, Matthew Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. 2017. Use privacy in data-driven systems: Theory and experiments with machine learnt programs. In *CCS'17*.
[20] Elvis Dohmatob. 2018. Limitations of adversarial robustness: strong No Free Lunch Theorem. *arXiv* (2018).
[21] Jeffrey M. Dudek, Kuldeep S. Meel, and Moshe Y. Vardi. 2016. Combining the k-CNF and XOR phase-transitions. In *IJCAI'16*.
[22] Jeffrey M. Dudek, Kuldeep S. Meel, and Moshe Y. Vardi. 2017. The Hard Problems Are Almost Everywhere For Random CNF-XOR Formulas. In *IJCAI'17*.
[23] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. 2018. A Dual Approach to Scalable Verification of Deep Networks. In *UAI'18*.
[24] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *ITCS'12*.
[25] Niklas Eén and Niklas Sörensson. 2006. Translating Pseudo-Boolean Constraints into SAT. *JSAT* 2, 1-4 (2006), 1–26.
[26] Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *ATVA'17*.

[27] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2013. Taming the Curse of Dimensionality: Discrete Integration by Hashing and Optimization. In *ICML'13*.

[28] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. 2018. Robust physical-world attacks on deep learning models. In *CVPR'18*.

[29] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and removing disparate impact. In *SIGKDD'15*.

[30] Nic Ford, Justin Gilmer, Nicolas Carlini, and Dogus Cubuk. 2019. Adversarial Examples Are a Natural Consequence of Test Error in Noise. In *ICML'19*.

[31] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS'15*.

[32] Angus Galloway, Graham W Taylor, and Medhat Moussa. 2018. Attacking Binarized Neural Networks. In *ICLR'18*.

[33] Yansong Gao, Chang Xu, Derui Wang, Shiping Chen, Damith C Ranasinghe, and Surya Nepal. 2019. STRIP: A Defence Against Trojan Attacks on Deep Neural Networks. *arXiv* (2019).

[34] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *SP'18*.

[35] Justin Gilmer, Ryan P Adams, Ian Goodfellow, David Andersen, and George E Dahl. 2018. Motivating the rules of the game for adversarial example research. *arXiv* (2018).

[36] Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian Goodfellow. 2018. Adversarial spheres. *arXiv* (2018).

[37] Alessandro Giusti, Jerome Guzzi, Dan C. Ciresan, Fang-Lin He, Juan P. Rodriguez, Flavio Fontana, Matthias Faessler, Christian Forster, Jurgen Schmidhuber, Gianni Di Caro, Davide Scaramuzza, and Luca M. Gambardella. 2016. A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters* (2016).

[38] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model counting: A new strategy for obtaining good bounds. In *AAAI'06*.

[39] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *ICLR'15*.

[40] Radu Grosu and Scott A Smolka. 2005. Monte carlo model checking. In *TACAS'05*.

[41] Moritz Hardt, Eric Price, Nati Srebro, et al. 2016. Equality of opportunity in supervised learning. In *NIPS'16*.

[42] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. (1970).

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR'16*.

[44] Steven Holtzen, Guy Broeck, and Todd Millstein. 2018. Sound Abstraction and Decomposition of Probabilistic Programs. In *ICML'18*.

[45] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV'17*.

[46] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *NIPS'16*.

[47] Mark R. Jerrum and Alistair Sinclair. 1996. The Markov chain Monte Carlo method: an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems* (1996), 482–520.

[48] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *DASC'16*.

[49] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV'17*.

[50] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS'12*.

[51] Jaeha Kung, David Zhang, Gooitzen van der Wal, Sek Chai, and Saibal Mukhopadhyay. 2018. Efficient object detection using embedded binarized neural networks. *Journal of Signal Processing Systems* (2018).

[52] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. (2010).

[53] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Xiaolin Hu, and Jun Zhu. 2018. Defense against adversarial attacks using high-level representation guided denoiser. In *CVPR'18*.

[54] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning Attack on Neural Networks. In *NDSS'18*.

[55] Saeed Mahloujifar, Dimitrios I Diochnos, and Mohammad Mahmoody. 2018. The curse of concentration in robust learning: Evasion and poisoning attacks from concentration of measure. *arXiv* (2018).

[56] Bradley McDanel, Surat Teerapittayanon, and H T Kung. 2017. Embedded Binarized Neural Networks. In *EWSN'17*.

[57] Kuldeep S. Meel. 2017. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. Ph.D. Dissertation. Rice University.

[58] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2018. Verifying properties of binarized deep neural networks. In *AAAI'18*.

[59] Nina Narodytska, Aditya Shrotri, Kuldeep S Meel, Alexey Ignatiev, and Joao Marques-Silva. 2019. Assessing Heuristic Machine Learning Explanations with Model Counting. In *SAT'19*.

[60] Radford M Neal. 1993. Probabilistic inference using Markov chain Monte Carlo methods. (1993).

[61] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. 2016. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv* (2016).

[62] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *EuroS&P'16*.

[63] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *SP'16*.

[64] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W'17*.

[65] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *SOSP'17*.

[66] Tobias Philipp and Peter Steinke. 2015. PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF. In *Theory and Applications of Satisfiability Testing – SAT 2015*.

[67] Yash Pote, Saurabh Joshi, and Kuldeep S Meel. 2019. Phase Transition Behavior of Cardinality and XOR Constraints. In *International Joint Conferences on Artificial Intelligence*.

[68] Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *CAV'10*.

[69] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Certified Defenses against Adversarial Examples. In *ICLR'18*.

[70] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*.

[71] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *IJCAI'18*.

[72] Andy Shih, Adnan Darwiche, and Arthur Choi. 2019. Verifying Binarized Neural Networks by Angluin-Style Learning. In *SAT'19*.

[73] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *SP'17*.

[74] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and effective robustness certification. In *NIPS'18*.

[75] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. In *PACMPL'19*.

[76] Carsten Sinz. 2005. Towards an optimal CNF encoding of boolean cardinality constraints. In *CP'05*.

[77] Mate Soos and Kuldeep S Meel. 2019. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *AAAI'19*.

[78] Larry Stockmeyer. 1983. The complexity of approximate counting. In *Proc. of STOC*. 118–126.

[79] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *NIPS'14*.

[80] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *CVPR'15*.

[81] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR'14*.

[82] Seinosuke Toda. 1989. On the computational power of PP and (+) P. In *FOCS'1989*.

[83] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2018. Ensemble adversarial training: Attacks and defenses. In *ICLR'18*.

[84] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*.

[85] Jonathan Uesato, Brendan O'Donoghue, Aaron van den Oord, and Pushmeet Kohli. 2018. Adversarial risk and the dangers of evaluating against weak attacks. In *ICML'18*.

[86] Leslie G. Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* (1979).

[87] Izhar Wallach, Michael Dzamba, and Abraham Heifets. 2015. AtomNet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery. *arXiv* (2015).

[88] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *USENIX'18*.

[89] Stefan Webb, Tom Rainforth, Yee Whye Teh, and M Pawan Kumar. 2019. A Statistical Approach to Assessing Neural Network Robustness. In *ICLR'19*.

[90] Eric Wong and Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *ICML'18*.

[91] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. 2015. Fairness constraints: Mechanisms for fair classification. In *AISTATS'15*.

APPENDIX

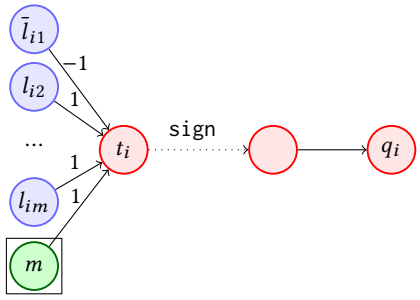


Figure 5. Disjunction gadget: Perceptron equivalent to an OR gate.

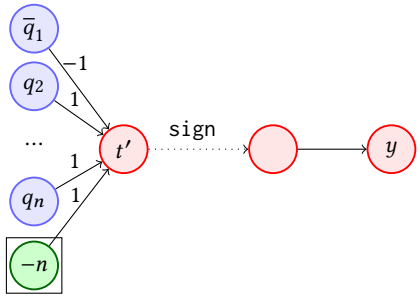


Figure 6. Conjunction gadget: Perceptron equivalent to an AND gate.

A.1 Lemma 5.1 Detailed Proof

For the ease of proof of Lemma 5.1, we first introduce the notion of independent support.

Independent Support. An independent support **ind** for a formula $F(\mathbf{x})$ is a subset of variables appearing in formula F, $\mathbf{ind} \subseteq \mathbf{x}$, that uniquely determine the values of the other variables in any satisfying assignment [17]. In other words, if there exist two satisfying assignments τ_1 and τ_2 that agree on **ind** then $\tau_1 = \tau_2$. Then $R(\mathrm{F}) = R(\mathrm{F}) \downarrow \mathbf{ind}$.

Proof. We prove that $R(\varphi) = R(\varphi) \downarrow \mathbf{x}$ by showing that $\mathbf{x}$ is an independent support for BNN. This follows directly from the construction of BNN. If $\mathbf{x}$ is an independent support then the following has to hold true:

$$\mathrm{G} = \Big(\mathrm{BNN}(\mathbf{x}, \mathbf{y}, \mathbf{a}_V) \wedge \mathrm{BNN}(\mathbf{x}', \mathbf{y}', \mathbf{a}'_V) \wedge (\mathbf{x} = \mathbf{x}') \Rightarrow (\mathbf{y} = \mathbf{y}') \wedge (\mathbf{a}_V = \mathbf{a}'_V)\Big)$$

As per Table 2, we expand $\mathrm{BNN}(\mathbf{x}, \mathbf{y})$:

$$G = \Big((\mathrm{BLK}_1(\mathbf{x}, \mathbf{v}_2^{(b)}) \wedge \mathrm{BLK}_2(\mathbf{v}_2^{(b)}, \mathbf{v}_3^{(b)}) \wedge \ldots \wedge \mathrm{OUT}(\mathbf{v}_d^{(b)}, \mathbf{ord}, \mathbf{y}) \wedge (\mathbf{x} = \mathbf{x}') \Rightarrow (\mathbf{y} = \mathbf{y}') \wedge (\mathbf{a}_V = \mathbf{a}'_V)\Big)$$

G is valid if and only if $\neg G$ is unsatisfiable.

$$\neg G = \Big((\mathrm{BLK}_1(\mathbf{x}, \mathbf{v}_2^{(b)}) \wedge \ldots \wedge \mathrm{OUT}(\mathbf{v}_d^{(b)}, \mathbf{y})) \wedge (\mathrm{BLK}_1(\mathbf{x}', \mathbf{v}'^{(b)}_2) \wedge \ldots \wedge \mathrm{OUT}(\mathbf{v}'^{(b)}_d, \mathbf{y}') \wedge (\mathbf{x} = \mathbf{x}') \wedge \neg(\mathbf{y} = \mathbf{y}')\Big) \vee \Big(\mathrm{BLK}_1(\mathbf{x}, \mathbf{v}_2^{(b)}) \wedge \ldots \wedge \mathrm{OUT}(\mathbf{v}_d^{(b)}, \mathbf{y}) \wedge (\mathrm{BLK}_1(\mathbf{x}', \mathbf{v}'^{(b)}_2) \wedge \ldots \wedge \mathrm{OUT}(\mathbf{v}'^{(b)}_d, \mathbf{y}') \wedge (\mathbf{x} = \mathbf{x}') \wedge \neg(\mathbf{a}_V = \mathbf{a}'_V)\Big)$$

The first block's formula's introduced variables $\mathbf{v}_2^{(b)}$ are uniquely determined by $\mathbf{x}$. For every formula BLK_k corresponding to an internal block the introduced variables are uniquely determined by the input variables. Similarly, for the output block (formula OUT in Table 2). If $\mathbf{x} = \mathbf{x}'$ then $\mathbf{v}_2^{(b)} = \mathbf{v}'^{(b)}_2, \ldots \Rightarrow \mathbf{a}_V = \mathbf{a}'_V$, so the second clause is not satisfied. Then, since $\mathbf{v}_d^{(b)} = \mathbf{v}'^{(b)}_d \Rightarrow \mathbf{y} = \mathbf{y}'$. Thus, G is a valid formula which implies that $\mathbf{x}$ forms an independent support for the BNN formula $\Rightarrow R(\varphi) = R(\varphi) \downarrow \mathbf{x}$.

□

A.2 Quantitative Verification is #P-hard

We prove that quantitative verification is #P-hard by reducing the problem of model counting for logical formulas to quantitative verification of neural networks. We show how an arbitrary CNF formula F can be transformed into a binarized neural net f and a specification φ such that the number of models for F is the same as φ, i.e., $|R(\varphi)| = |R(F)|$. Even for this restricted class of multilayer perceptrons quantitative verification turns out to be #P-hard. Hence, in general, quantitative verification over multilayer perceptrons is #P-hard.

Theorem A.1. *NQV (φ) is #P-hard, where φ is a specification for a property P over binarized neural nets.*

Proof. We proceed by constructing a mapping between the propositional variables of the formula F and the inputs of the BNN. We represent the logical formula as a logical circuit with the gates AND, OR, NOT corresponding to $\wedge, \vee, \neg$. In the following, we show that for each of the gates there exist an equivalent representation as a perceptron. For the OR gate we construct an equivalent perceptron, i.e., for every clause C_i of the formula F, we construct a perceptron. The perceptron is activated only if the inputs correspond to a satisfying assignment to the formula F. Similarly, we show a construction for the AND gate. Thus, we construct a BNN that composes these gates such that it can represent the logical formula exactly.

Let F be a CNF formula $F = C_1 \wedge C_2 \wedge \ldots C_n$. We denote the literals appearing in clause C_i as $l_{ij}, j = 1, ..m$. Let $\tau : Supp(F) \rightarrow \{0, 1\}$ be an assignment for F where $Supp(F)$ represents the propositional variables F is defined on. We say F is satisfiable if there exists an assignment τ such that $\tau(F) = 1$. The binarized neural net f has inputs $\mathbf{x}$ and one output y, $y = N(\mathbf{x})$, and $f : \{-1, 1\}^{m \cdot n} \rightarrow \{0, 1\}$. This can be easily extended to multi-class output.

We first map the propositional variables in $Supp(F)$ to variables in the binary domain $\{-1, 1\}$. For every clause C_i, for every literal

$l_{ij} \in \{0,1\}$ there is a corresponding input to the neural net $x_{ij} \in \{-1,1\}$ such that $l_{ij} \Leftrightarrow x_{ij} = 1 \wedge \overline{l_{ij}} \Leftrightarrow x_{ij} = -1$. For each input variable x_{ij} the weight of the neuron connection is 1 if the propositional variable l_{ij} appears as a positive literal in the C_i clause and -1 if it appears as a negative literal $\overline{l_{ij}}$ in C_i.

For every clause C_i we construct a *disjunction gadget*, a perceptron equivalent function to the OR gate (Figure 5). Given m inputs $x_{i1}, x_{i2}, \ldots x_{im} \in \{-1,1\}$, the disjunction gadget determines the output of neuron q_i. The output is the linear layer is $t_i = \sum_{j=1}^{m} w_j \cdot x_{ij} + m$. The output neuron q_i is 1 if the activation function $\texttt{sign}(t_i)$ returns 1. Namely, the output is 1 only if at least one literal is true, i.e., not all $w_j \cdot x_{ij}$ terms evaluate to -1. Notice that we only need $m+2$ neurons (m for the inputs and 2 for the intermediate outputs) for each clause C_i with m literals. Next, we introduce the *conjunction gadget* which, given n inputs $q_1, \ldots, q_n \in \{-1,1\}$ outputs $y = 1$ only if $q_1 + q_2 + \ldots + q_n \geq n$ (Figure 6). The linear layer's output is $t' = \sum_{i=1}^{n} w_i \cdot q_i - n$ over which we apply the `sign` activation function. The output of this gadget, $y = \sum_{i=1}^{n} w_i \cdot q_i \geq n$, is 1 if all of the variables q_i are 1, i.e., if all the clauses are satisfied.Notice that if we consider the batch normalization a transformation over t_i that returns t_i, we can obtain a binarized neural network f.

If the output of f on inputs $\mathbf{x}$ is 1 the formula F is SAT, otherwise it is UNSAT. Moreover, the binarized neural network constructed for binary input vectors of size $m \times n$ outputs $y = 1$ for every satisfying assignment τ of the formula F, i.e., $f(\tau(\mathbf{x})) = 1$. Given a procedure #SAT(F) that accepts formula F and outputs a number r which is the number of satisfying assignments, it will also compute the number of inputs for which the output of the BNN is 1. Specifically, we can construct a quantitative verifier for the neural net f and a specification $\varphi(\mathbf{x}, y) = (y = N(\mathbf{x})) \wedge y = 1$ using #SAT(F).

Reduction is polynomial. The size of the formula F is the size of the input $\mathbf{x}$ to the neural net, i.e., $m \cdot n$. The neural net has $n+1$ perceptrons (n for each disjunction gadget and one for the conjunction gadget).

□