Precise Quantitative Analysis of Binarized Neural Networks: A BDD-based Approach

YEDI ZHANG, ShanghaiTech University, China
ZHE ZHAO, ShanghaiTech University, China
GUANGKE CHEN, ShanghaiTech University, China
FU SONG[*][†], ShanghaiTech University, China
TAOLUE CHEN, Birkbeck, University of London, UK

As a new programming paradigm, neural network based machine learning has expanded its application to many real-world problems. Due to the black-box nature of neural networks, verifying and explaining their behavior is becoming increasingly important, especially when they are deployed in safety-critical applications. Existing verification work mostly focuses on qualitative verification which asks whether there exists an input (in a specified region) for a neural network such that a property (e.g., local robustness) is violated. However, in many practical applications, such an (adversarial) input almost surely exists which makes a qualitative answer less meaningful. In this work, we study a more interesting yet more challenging problem, i.e., *quantitative* verification of neural networks, which asks how often a property is satisfied or violated. We target binarized neural networks (BNNs), the 1-bit quantization of general neural networks. BNNs have attracted increasing attentions in deep learning recently, as they can drastically reduce memory storage and execution time with bit-wise operations, which is crucial in recourse-constrained scenarios, e.g., embedded devices for Internet of Things. Towards quantitative verification of BNNs, we propose a novel algorithmic approach for encoding BNNs as Binary Decision Diagrams (BDDs), a widely studied model in formal verification and knowledge representation. By exploiting the internal structure of the BNNs, our encoding translates the input-output relation of blocks in BNNs to cardinality constraints which are then encoded by BDDs. Based on the new BDD encoding, we develop a quantitative verification framework for BNNs where precise and comprehensive analysis of BNNs can be performed. To improve the scalability of BDD encoding, we also investigate parallelization strategies at various levels. We demonstrate applications of our framework by providing quantitative robustness verification and interpretability for BNNs. An extensive experimental evaluation confirms the effectiveness and efficiency of our approach.

CCS Concepts: • **Computing methodologies → Neural networks**; • **Security and privacy → Software security engineering**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: binarized neural networks, binary decision diagrams, formal verification, robustness, interpretability

[*] Also with Shanghai Engineering Research Center of Intelligent Vision and Imaging.
[†] Corresponding author

Authors' addresses: Yedi Zhang, zhangyd1@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Zhe Zhao, zhaozhe1@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Guangke Chen, chengk@shanghaitech.edu.cn, ShanghaiTech University, Shanghai, China; Fu Song, songfu@shanghaitech.edu.cn, ShanghaiTech University, China; Taolue Chen, t.chen@bbk.ac.uk, Birkbeck, University of London, London, UK.

1049-331X/2018/9-ART $15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, and Taolue Chen. 2018. Precise Quantitative Analysis of Binarized Neural Networks: A BDD-based Approach. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (September 2018), 51 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Background. Neural network based machine learning has become a new programming paradigm [69], which arguably takes over traditional software programs in various application domains. It has achieved state-of-the-art performance in real-world tasks such as autonomous driving [6] and medical diagnostics [82]. Deep neural networks (DNNs) usually contain a great number of parameters which are typically stored as 32/64-bit floating-point numbers, and require a massive amount of floating-point operations to compute the output for a single input [95]. As a result, it is often challenging to deploy them on resource-constrained embedded devices for, e.g., Internet of Things and mobile devices. To mitigate the issue, quantization, which quantizes 32/64-bit floating-points to low bit-width fixed-points (e.g., 4-bits) with little accuracy loss [39], emerges as a promising technique to reduce the resource requirement. In particular, binarized neural networks (BNNs) [45] represent the case of 1-bit quantization using the bipolar binaries ±1. BNNs can drastically reduce memory storage and execution time with bit-wise operations, hence substantially improve the time and energy efficiency. They have demonstrated high accuracy for a wide variety of applications [53, 65, 80].

Despite their great success, the intrinsic black-box nature of DNNs hinders the understanding of their behaviors, e.g., explanation of DNN's decision [43]. Moreover, they are notoriously vulnerable to subtle input perturbations and thus are lacking of robustness [14, 16–18, 23, 30, 54, 75, 76, 91, 94]. This is concerning as such error may lead to catastrophes when they are deployed in safety-critical applications. For example, a self-driving car can interpret a stop sign as a speed limit sign [30]. As a result, along with traditional verification and validation research in software engineering, there is a large and growing body of work on developing quality assurance techniques for DNNs, which has become one of the foci of software engineering researchers and practitioners recently. Many testing techniques have been proposed to analyze DNNs, e.g., [7, 16, 62, 63, 77, 92, 96, 114]; cf. [116] for a survey. While testing techniques are often effective in finding violations of properties (e.g., robustness), they cannot prove their absence. In a complementary direction, various formal techniques have been proposed, such as (local) robustness verification and output range analysis, which are able to prove absence of violations of properties. Typically, these methods resort to constraint solving where verification problems are encoded as a set of constraints which can be solved by off-the-shelf SAT/SMT/MILP solvers [21, 25, 28, 49, 78, 97]. Although this class of approaches is sound and complete, they usually suffer from scalability issues. In contrast, incomplete methods usually rely on approximation for better scalability, but may produce false positives. Such techniques include layer-by-layer approximation [112], layer-by-layer discretization [44], abstract interpretation [33, 87, 88] and interval analysis [104], to name a few.

Verification for quantized DNNs. Most existing DNN verification techniques focus on *real-numbered* DNNs only. Verification of *quantized* DNNs has not been thoroughly explored so far, although recent results have highlighted its importance: it was shown that a quantized counterpart does not necessarily preserve the properties satisfied by the real-numbered DNN after quantization [14, 35]. Indeed, the fixed-point number semantics effectively yields a discrete state space for the verification of quantized DNNs whereas real-numbered DNNs feature a continuous state space. The discrepancy could invalidate the current verification techniques for real-numbered DNNs when they are directly applied to their quantized counterparts (e.g., both false negatives and false positives could occur). Therefore, dedicated techniques have to be investigated for rigorously verifying quantized DNNs.

Broadly speaking, the existing verification techniques for quantized DNNs make use of constraint solving which is based on either SAT/SMT or (reduced, ordered) binary decision diagrams (BDDs). A majority of work resorts to SAT/SMT solving. For the 1-bit quantization (i.e., BNNs), typically BNNs are transformed into Boolean formulas where SAT solving is harnessed [20, 52, 71, 72]. Some recent work also studies variants of BNNs [47, 74], i.e., BNNs with ternary weights. For quantized DNNs with multiple bits (i.e., fixed-points), it is natural to encode them as quantifier-free SMT formulas (e.g., using bit-vector and fixed-point theories [9, 35, 41]), so that off-the-shelf SMT solvers can be leveraged. In another direction, BDD-based approaches currently can tackle BNNs only [83]. The general method is to encode a BNN and an input region as a BDD, based on which various analysis can be performed via queries on the BDD. The crux of the approach is how to generate the BDD model efficiently. In the work [83], the BDD model is constructed by BDD-learning [70], which, similar to Angluin's L^* learning algorithm [3], requires both membership checking and equivalence checking. To this end, in [83], the membership checking is done by querying the BDD for each input; the equivalence checking is done by transforming the BDD model and BNN to two Boolean formulas, and then checking the equivalence of the two Boolean formulas under the input region (encoded in Boolean formula) via SAT solving. This construction requires n equivalence queries and $6n^2 + n \cdot \log(m)$ membership queries, where n is the number of BDD nodes and m is the number of variables in the BDD. Due to the intractability of SAT solving (i.e., NP-complete), currently this technique is limited to toy BNNs, e.g., 64 input size, 5 hidden neurons, and 2 output size with relatively small input regions.

Quantitative verification. The existing work mostly focuses on *qualitative* verification, which typically asks whether there exists an input in a specified region for a neural network such that a property (e.g., local robustness) is violated. Qualitative verification is able to prove various properties, or otherwise often produce a counterexample when a property is violated. However, in many practical applications, checking the existence only is not sufficient. Indeed, for local robustness, such an (adversarial) input almost surely exists [14, 19, 24, 36, 37, 64, 100, 120], which makes a qualitative answer less meaningful. Instead, *quantitative* verification, which asks how often a property is satisfied or violated, is far more useful as it could provide a quantitative guarantee for neural networks. Such a quantitative guarantee is essential to certify, for instance, certain implementations of neural network based perceptual components against safety standards of autonomous vehicles specifying failure rates of these components [48, 51]. Quantitative analysis of general neural networks is challenging, and has received little attention so far. To the best of our knowledge, DeepSRGR [115] is the first quantitative robustness verification approach for real-numbered DNNs. DeepSRGR leverages the abstract interpretation technique, hence is sound but incomplete. For BNNs, approximate SAT model-counting solvers (♯SAT) are leveraged [8, 73] based on the SAT encoding for the qualitative counterpart. Though probably approximately correct (PAC) guarantees can be provided, the verification cost is usually prohibitively high to achieve higher precision and confidence.

Our contribution. In this article, we propose a BDD-based framework, named BNNQuanalyst, to support quantitative analysis of BNNs. The main challenge of the analysis is to efficiently build BDD models from BNNs [73]. In contrast to the prior work [83] which largely treats the BNN as a blackbox and uses BDD-learning, we directly encode a BNN and the associated input region into a BDD model. Our encoding is based on the structure characterization of BNNs. In a nutshell, a BNN is a sequential composition of multiple internal blocks and one output block. Each internal block comprises a handful of layers and captures a function $f : \{+1, -1\}^n \to \{+1, -1\}^m$ over the bipolar domain $\{+1, -1\}$, where n (resp. m) denotes the number of inputs (resp. outputs) of the block. (Note that the inputs and outputs may not be binarized for the layers inside the blocks.) By encoding the bipolar domain $\{+1, -1\}$ as the Boolean domain $\{0, 1\}$, the function f can be alternatively rewritten as a function over the standard Boolean domain, i.e., $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$. A cornerstone of our encoding is the observation that the i-th output y_i of each internal block can be captured by a cardinality constraint of the form $\sum_{j=1}^{n} \ell_j \geq k$ such that $y_i \Leftrightarrow \sum_{j=1}^{n} \ell_j \geq k$, namely, the i-th output y_i of the internal block is 1 if and only if the cardinality constraint $\sum_{j=1}^{n} \ell_j \geq k$ holds, where each literal ℓ_j is either x_j or $\neg x_j$ for the input variable x_j, and k is a constant. An output of the output block is one of the s classification labels, and can be captured by a conjunction of cardinality constraints $\bigwedge_{i=1}^{m} \sum_{j=1}^{n_m} \ell_{i,j} \geq k_i$ such that the class is produced by the output block if and only if the constraint $\bigwedge_{i=1}^{m} \sum_{j=1}^{n_m} \ell_{i,j} \geq k_i$ holds. We then present an efficient algorithm to encode a cardinality constraint $\sum_{j=1}^{n} \ell_j \geq k$ as a BDD with $O((n - k) \cdot k)$ nodes in $O((n - k) \cdot k)$ time. As a result, the input-output relation of each block can be encoded as a BDD, the composition of which yields the BDD model for the entire BNN. A distinguished advantage of our BDD encoding lies in its support of incremental encoding. In particular, when different input regions are of interest, there is no need to construct the BDD of the entire BNN from scratch.

To improve the efficiency of BDD encoding, we propose two strategies, namely, input propagation and divide-and-conquer. The former forward propagates a given input region block-by-block. This can give the feasible input space of each block, which can reduce the number of BDD nodes when constructing BDD models from cardinality constraints. The latter is used when constructing the BDD model of an internal block. Namely, we recursively compute the BDDs for the first half and the second half of the cardinality constraints, which are to be combined by the BDD AND-operation. The divide-and-conquer strategy does not reduce the number of AND-operations, but can reduce the size of the intermediate BDDs. To leverage modern CPUs' computing capability, we also investigate parallelization strategies at various levels, namely, BDD operations, BDD encoding of each block, and BDD construction of an entire BNN. We show that these strategies (except for parallel BDD construction of an entire BNN) can significantly improve the efficiency of BDD encoding for large BNNs and input regions.

Encoding BNNs as BDDs enables a wide variety of applications in security analysis and decision explanation of BNNs. In this work, we highlight two of them within our framework, i.e., robustness analysis and interpretability. For the former, we consider two quantitative variants of the robustness analysis: (1) how many adversarial examples does the BNN have in the input region, and (2) how many of them are misclassified to each class? We further provide an algorithm to incrementally compute the (locally) maximal Hamming distance within which the BNN satisfies the desired robustness properties. For the latter, we consider two problems: (1) why some inputs are (mis)classified into a class by the BNN and (2) are there any essential features in the input region that are common for all samples classified into a class?

Experimental results. We implemented our approach in a tool BNNQuanalyst using two BDD packages: CU Decision Diagram (CUDD) [90] and Sylvan [102], where CUDD is a widely used sequential BDD package while Sylvan is a promising parallel BDD package. We have evaluated BNNQuanalyst by encoding and verifying properties of various BNNs trained on two popular datasets, i.e., MNIST [55] and Fashion-MNIST [113]. The experiments show that BNNQuanalyst scales to BNNs with 4 internal blocks (i.e., 12 layers), 200 hidden neurons, and 784 input size. To the best of our knowledge, it is the first work to precisely and quantitatively analyze such large BNNs that go significantly beyond the state-of-the-art, and is significantly more efficient and scalable than the BDD-learning-based technique [83]. Then, we demonstrate how BNNQuanalyst can be used in quantitative robustness analysis and decision explanation of BNNs. For quantitative robustness analysis, our experimental results show that BNNQuanalyst can be considerably (hundreds of times on average) faster and more accurate than the state-of-the-art approximate ♯SAT-based approach [8]. It can also compute precisely the distribution of predicted classes of the images in the input region as well as the locally maximal Hamming distances on several BNNs. For decision explanation, we show the effectiveness of BNNQuanalyst in computing prime-implicant features and essential features of the given input region for some target classes, where prime-implicant features are a sufficient condition, if fixed, the prediction is guaranteed no matter how the remaining features change in the input region, while essential features are a necessary condition, on which all samples in an input region that are classified into the same class must agree.

In general, our main contributions can be summarized as follows.

- We introduce a novel algorithmic approach for encoding BNNs into BDDs that exactly preserves the semantics of BNNs, which supports incremental encoding.
- We explore parallelization strategies at various levels to accelerate BDD encoding, most of which can significantly improve the BDD encoding efficiency.
- We propose a framework for quantitative verification of BNNs and in particular, we demonstrate the robustness analysis and interpretability of BNNs.
- We implement the framework as an end-to-end tool BNNQuanalyst, and conduct thorough experiments on various BNNs, demonstrating its efficiency and effectiveness.

Outline. The remainder of this paper is organized as follows. In Section 2, we introduce binarized neural networks, binary decision diagrams and two binary decision diagram packages used in this work. Section 3 presents our BDD-based quantitative analysis framework, including some design choices to improve the overall encoding efficiency. In Section 4, we investigate feasible parallelization strategies. Section 5 presents two applications of our BDD encoding, namely, robustness analysis and interpretability. In Section 6, we report the evaluation results. We discuss the related work in Section 7. Finally, we conclude this work in Section 8.

This article significantly extends the results presented in [118]. (1) We add more detailed descriptions of the algorithms, missing proofs of lemmas and theorems, and provide a more up-to-date discussion of the related work. (2) We investigate various parallelization strategies to accelerate BDD encoding, i.e., parallel BDD operations, parallel BDD encoding of blocks and parallel BDD encoding of an entire BNN (cf. Section 4). We thoroughly evaluate these parallelization strategies (cf. Section 6.1.4, Section 6.2 and Section 6.3). To the best of our knowledge, it is the first work which explores parallelization for BNN verification. (3) We empirically study the performance of our graph-based algorithm and the DP-based algorithm [27] for compiling cardinality constraints into BDDs (cf. Section 6.1.1), the divide-and-conquer strategy for BDD encoding of blocks (cf. Section 6.1.2), and our input propagation (cf. Section 6.1.3). (4) We compare our approach with the BDD-learning-based approach [83] for BDD encoding (cf. Section 6.2.2) and other possible approaches (Section 6.3.2) for quantitative robustness verification.

2 PRELIMINARIES

In this section, we briefly introduce binarized neural networks (BNNs) and (reduced, ordered) binary decision diagrams (BDDs), as well as the two BDD packages used in this work.

We denote by $\mathbb{R}$, $\mathbb{N}$, $\mathbb{B}$, and $\mathbb{B}_{\pm 1}$ the set of real numbers, the set of natural numbers, the standard Boolean domain $\{0, 1\}$ and the bipolar domain $\{+1, -1\}$, respectively. For a given positive integer $n \in \mathbb{N}$, we let $[n] := \{1, \cdots, n\}$. We will use $\mathbf{W}, \mathbf{W}', \cdots$ to denote (2-dimensional) matrices, $\mathbf{x}, \mathbf{v}, \cdots$ to denote (row) vectors, and $x, v, \cdots$ to denote scalars. We denote by $\mathbf{W}_{i,:}$ and $\mathbf{W}_{:,j}$ the i-th row and j-th column of the matrix $\mathbf{W}$. Similarly, we denote by $\mathbf{x}_j$ and $\mathbf{W}_{i,j}$ the j-th entry of the vector $\mathbf{x}$ and

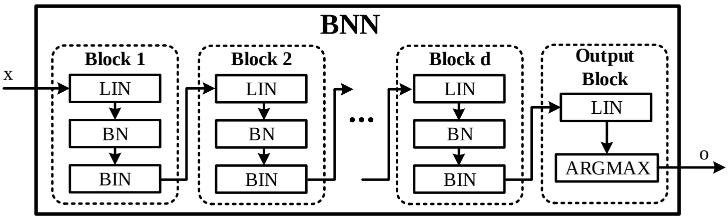


Fig. 1. Architecture of a BNN with $d + 1$ blocks

matrix $\mathbf{W}_{i,:}$. In this work, Boolean values 1/0 will be used as integers 1/0 in arithmetic computations without typecasting.

2.1 Binarized Neural Networks

A Binarized Neural Network (BNN) [45] is a neural network where weights and activations are predominantly binarized over the bipolar domain $\mathbb{B}_{\pm 1}$. In this work, we consider feed-forward BNNs. As shown in Figure 1, a (feed-forward) BNN can be seen as a sequential composition of several internal blocks and one output block. Each internal block comprises three layers: a linear layer (LIN), a batch normalization layer (BN), and a binarization layer (BIN). The output block comprises a linear layer and an ARGMAX layer. Note that the input/output of the internal blocks and the input of the output block are all vectors over the bipolar domain $\mathbb{B}_{\pm 1}$.

Definition 2.1. A BNN $\mathcal{N} : \mathbb{B}_{\pm 1}^{n_1} \rightarrow \mathbb{B}^s$ with s classes (i.e., classification labels) is given by a tuple of blocks $(t_1, \cdots, t_d, t_{d+1})$ such that

$$\mathcal{N} = t_{d+1} \circ t_d \circ \cdots \circ t_1,$$

Here, we have that

- for every $i \in [d]$, $t_i : \mathbb{B}_{\pm 1}^{n_i} \rightarrow \mathbb{B}_{\pm 1}^{n_{i+1}}$ is the i-th internal block comprising a LIN layer t_i^{lin}, a BN layer t_i^{bn} and a BIN layer t_i^{bin} with $t_i = t_i^{bin} \circ t_i^{bn} \circ t_i^{lin}$,
- $t_{d+1} : \mathbb{B}_{\pm 1}^{n_{d+1}} \rightarrow \mathbb{B}^s$ is the output block comprising a LIN layer t_{d+1}^{lin} and an ARGMAX layer t_{d+1}^{am} with $t_{d+1} = t_{d+1}^{am} \circ t_{d+1}^{lin}$,

where t_i^{bin}, t_i^{bn}, t_i^{lin} for $i \in [d]$, t_{d+1}^{lin} and t_{d+1}^{am} are given in Table 1.

Intuitively, a LIN layer is a fully connected layer, acted as a linear transformation $t^{lin} : \mathbb{B}_{\pm 1}^m \rightarrow \mathbb{R}^n$ over vectors such that $t^{lin}(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$, where $\mathbf{W} \in \mathbb{B}_{\pm 1}^{m \times n}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^n$ is the bias vector. A BN layer following a LIN layer forms a linear transformation $t^{bn} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ such that $t^{bn}(\mathbf{x}) = \mathbf{y}$, where for every $j \in [n]$, $\mathbf{y}_j = \alpha_j \cdot (\frac{\mathbf{x}_j - \mu_j}{\sigma_j}) + \gamma_j$, α_j and γ_j denote the j^{th} elements of the weight vector $\alpha \in \mathbb{R}^n$ and the bias vector $\gamma \in \mathbb{R}^n$, μ_j and σ_j denote the mean and standard deviation (assuming $\sigma_i > 0$). A BN layer is used to standardize and normalize the output vector of the preceding LIN layer. A BIN layer is used to binarize the real-numbered output vector of the preceding BN layer. In this work, we consider the `sign` function which is widely used in BNNs to binarize real-numbered vectors. Thus, a BIN layer with n inputs forms a non-linear transformation $t^{bin} : \mathbb{R}^n \rightarrow \mathbb{B}_{\pm 1}^n$ such that the j-the entry of $t^{bin}(\mathbf{x})$ is +1 if $\mathbf{x}_j \geq 0$, and -1 otherwise. An ARGMAX layer $t^{am} : \mathbb{R}^s \rightarrow \mathbb{B}^s$ follows a LIN layer and outputs the index of the largest entry as the predicted class which is represented by a one-hot vector. (In case there is more than one such entry, the first one is returned.) Formally, given a BNN $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$ and an input $\mathbf{x} \in \mathbb{B}_{\pm 1}^{n_1}$, $\mathcal{N}(\mathbf{x}) \in \mathbb{B}^s$ is a one-hot vector in which the index of the non-zero entry is the predicted class.

Table 1. Definitions of layers in BNNs, where $n_{d+2} = s$, t_i^{bin} is the $\mathtt{sign}$ function and $\arg\max(\cdot)$ returns the index of the largest entry which occurs first.

Layer	Function	Parameters	Definition
LIN	$t_i^{lin} : \mathbb{B}_{\pm 1}^{n_i} \longrightarrow \mathbb{R}^{n_{i+1}}$	Weight matrix: $\mathbf{W} \in \mathbb{B}_{\pm 1}^{n_i \times n_{i+1}}$ Bias (row) vector: $\mathbf{b} \in \mathbb{R}^{n_{i+1}}$	$t_i^{lin}(\mathbf{x}) = \mathbf{y}$ where $\forall j \in [n_{i+1}]$, $\mathbf{y}_j = \langle \mathbf{x}, \mathbf{W}_{:,j} \rangle + \mathbf{b}_j$
BN	$t_i^{bn} : \mathbb{R}^{n_{i+1}} \longrightarrow \mathbb{R}^{n_{i+1}}$	Weight vectors: $\alpha \in \mathbb{R}^{n_{i+1}}$ Bias vector: $\gamma \in \mathbb{R}^{n_{i+1}}$ Mean vector: $\mu \in \mathbb{R}^{n_{i+1}}$ Std. dev. vector: $\sigma \in \mathbb{R}^{n_{i+1}}$	$t_i^{bn}(\mathbf{x}) = \mathbf{y}$ where $\forall j \in [n_{i+1}]$, $\mathbf{y}_j = \alpha_j \cdot (\frac{\mathbf{x}_j - \mu_j}{\sigma_j}) + \gamma_j$
BIN	$t_i^{bin} : \mathbb{R}^{n_{i+1}} \longrightarrow \mathbb{B}_{\pm 1}^{n_{i+1}}$	-	$t_i^{bin}(\mathbf{x}) = \mathtt{sign}(\mathbf{x}) = \mathbf{y}$ where $\forall j \in [n_{i+1}]$, $\mathbf{y}_j = \begin{cases} +1, & \text{if } \mathbf{x}_j \geq 0; \\ -1, & \text{otherwise.} \end{cases}$
ARGMAX	$t_{d+1}^{am} : \mathbb{R}^s \longrightarrow \mathbb{B}^s$	-	$t_{d+1}^{am}(\mathbf{x}) = \mathbf{y}$ where $\forall j \in [s]$, $\mathbf{y}_j = 1 \Leftrightarrow j = \arg\max(\mathbf{x})$

2.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [12] is a rooted acyclic directed graph, where non-terminal nodes v are labeled by Boolean variables $\mathsf{var}(v)$ and terminal nodes (leaves) v are labeled with values $\mathsf{val}(v) \in \mathbb{B}$, referred to as the 1-leaf and the 0-leaf, respectively. Each non-terminal node v has two outgoing edges: $\mathsf{hi}(v)$ and $\mathsf{lo}(v)$, where $\mathsf{hi}(v)$ denotes that the variable $\mathsf{var}(v)$ is assigned by 1 (i.e., $\mathsf{var}(v) = 1$), and $\mathsf{lo}(v)$ denotes that the variable $\mathsf{var}(v)$ is assigned by 0 (i.e., $\mathsf{var}(v) = 0$). By a slight abuse of notation, we will also refer to $\mathsf{hi}(v)$ and $\mathsf{lo}(v)$ as the hi- and lo-children of v, respectively, when it is clear from the context.

A BDD with a pre-defined total variable ordering is called Ordered Binary Decision Diagram (OBDD). Assuming that $x_1 < x_2 < \cdots < x_m$ is the variable ordering, OBDD satisfies that for each pair of nodes v and v', if $v' \in \{\mathsf{hi}(v), \mathsf{lo}(v)\}$, then $\mathsf{var}(v) < \mathsf{var}(v')$. In the graphical representation of BDDs, the edges $\mathsf{hi}(v)$ and $\mathsf{lo}(v)$ are depicted by solid and dashed lines, respectively. Multi-Terminal Binary Decision Diagrams (MTBDDs) are a generalization of BDDs in which the terminal nodes are not restricted to be 0 or 1.

A BDD is *reduced* if the following conditions hold:

(1) it has only one 1-leaf and one 0-leaf, i.e., no duplicate terminal nodes;
(2) it does not contain a node v such that $\mathsf{hi}(v) = \mathsf{lo}(v)$, i.e., the hi- and lo-children of each node v should be distinct; and
(3) it does not contain two distinct non-terminal nodes v and v' such that $\mathsf{var}(v) = \mathsf{var}(v')$, $\mathsf{hi}(v) = \mathsf{hi}(v')$ and $\mathsf{lo}(v) = \mathsf{lo}(v')$, namely, no isomorphic sub-trees.

Hereafter, we assume that BDDs are reduced and ordered.

Bryant [12] showed that BDDs can serve as a canonical form of Boolean functions. Given a BDD over variables $x_1, \cdots, x_m$, each non-terminal node v with $\mathsf{var}(v) = x_i$ represents a Boolean function $f_v = (x_i \wedge f_{\mathsf{hi}(v)}) \vee (\neg x_i \wedge f_{\mathsf{lo}(v)})$. Operations on Boolean functions can usually be efficiently implemented via manipulating their BDD representations. A good variable ordering is crucial for the performance of BDD manipulations while the task of finding an optimal ordering for a Boolean function is NP-hard in general. In practice, to store and manipulate BDDs efficiently, nodes are stored in a hash table and recent computed results are stored in a cache to avoid duplicated computations. In this work, we will use some basic BDD operations such as ITE (If-Then-Else), Xor (exclusive-OR), Xnor (exclusive-NOR, i.e., a Xnor $b = \neg(a$ Xor $b)$), SatAll(v) (i.e., returning the

Table 2. Some Basic BDD operations, where $op \in \{\textsc{And}, \textsc{Or}, \textsc{Xor}, \textsc{Xnor}\}$, m denotes the number of the Boolean variables and $|v|$ denotes the number of nodes in the BDD v.

Operation	Definition	Complexity	Operation	Definition	Complexity
$v = \textsc{New}(x)$	$f_v = x$	$O(1)$	$v = \textsc{Const}(1)$	$f_v = 1$	$O(1)$
$\textsc{Not}(v)$	$\neg f_v$	$O(\|1\|)$	$v = \textsc{Const}(0)$	$f_v = 0$	$O(1)$
$\textsc{Apply}(v, v', op)$	$f_v \ op \ f_{v'}$	$O(\|v\| \cdot \|v'\|)$	$\textsc{Exists}(v, X)$	$\exists X.f_v$	$O(\|v\| \cdot 2^{2m})$
$\textsc{SatAll}(v)$	$\textsc{SatAll}(f_v)$	$O(m \cdot \|\textsc{SatAll}(f_v)\|)$	$\textsc{RelProd}(v, v', X)$	$\exists X.f_{v'} \circ f_v$	$O(\|v\| \cdot \|v'\| \cdot 2^{2m})$
$\textsc{SatCount}(v)$	$\|\textsc{SatAll}(f_v)\|$	$O(\|v\|)$	$\text{ITE}(x, v, v')$	$(x \wedge f_v) \vee (\neg x \wedge f_{v'})$	$O(\|v\| \cdot \|v'\|)$

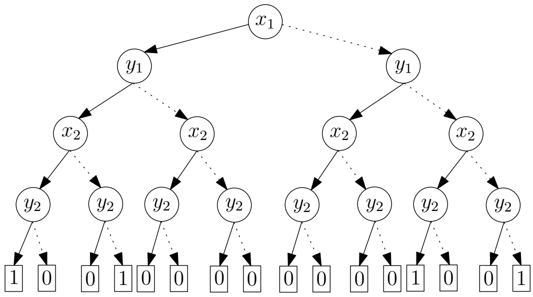


Fig. 2. The OBDD for $f(x_1, y_1, x_2, y_2)$

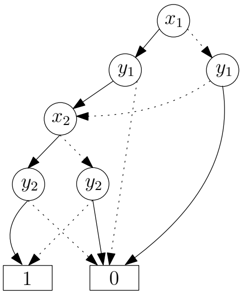


Fig. 3. The Reduced OBDD for $f(x_1, y_1, x_2, y_2)$

set of all solutions of the Boolean formula f_v) and $\textsc{SatCount}(v)$ (i.e., returning $|\textsc{SatAll}(v)|$). We denote by $\mathcal{L}(v)$ the set $\textsc{SatAll}(f_v)$. For easy reference, more operations and their worst-case time complexities are given in Table 2 [12, 13, 66], where m denotes the number of Boolean variables and $|v|$ denotes the number of nodes in the BDD v. We denote by $op(v, v')$ the operation $\textsc{Apply}(v, v', op)$, where $op \in \{\textsc{And}, \textsc{Or}, \textsc{Xor}, \textsc{Xnor}\}$.

In this work, we use BDDs to symbolically represent sets of Boolean vectors and multiple output functions. For each BDD v over Boolean variables $x_1, \cdots, x_m$ that represents a Boolean function $f_v : \{x_1, \cdots, x_m\} \to \mathbb{B}$, the BDD v essentially represents the set $\mathcal{L}(v) \subseteq \mathbb{B}^m$, i.e., all solutions of the Boolean formula $f_v(x_1, \cdots, x_m)$. The function f_v is often called the characteristic function of the set $\mathcal{L}(v)$. A multiple output function $f : \mathbb{B}^m \to \mathbb{B}^n$ can be seen as a Boolean function $f' : \mathbb{B}^{m+n} \to \mathbb{B}$ such that $f(x_1, \cdots, x_m) = (y_1, \cdots, y_n)$ iff $f'(x_1, \cdots, x_m, y_1, \cdots, y_n) = 1$, hence can also be represented by a BDD.

Example 2.2. Consider the Boolean function $f(x_1, y_1, x_2, y_2) = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. Assume that the variable ordering is $x_1 < y_1 < x_2 < y_2$. Figure 2 and Figure 3 respectively show the OBDD and Reduced OBDD of the Boolean function $f(x_1, y_1, x_2, y_2)$. The set $\textsc{SatAll}(f)$ of its solutions is $\{(1, 1, 1, 1), (1, 1, 0, 0), (0, 0, 1, 1), (0, 0, 0, 0)\}$. The Boolean function $f(x_1, y_1, x_2, y_2)$ can be seen as a multiple output function $f' : \{x_1, x_2\} \to \{y_1, y_2\}$ such that $f'(1, 1) = (1, 1)$, $f'(1, 0) = (1, 0)$, $f'(0, 1) = (0, 1)$ and $f'(0, 0) = (0, 0)$.

2.3 BDD Packages

In this section, we give a brief introduction of two BDD packages used in this work, i.e., CUDD [90] and Sylvan [102].

CUDD. CUDD is a widely used decision diagram package implemented in C. To facilitate the performance of BDD manipulation, CUDD stores all the decision diagram nodes in a unique hash table and features a cache for storing recent computed results, both of which can be automatically adjusted at runtime. The unique hash table and some auxiliary data structures make up a decision

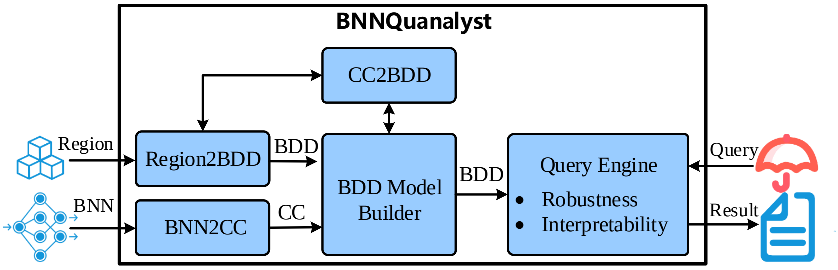


Fig. 4. Overview of BNNQuanalyst

diagram manager (DdManager), which should be initialized by calling an appropriate function with initial sizes of the subtables and cache.

The CUDD package provides a C++ interface which facilitates application via automatic garbage collection and operator overloading. Although CUDD allows to execute BDD operations from multiple threads, each thread has to use a separate DdManager, i.e., a separate unique hash table for storing decision diagram nodes, which limits its usage for parallel computing [102]. In this work, CUDD is only used for sequential computing.

Sylvan. Sylvan is a parallel decision diagram package implemented in C. It leverages the working-stealing framework Lace [103] and scalable parallel data structures to provide parallel operations on decision diagrams. Similar to CUDD, Sylvan maintains a hash table, an automatic garbage collector and a cache for managing decision diagram nodes and storing recent computed results. Both the minimum and maximum sizes of the hash table and cache are initialized by calling an appropriate function, but the sizes can be automatically adjusted at runtime. In contrast to CUDD which implements the unique hash table using several subtables and collects garbage in sequence, Sylvan directly maintains one single hash table and collects garbage in parallel. Though Sylvan implements less BDD operations than CUDD, it provides many parallel implementations of common BDD operations, e.g., Apply, Exists, RelProd, SatAll, SatCount and ITE. It also allows developers to implement parallel BDD operations at the algorithmic level. In this work, Sylvan is primarily used for parallel computing.

3 FRAMEWORK DESIGN

3.1 Overview of BNNQuanalyst

An overview of BNNQuanalyst is depicted in Figure 4. It comprises five main components: Region2BDD, BNN2CC, CC2BDD, BDD Model Builder, and Query Engine. For a fixed BNN $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$ and a region R of the input space of $\mathcal{N}$, BNNQuanalyst constructs the BDDs $(G_i^{out})_{i\in[s]}$, where the BDD G_i^{out} encodes the input-output relation of the BNN $\mathcal{N}$ in the region R for the class $i \in [s]$. Technically, the region R is partitioned into s parts represented by the BDDs $(G_i^{out})_{i\in[s]}$. For each query of the property, BNNQuanalyst analyzes $(G_i^{out})_{i\in[s]}$ and outputs the query result.

The general workflow of our approach is as follows. First, Region2BDD builds up a BDD G_R^{in} from the region R which represents the desired input space of $\mathcal{N}$ for analysis. Second, BNN2CC transforms each block of the BNN $\mathcal{N}$ into a set of cardinality constraints (CCs) similar to [8, 72]. Third, BDD Model Builder builds the BDDs $(G_i^{out})_{i\in[s]}$ from all the cardinality constraints and the BDD G_R^{in}. Both Region2BDD and BDD Model Builder invoke CC2BDD which encodes a given cardinality constraint as a BDD. Finally, Query Engine answers queries by analyzing the BDDs

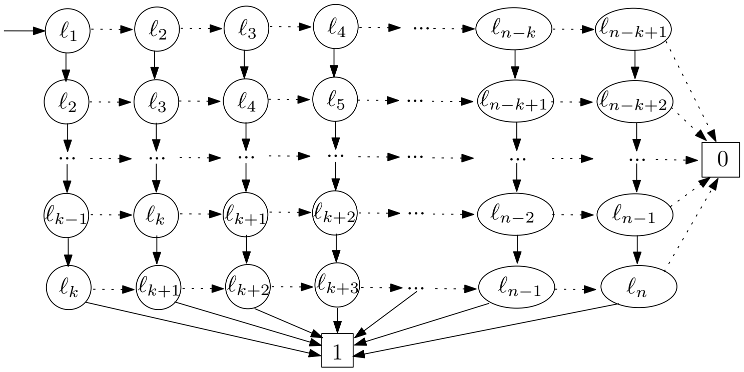


Fig. 5. A BDD-like graph representation of the cardinality constraint $\sum_{j=1}^{n} \ell_j \geq k$

Algorithm 1: BDD Construction for cardinality constraints

```
Procedure CC2BDD(CC : $\sum_{j=1}^{n} \ell_j \geq k$)
    $G_{k+1,1} = G_{k+1,2} = \cdots = G_{k+1,n-k+1} = \textsc{Const}(1)$;
    $G_{1,n-k+2} = G_{2,n-k+2} = \cdots = G_{k,n-k+2} = \textsc{Const}(0)$;
    for ($i = k;\ i \geq 1;\ i--$) do
        for ($j = n - k + 1;\ j \geq 1;\ j--$) do
            if ($\ell_{i+j-1} == \mathbf{x}_{i+j-1}$) then
                $G_{i,j} = \text{ITE}(\mathbf{x}_{i+j-1}, G_{i+1,j}, G_{i,j+1})$
            else
                $G_{i,j} = \text{ITE}(\mathbf{x}_{i+j-1}, G_{i,j+1}, G_{i+1,j})$
    return $G_{1,1}$
```

$(G_i^{out})_{i \in [s]}$. Our Query Engine currently supports two types of application queries: robustness analysis and interpretability.

In the rest of this section, we first introduce the key component CC2BDD, and then provide details of the components Region2BDD, BNN2CC, and BDD Model Builder. The Query Engine will be described in Section 5.

3.2 CC2BDD: Cardinality Constraints to BDDs

A *cardinality constraint* is a constraint of the form $\sum_{j=1}^{n} \ell_j \geq k$ over a vector $\mathbf{x}$ of Boolean variables $\{x_1, \cdots, x_n\}$ with length n, where the literal ℓ_j is either $\mathbf{x}_j$ or $\neg\mathbf{x}_j$ for each $j \in [n]$. A *solution* of the constraint $\sum_{j=1}^{n} \ell_j \geq k$ is a valuation of the Boolean variables $\{x_1, \cdots, x_n\}$ under which the constraint holds. Note that constraints of the form $\sum_{j=1}^{n} \ell_j > k$, $\sum_{j=1}^{n} \ell_j \leq k$ and $\sum_{j=1}^{n} \ell_j < k$ are equivalent to $\sum_{j=1}^{n} \ell_j \geq k+1$, $\sum_{j=1}^{n} \neg\ell_j \geq n-k$ and $\sum_{j=1}^{n} \neg\ell_j \geq n-k+1$, respectively. We assume that 1 (resp. 0) is a special cardinality constraint that always holds (resp. never holds).

To encode a cardinality constraint $\sum_{j=1}^{n} \ell_j \geq k$ as a BDD, we observe that all the possible solutions of $\sum_{j=1}^{n} \ell_j \geq k$ can be compactly represented by a BDD-like graph shown in Figure 5, where each node is labeled by a literal, and a solid (resp. dashed) edge from a node labeled by ℓ_j means that the value of the literal ℓ_j is 1 (resp. 0), called *positive literal.* Thus, each path from the ℓ_1-node to the 1-leaf through the ℓ_j-node (where $1 \leq j \leq n$) captures a set of valuations where ℓ_j followed by a (horizontal) dashed line is set to be 0 while ℓ_j followed by a (vertical) solid line is set to be 1, and all the other literals which are not along the path can take arbitrary values. Along a path, the number of positive literals is counted, and the path ends with the 1-leaf iff the number of positive literals is no less than k. Clearly, for each of these valuations of a path from the ℓ_1-node to the 1-leaf, there are at least k positive literals, hence the constraint $\sum_{j=1}^{n} \ell_j \geq k$ holds.

Based on the above observation, we build the BDD for $\sum_{j=1}^{n} \ell_j \geq k$ using Algorithm 1. It builds a BDD for each node in Figure 5, row-by-row (the index i in Algorithm 1) and from right to left (the index j in Algorithm 1). For each node at the i-th row and j-th column, the label of the node must be the literal ℓ_{i+j-1}. We build the BDD $G_{i,j} = \text{ITE}(\mathbf{x}_{i+j-1}, G_{i+1,j}, G_{i,j+1})$ if ℓ_{i+j-1} is of the form $\mathbf{x}_{i+j-1}$ (Line 7), otherwise we build the BDD $G_{i,j} = \text{ITE}(\mathbf{x}_{i+j-1}, G_{i,j+1}, G_{i+1,j})$ (Line 9). Finally, we obtain the BDD $G_{1,1}$ that encodes the solutions of $\sum_{j=1}^{n} \ell_j \geq k$.

Lemma 3.1. *For each cardinality constraint $\sum_{j=1}^{n} \ell_j \geq k$, a BDD G with $O((n-k) \cdot k)$ nodes can be computed in $O((n-k) \cdot k)$ time such that $\mathcal{L}(G)$ is the set of all the solutions of $\sum_{j=1}^{n} \ell_j \geq k$, i.e., $\mathbf{u} \in \mathcal{L}(G)$ iff ξ is a solution of $\sum_{j=1}^{n} \ell_j \geq k$, where $\xi(\mathbf{x}_j) = \mathbf{u}_j$ for each $j \in [n]$.*

Proof. Consider the cardinality constraint $\sum_{j=1}^{n} \ell_j \geq k$ and the variable ordering $\mathbf{x}_1 < \cdots < \mathbf{x}_n$, the procedure CC2BDD in Algorithm 1 constructs a BDD $G_{1,1}$ with n Boolean variables and $k(n-k+1)+2$ nodes. The outer-loop executes k iterations and the inner-loop executes $n-k+1$ iterations for each iteration of the outer-loop. For each iteration of the inner-loop, one ITE-operation is performed. Although the time complexity of $\text{ITE}(\mathbf{x}_{i+j-1}, G_{i+1,j}, G_{i,j+1})$ (resp. $\text{ITE}(\mathbf{x}_{i+j-1}, G_{i,j+1}, G_{i+1,j})$) is $O(|G_{i+1,j}| \cdot |G_{i,j+1}|)$ in general, the variable ordering $x_1 < \cdots < x_n$ ensures that only one node v with $\text{New}(v) = \mathbf{x}_{i+j-1}$ and two edges between the node v and the roots of the BDDs $G_{i+1,j}$ and $G_{i,j+1}$ are added, which can be done in $O(1)$ time, as all the roots of $G_{i+1,j}$ and $G_{i,j+1}$ are either leaves or labeled by the Boolean variable $\mathbf{x}_{i+j}$. Thus, the BDD $G_{1,1}$ can be constructed in $O((n-k) \cdot k)$ time.

To prove that $\mathcal{L}(G_{1,1})$ is the set of all the solutions of $\sum_{j=1}^{n} \ell_j \geq k$, we prove the following claim:

> For any indices i and j such that $1 \leq i \leq k$ and $1 \leq j \leq n-k+1$, any valuation ξ of $\sum_{j=1}^{n} \ell_j \geq k$, if there are at least (resp. less than) $k-i+1$ positive literals among the literals $\{\ell_{i+j-1}, \cdots, \ell_n\}$ under the valuation ξ, then the path starting from the root of the BDD $G_{i,j}$ and following the valuation ξ ends at the 1-leaf (resp. 0-leaf).

We apply induction on the indices i and j.

Base case $i = k$ and $j = n-k+1$. The BDD $G_{k,n-k+1}$ is built via

- $\text{ITE}(\mathbf{x}_n, G_{k+1,n-k+1}, G_{k,n-k+2})$ if ℓ_n is $\mathbf{x}_n$, or
- $\text{ITE}(\mathbf{x}_n, G_{k,n-k+2}, G_{k+1,n-k+1})$ if ℓ_n is $\neg\mathbf{x}_n$.

Recall that $G_{k+1,n-k+1} = \text{Const}(1)$ and $G_{k,n-k+2} = \text{Const}(0)$. If ($\ell_n$ is $\mathbf{x}_n$ and $\xi(\mathbf{x}_n) = 1$) or (ℓ_n is $\neg\mathbf{x}_n$ and $\xi(\mathbf{x}_n) = 0$), then ℓ_n holds under the valuation ξ and the edge starting from the root of the BDD $G_{k,n-k+1}$ with label $\xi(\mathbf{x}_n)$ points to the root of $G_{k+1,n-k+1}$, i.e., the 1-leaf. If (ℓ_n is $\mathbf{x}_n$ and $\xi(\mathbf{x}_n) = 0$) or (ℓ_n is $\neg\mathbf{x}_n$ and $\xi(\mathbf{x}_n) = 1$), then ℓ_n does not hold under the valuation ξ and the edge starting from the root of BDD $G_{k,n-k+1}$ with label $\xi(\mathbf{x}_n)$ points to the root of $G_{k,n-k+2}$, i.e., the 0-leaf. The claim follows.

Induction step $1 \leq i \leq k$ and $1 \leq j \leq n-k+1$ such that $i \neq k$ or/and $j \neq n-k+1$. The BDD $G_{i,j}$ is built via $\text{ITE}(\mathbf{x}_{i+j-1}, G_{i+1,j}, G_{i,j+1})$ if ℓ_{i+j-1} is $\mathbf{x}_{i+j-1}$ or $\text{ITE}(\mathbf{x}_{i+j-1}, G_{i,j+1}, G_{i+1,j})$ if ℓ_{i+j-1} is $\neg\mathbf{x}_{i+j-1}$.

- If (ℓ_{i+j-1} is $\mathbf{x}_{i+j-1}$ and $\xi(\mathbf{x}_{i+j-1}) = 1$) or ($\ell_{i+j-1}$ is $\neg\mathbf{x}_{i+j-1}$ and $\xi(\mathbf{x}_{i+j-1}) = 0$), then the literal ℓ_{i+j-1} holds under the valuation ξ and the edge from the root of the BDD $G_{i,j}$ with label $\xi(\mathbf{x}_{i+j-1})$ points to the root of the BDD $G_{i+1,j}$.

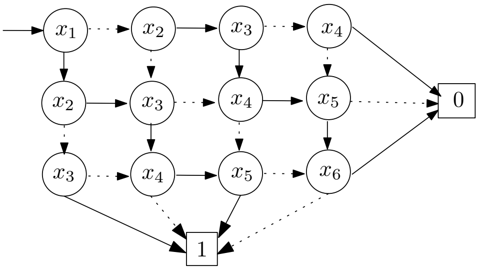


Fig. 6. The BDD of the cardinality constraint $x_1 + \neg x_2 + x_3 + \neg x_4 + x_5 + \neg x_6 \geq 3$

  - If $i = k$, then $j < n - k + 1$, $G_{i+1,j}$ is $\textsc{Const}(1)$ and there is at least one $(k - i + 1 = 1)$ positive literal (e.g., ℓ_{i+j-1}) among the literals $\{\ell_{i+j-1}, \cdots, \ell_n\}$. Since the root of $\textsc{Const}(1)$ is the 1-leaf, the result follows.
  - If $i < k$, by applying the induction hypothesis to $i + 1$, we get that if there are at least (resp. less than) $k - i$ positive literals among the literals $\{\ell_{i+j}, \cdots, \ell_n\}$ under the valuation ξ, then the path starting from the root of the BDD $G_{i+1,j}$ and following the valuation ξ ends at the 1-leaf (resp. 0-leaf). Since the literal ℓ_{i+j-1} holds under the valuation ξ, we get that if there are at least (resp. less than) $k - i$ positive literals among the literals $\{\ell_{i+j}, \cdots, \ell_n\}$ under the valuation ξ, then there are at least (resp. less than) $k - i + 1$ positive literals among the literals $\{\ell_{i+j-1}, \cdots, \ell_n\}$ under the valuation ξ and the path starting from the root of the BDD $G_{i,j}$ and following the valuation ξ ends at the 1-leaf (resp. 0-leaf).
- If (ℓ_{i+j-1} is $\mathbf{x}_{i+j-1}$ and $\xi(\mathbf{x}_{i+j-1}) = 0$) or ($\ell_{i+j-1}$ is $\neg\mathbf{x}_{i+j-1}$ and $\xi(\mathbf{x}_{i+j-1}) = 1$), then the literal ℓ_{i+j-1} does not hold under the valuation ξ and the edge from the root of the BDD $G_{i,j}$ with label $\xi(\mathbf{x}_{i+j-1})$ points to the root of the BDD $G_{i,j+1}$.
  - If $j = n - k + 1$, then $i < k$, $G_{i,j+1}$ is $\textsc{Const}(0)$ and there are less than $(k - i + 1)$ positive literal among the literals $\{\ell_{i+j-1}, \cdots, \ell_n\}$. Since the root of $\textsc{Const}(0)$ is the 0-leaf, the result follows.
  - If $j < n - k + 1$, by applying the induction hypothesis to $j + 1$, we get that if there are at least (resp. less than) $k - i + 1$ positive literals among the literals $\{\ell_{i+j}, \cdots, \ell_n\}$ under the valuation ξ, then the path starting from the root of the BDD $G_{i,j+1}$ and following the valuation ξ ends at the 1-leaf (resp. 0-leaf). Thus, the result follows from the facts that the literal ℓ_{i+j-1} does not hold under the valuation ξ and the edge from the root of the BDD $G_{i,j}$ with label $\xi(\mathbf{x}_{i+j-1})$ points to the root of the BDD $G_{i,j+1}$.

□

Example 3.2. Consider the cardinality constraint $x_1 + \neg x_2 + x_3 + \neg x_4 + x_5 + \neg x_6 \geq 3$, by Algorithm 1, we obtain the BDD shown in Figure 6.

Compared to the prior work [10, 67] which transforms general arithmetic constraints into BDDs, we devise a dedicated BDD encoding algorithm for the cardinality constraints without applying the reduction, thus our algorithm is more efficient. An alternative approach, called "DP-based" algorithm [27], recursively constructs the desired BDD from a cardinality constraint via dynamic programming. Although the DP-based algorithm shares the similar idea to ours during the BDD construction (i.e., counting the number of positive literals), our "graph-based" algorithm significantly outperforms, as shown in Section 6.1.1.

3.3 Region2BDD: Input Regions to BDDs

In this paper, we consider the following two types of input regions.

- *Input region based on the Hamming distance.* For an input $\mathbf{u} \in \mathbb{B}_{\pm 1}^{n_1}$ and an integer $r \geq 0$, let $R(\mathbf{u}, r) := \{\mathbf{x} \in \mathbb{B}_{\pm 1}^{n_1} \mid \mathsf{HD}(\mathbf{x}, \mathbf{u}) \leq r\}$, where $\mathsf{HD}(\mathbf{x}, \mathbf{u})$ denotes the Hamming distance between $\mathbf{x}$ and $\mathbf{u}$. Intuitively, $R(\mathbf{u}, r)$ includes the input vectors which differ from $\mathbf{u}$ by at most r positions.
- *Input region with fixed indices.* For an input $\mathbf{u} \in \mathbb{B}_{\pm 1}^{n_1}$ and a set of indices $I \subseteq [n_1]$, let $R(\mathbf{u}, I) := \{\mathbf{x} \in \mathbb{B}_{\pm 1}^{n_1} \mid \forall i \in [n_1] \setminus I.\ \mathbf{u}_i = \mathbf{x}_i\}$. Intuitively, $R(\mathbf{u}, I)$ includes the input vectors which differ from $\mathbf{u}$ only at the indices in I.

Note that both $R(\mathbf{u}, n_1)$ and $R(\mathbf{u}, [n_1])$ denote the full input space $\mathbb{B}_{\pm 1}^{n_1}$.

Recall that each input sample is an element from the space $\mathbb{B}_{\pm 1}^{n_1}$, namely, value at each index of an input sample is +1 or -1. To represent the region R by a BDD, we transform each value ±1 into a Boolean value 1/0. To this end, for each input $\mathbf{u} \in \mathbb{B}_{\pm 1}^{n_1}$, we create a new sample $\mathbf{u}^{(b)} \in \mathbb{B}^{n_1}$ such that for every $i \in [n_1]$, $\mathbf{u}_i = 2\mathbf{u}_i^{(b)} - 1$. Therefore, $R(\mathbf{u}, r)$ and $R(\mathbf{u}, I)$ will be represented by $R(\mathbf{u}^{(b)}, r)$ and $R(\mathbf{u}^{(b)}, I)$, respectively. Hereafter, for ease of presentation, $R(\mathbf{u}^{(b)}, r)$ and $R(\mathbf{u}^{(b)}, I)$ are denoted by $R(\mathbf{u}, r)$ and $R(\mathbf{u}, I)$. The transformation functions t_i^{lin}, t_i^{bn}, t_i^{bin} and t_{d+1}^{am} of the LIN, BN, BIN, and ARGMAX layers (cf. Table 1) will be handled accordingly. Note that for convenience, vectors over the Boolean domain $\mathbb{B}$ may be directly given by $\mathbf{u}$ or $\mathbf{x}$ instead of $\mathbf{u}^{(b)}$ or $\mathbf{x}^{(b)}$ when it is clear from the context.

Region encoding under Hamming distance. Given an input $\mathbf{u} \in \mathbb{B}^{n_1}$ and an integer $r \geq 0$, the region $R(\mathbf{u}, r) = \{\mathbf{x} \in \mathbb{B}^{n_1} \mid \mathsf{HD}(\mathbf{x}, \mathbf{u}) \leq r\}$ can be expressed by a cardinality constraint $\sum_{j=1}^{n_1} \ell_j \leq r$ (which is equivalent to $\sum_{j=1}^{n_1} \neg\ell_j \geq n_1 - r$), where for every $j \in [n_1]$, $\ell_j = \mathbf{x}_j$ if $\mathbf{u}_j = 0$, otherwise $\ell_j = \neg\mathbf{x}_j$. For instance, consider $\mathbf{u} = (1, 1, 1, 0, 0)$ and $r = 2$, we have

$$\mathsf{HD}(\mathbf{u}, \mathbf{x}) = 1 \oplus \mathbf{x}_1 + 1 \oplus \mathbf{x}_2 + 1 \oplus \mathbf{x}_3 + 0 \oplus \mathbf{x}_4 + 0 \oplus \mathbf{x}_5 = \neg\mathbf{x}_1 + \neg\mathbf{x}_2 + \neg\mathbf{x}_3 + \mathbf{x}_4 + \mathbf{x}_5.$$

Thus, $R((1, 1, 1, 0, 0), 2)$ can be expressed by the cardinality constraint $\neg\mathbf{x}_1 + \neg\mathbf{x}_2 + \neg\mathbf{x}_3 + \mathbf{x}_4 + \mathbf{x}_5 \leq 2$, or equivalently $\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \neg\mathbf{x}_4 + \neg\mathbf{x}_5 \geq 3$.

By Algorithm 1, the cardinality constraint of $R(\mathbf{u}, r)$ can be encoded by a BDD $G_{\mathbf{u},r}^{in}$ such that $\mathcal{L}(G_{\mathbf{u},r}^{in}) = R(\mathbf{u}, r)$. Following Lemma 3.1, we get that:

Lemma 3.3. *For an input region R given by an input $\mathbf{u} \in \mathbb{B}^{n_1}$ and an integer $r \geq 0$, a BDD $G_{\mathbf{u},r}^{in}$ with $O(r \cdot (n_1 - r))$ nodes can be computed in $O(r \cdot (n_1 - r))$ time such that $\mathcal{L}(G_{\mathbf{u},r}^{in}) = R(\mathbf{u}, r)$.*

Proof. We first prove that the input region $R(\mathbf{u}, r) = \{\mathbf{x} \in \mathbb{B}^{n_1} \mid \mathsf{HD}(\mathbf{u}, \mathbf{x}) \leq r\}$ given by an input $\mathbf{u} \in \mathbb{B}^{n_1}$ and a Hamming distance r is equal to the set of all the solutions of the cardinality constraint $\sum_{j=1}^{n_1} \ell_j \leq r$.

Consider an input $\mathbf{u}' \in \mathbb{B}^{n_1}$. For every literal ℓ_j, since $\ell_j = \mathbf{x}_j$ if $\mathbf{u}_j = 0$ and $\ell_j = \neg\mathbf{x}_j$ otherwise, the literal ℓ_j is positive iff $\mathbf{u}'_j$ is different from $\mathbf{u}_j$. Since a valuation ξ is a solution of $\sum_{j=1}^{n_1} \ell_j \leq r$ iff there are at most r positive literals under the valuation ξ, and $\mathbf{u}' \in R(\mathbf{u}, r)$ iff $\mathbf{u}$ and $\mathbf{u}'$ differ at most r positions, we have that $\mathbf{u}' \in R(\mathbf{u}, r)$ iff ξ is a solution of $\sum_{j=1}^{n_1} \ell_j \leq r$, where $\xi(\mathbf{x}_j) = \mathbf{u}'_j$ for every $j \in [n_1]$.

By Lemma 3.1, a BDD $G_{\mathbf{u},r}^{in}$ with $O(r \cdot (n_1 - r))$ nodes can be constructed in $O(r \cdot (n_1 - r))$ time such that $\mathbf{u}' \in \mathcal{L}(G_{\mathbf{u},r}^{in})$ iff ξ is a solution of $\sum_{j=1}^{n_1} \neg\ell_j \geq n_1 - r$, where $\xi(\mathbf{x}_j) = \mathbf{u}'_j$ for every $j \in [n_1]$. The result then follows from that $\sum_{j=1}^{n_1} \ell_j \leq r$ is equivalent to $\sum_{j=1}^{n_1} \neg\ell_j \geq n_1 - r$ by replacing each literal ℓ_j with $1 - \neg\ell_j$. □

Region encoding under fixed indices. Given an input $\mathbf{u} \in \mathbb{B}^{n_1}$ and a set of indices $I \subseteq [n_1]$, the region $R(\mathbf{u}, I) = \{\mathbf{x} \in \mathbb{B}^{n_1} \mid \forall i \in [n_1] \setminus I.\ \mathbf{u}_i = \mathbf{x}_i\}$ can be represented by the BDD $G_{\mathbf{u},I}^{in}$, where

$$G_{\mathbf{u},I}^{in} \triangleq \textsc{And}_{i \in [n_1] \setminus I}\Big((\mathbf{u}_i == 1) ? \textsc{New}(\mathbf{x}_i) : \textsc{Not}(\textsc{New}(\mathbf{x}_i))\Big).$$

Intuitively, $G^{in}_{\mathbf{u},I}$ states that the value at the index $i \in [n_1] \setminus I$ should be the same as the one in $\mathbf{u}$ while the value at the index $i \in I$ is unrestricted. For instance, consider $\mathbf{u} = (1, 0, 0, 0)$ and $I = \{3, 4\}$, we have:

$$R((1, 0, 0, 0), \{3, 4\}) = \{(1, 0, 0, 0), (1, 0, 0, 1), (1, 0, 1, 0), (1, 0, 1, 1)\} = \mathbf{x}_1 \wedge \neg\mathbf{x}_2.$$

Lemma 3.4. *For an input region R given by an input $\mathbf{u} \in \mathbb{B}^{n_1}$ and a set of indices $I \subseteq [n_1]$, a BDD $G^{in}_{\mathbf{u},I}$ with $O(n_1 - |I|)$ nodes can be computed in $O(n_1)$ time such that $\mathcal{L}(G^{in}_{\mathbf{u},I}) = R(\mathbf{u}, I)$.*

Proof. Consider an input $\mathbf{u}' \in \mathbb{B}^{n_1}$. $\mathbf{u}' \in R(\mathbf{u}, I)$ iff $\mathbf{u}_i = \mathbf{u}'_i$ for all $i \in [n_1] \setminus I$. By the definition of $G^{in}_{\mathbf{u},I}$, $\mathbf{u}' \in \mathcal{L}(G^{in}_{\mathbf{u},I})$ iff $\mathbf{u}_i = \mathbf{u}'_i$ for all $i \in [n_1] \setminus I$. Therefore $\mathcal{L}(G^{in}_{\mathbf{u},I}) = R(\mathbf{u}, I)$.

By the definition of $G^{in}_{\mathbf{u},I}$, $G^{in}_{\mathbf{u},I}$ can be built from by applying $n_1 - |I|$ New-operations, $n_1 - |I|$ And-operations, and at most $n_1 - |I|$ Not-operations (cf. Table 2 for BDD operations). The New-operation and Not-operation can be done in $O(1)$ time. To achieve $O(1)$ time for each And-operation, the variables are processed according to the variable ordering $\mathbf{x}_1 < \cdots < \mathbf{x}_{n_1}$, namely, $\mathbf{x}_{i+1}$ is processed earlier than $\mathbf{x}_i$. Therefore, the BDD $G^{in}_{\mathbf{u},I}$ can be computed in $O(n_1)$ time. Finally, $G^{in}_{\mathbf{u},I}$ has $n_1 - |I| + 2$ nodes. This is because, for every $i \in [n_1]$, $\mathbf{x}_i$ along the path to the 1-leaf in $G^{in}_{\mathbf{u},I}$ can only be $\mathbf{u}_i$ if $i \notin I$ or any value if $i \in I$. □

3.4 BNN2CC: BNNs to Cardinality Constraints

As mentioned before, to encode the BNN $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$ as BDDs $(G^{out}_i)_{i \in [s]}$, we transform the BNN $\mathcal{N}$ into cardinality constraints from which the desired BDDs $(G^{out}_i)_{i \in [s]}$ are constructed. To this end, we first transform each internal block $t_i : \mathbb{B}^{n_i}_{\pm 1} \longrightarrow \mathbb{B}^{n_{i+1}}_{\pm 1}$ into n_{i+1} cardinality constraints, each of which corresponds to one entry of the output vector of the internal block t_i. Then we transform the output block $t_{d+1} : \mathbb{B}^{n_{d+1}}_{\pm 1} \longrightarrow \mathbb{B}^s$ into $s(s-1)$ cardinality constraints, where one output class yields $(s-1)$ cardinality constraints.

For each vector-valued function t, we denote by $t_{\downarrow j}$ the (scalar-valued) function returning the j-th entry of the output vector of t. Specifically, we use $t^{bin}_{i\downarrow j}$ and $t^{bn}_{i\downarrow j}$ to denote the element-wise counterparts of the Binarization and Batch Normalization functions respectively. Namely, $t^{bin}_{i\downarrow j}$ (resp. $t^{bn}_{i\downarrow j}$) takes the j-th entry of the input vector of t^{bin}_i (resp. t^{bn}_i) as input and returns the j-th entry of the output vector of t^{bin}_i (resp. t^{bn}_i).

Transformation for internal blocks. Consider the internal block $t_i : \mathbb{B}^{n_i}_{\pm 1} \longrightarrow \mathbb{B}^{n_{i+1}}_{\pm 1}$ for $i \in [d]$. Recall that for every $j \in [n_{i+1}]$ and $\mathbf{x} \in \mathbb{B}^{n_i}_{\pm 1}$, $t_{i\downarrow j}(\mathbf{x}) = t^{bin}_{i\downarrow j}(t^{bn}_{i\downarrow j}(\langle \mathbf{x}, \mathbf{W}_{:,j}\rangle + \mathbf{b}_j))$, and each input $\mathbf{x} \in \mathbb{B}^{n_1}_{\pm 1}$ can be replaced by $2\mathbf{x}^{(b)} - \mathbf{1} \in \mathbb{B}^{n_1}$ (cf. Section 3.3), where $\mathbf{1}$ denotes the vector of 1's with width n_i. To be consistent, we reformulate the function $t_{i\downarrow j} : \mathbb{B}^{n_i}_{\pm 1} \longrightarrow \mathbb{B}_{\pm 1}$ as the function $t^{(b)}_{i\downarrow j} : \mathbb{B}^{n_i} \longrightarrow \mathbb{B}$ such that for every $\mathbf{x}^{(b)} \in \mathbb{B}^{n_i}$:

$$t^{(b)}_{i\downarrow j}(\mathbf{x}^{(b)}) = \frac{1}{2} \times (t^{bin}_{i\downarrow j}(t^{bn}_{i\downarrow j}(\langle 2\mathbf{x}^{(b)} - \mathbf{1}, \mathbf{W}_{:,j}\rangle + \mathbf{b}_j)) + 1).$$

Intuitively, an input $\mathbf{x} \in \mathbb{B}^{n_i}_{\pm 1}$ of the function $t_{i\downarrow j}$ is transformed into the input $\mathbf{x}^{(b)} = \frac{1}{2} \times (\mathbf{x}+1) \in \mathbb{B}^{n_i}$ of the function $t^{(b)}_{i\downarrow j}$, where the output $t_{i\downarrow j}(\mathbf{x}) \in \mathbb{B}_{\pm 1}$ becomes $t^{(b)}_{i\downarrow j}(\mathbf{x}^{(b)}) = \frac{1}{2}(t_{i\downarrow j}(\mathbf{x}) + 1) \in \mathbb{B}$. Note that for convenience, vectors over the Boolean domain $\mathbb{B}$ may be directly given by $\mathbf{u}$ or $\mathbf{x}$ instead of $\mathbf{u}^{(b)}$ or $\mathbf{x}^{(b)}$ in the following part when it is clear from the context.

To encode the function $t^{(b)}_{i\downarrow j}$ as a BDD, we show how to encode the function $t^{(b)}_{i\downarrow j}$ as an equivalent cardinality constraint via a series of equivalent transformations, based on which the BDD is built by applying Algorithm 1. We first get ride of the functions $t^{bin}_{i\downarrow j}$ and $t^{bn}_{i\downarrow j}$ according to their definitions, namely, for every $\mathbf{x} \in \mathbb{B}^{n_i}$, the function $t^{(b)}_{i\downarrow j}$ is reformulated as

$$\begin{aligned} t^{(b)}_{i\downarrow j}(\mathbf{x}) &= \tfrac{1}{2} \times (t^{bin}_{i\downarrow j}(t^{bn}_{i\downarrow j}(\langle 2\mathbf{x} - \mathbf{1}, \mathbf{W}_{:,j}\rangle + \mathbf{b}_j)) + 1) \\ &= \tfrac{1}{2} \times (t^{bin}_{i\downarrow j}(t^{bn}_{i\downarrow j}(\textstyle\sum_{k=1}^{n_i}(2\mathbf{x}_k - 1) \cdot \mathbf{W}_{k,j} + \mathbf{b}_j)) + 1) \\ &= \tfrac{1}{2} \times (t^{bin}_{i\downarrow j}(\alpha_j \cdot (\tfrac{\sum_{k=1}^{n_i}(2\mathbf{x}_k-1)\cdot\mathbf{W}_{k,j}+\mathbf{b}_j-\mu_j}{\sigma_j}) + \gamma_j) + 1) \\ &= \begin{cases} 1, & \text{if } \alpha_j \cdot (\frac{\sum_{k=1}^{n_i}(2\mathbf{x}_k-1)\cdot\mathbf{W}_{k,j}+\mathbf{b}_j-\mu_j}{\sigma_j}) + \gamma_j \geq 0; \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

By the above reformulation, the function $t^{(b)}_{i\downarrow j}$ now can be represented by the following constraint

$$t^{(b)}_{i\downarrow j}(\mathbf{x}) = 1 \text{ iff } \alpha_j \cdot (\frac{\sum_{k=1}^{n_i}(2\mathbf{x}_k - 1) \cdot \mathbf{W}_{k,j} + \mathbf{b}_j - \mu_j}{\sigma_j}) + \gamma_j \geq 0.$$

To convert the constraint $\alpha_j \cdot (\frac{\sum_{k=1}^{n_i}(2\mathbf{x}_k-1)\cdot\mathbf{W}_{k,j}+\mathbf{b}_j-\mu_j}{\sigma_j}) + \gamma_j \geq 0$ to an equivalent cardinality constraint, we consider different cases of α_j, i.e., $\alpha_j > 0$, $\alpha_j < 0$ and $\alpha_j = 0$, based on which α_j can be eliminated from the constraint.

- Case $\alpha_j > 0$. The constraint $\alpha_j \cdot (\frac{\sum_{k=1}^{n_i}(2\mathbf{x}_k-1)\cdot\mathbf{W}_{k,j}+\mathbf{b}_j-\mu_j}{\sigma_j}) + \gamma_j \geq 0$ can be rewritten as

$$\sum_{k=1}^{n_i} \mathbf{x}_k \cdot \mathbf{W}_{k,j} \geq \frac{1}{2} \cdot (\sum_{k=1}^{n_i} \mathbf{W}_{k,j} + \mu_j - \mathbf{b}_j - \frac{\gamma_j \cdot \sigma_j}{\alpha_j}). \tag{1}$$

By partitioning the set $[n_i]$ of indices into two subsets $\mathbf{W}^+_{:,j}$ and $\mathbf{W}^-_{:,j}$, where

$$\mathbf{W}^+_{:,j} = \{k \in [n_i] \mid \mathbf{W}_{k,j} = +1\} \text{ and } \mathbf{W}^-_{:,j} = \{k \in [n_i] \mid \mathbf{W}_{k,j} = -1\}.$$

Intuitively, $\mathbf{W}^+_{:,j}$ contains the indices $k \in [n_i]$ such that the weight $\mathbf{W}_{k,j}$ is +1, while $\mathbf{W}^-_{:,j}$ contains the indices $k \in [n_i]$ such that the weight $\mathbf{W}_{k,j}$ is -1. Using $\mathbf{W}^+_{:,j}$ and $\mathbf{W}^-_{:,j}$, the expression $\sum_{k=1}^{n_i} \mathbf{x}_k \cdot \mathbf{W}_{k,j}$ can be written as $\sum_{k\in\mathbf{W}^+_{:,j}} \mathbf{x}_k - \sum_{k\in\mathbf{W}^-_{:,j}} \mathbf{x}_k$ and the expression $\sum_{k=1}^{n_i} \mathbf{W}_{k,j}$ can be written as $|\mathbf{W}^+_{k,j}| - |\mathbf{W}^-_{k,j}|$. Therefore, the constraint in Equation (1) can be written as the constraint

$$\sum_{k\in\mathbf{W}^+_{:,j}} \mathbf{x}_k - \sum_{k\in\mathbf{W}^-_{:,j}} \mathbf{x}_k \geq \frac{1}{2} \cdot (|\mathbf{W}^+_{k,j}| - |\mathbf{W}^-_{k,j}| + \mu_j - \mathbf{b}_j - \frac{\gamma_j \cdot \sigma_j}{\alpha_j}). \tag{2}$$

After replacing $-\mathbf{x}_k$ by $\neg\mathbf{x}_k - 1$ for every $k \in \mathbf{W}^-_{:,j}$, the constraint in Equation (2) can be reformulated into the constraint

$$\sum_{k\in\mathbf{W}^+_{:,j}} \mathbf{x}_k + \sum_{k\in\mathbf{W}^-_{:,j}} \neg\mathbf{x}_k \geq \frac{1}{2} \cdot (|\mathbf{W}^+_{k,j}| - |\mathbf{W}^-_{k,j}| + \mu_j - \mathbf{b}_j - \frac{\gamma_j \cdot \sigma_j}{\alpha_j}) + |\mathbf{W}^-_{k,j}|,$$

This transformation eliminates the subtraction operation $-$ from $\sum_{k\in\mathbf{W}^+_{:,j}} \mathbf{x}_k - \sum_{k\in\mathbf{W}^-_{:,j}} \mathbf{x}_k$ using the negation operation $\neg$. The resulting constraint now can be further rewritten as the following cardinality constraint:

$$\sum_{k\in\mathbf{W}^+_{:,j}} \mathbf{x}_k + \sum_{k\in\mathbf{W}^-_{:,j}} \neg\mathbf{x}_k \geq \lceil \frac{1}{2} \cdot (n_i + \mu_j - \mathbf{b}_j - \frac{\gamma_j \cdot \sigma_j}{\alpha_j}) \rceil.$$

Remark that $\mathbf{W}^+_{:,j}$, $\mathbf{W}^-_{:,j}$ and $\lceil \frac{1}{2} \cdot (n_i + \mu_j - \mathbf{b}_j - \frac{\gamma_j\cdot\sigma_j}{\alpha_j}) \rceil$ are independent of the input $\mathbf{x}$ and thus can be computed during the equivalent transformations.

- Case $\alpha_j < 0$. The constraint $\alpha_j \cdot (\frac{\sum_{k=1}^{n_i}(2\mathbf{x}_k-1)\cdot\mathbf{W}_{k,j}+\mathbf{b}_j-\mu_j}{\sigma_j}) + \gamma_j \geq 0$ can be rewritten as
$$\sum_{k=1}^{n_i}\mathbf{x}_k \cdot \mathbf{W}_{k,j} \leq \frac{1}{2} \cdot (\sum_{k=1}^{n_i}\mathbf{W}_{k,j} + \mu_j - \mathbf{b}_j - \frac{\gamma_j \cdot \sigma_j}{\alpha_j}),$$
which can further be reformulated into the following constraint using $\mathbf{W}^+_{:,j}$ and $\mathbf{W}^-_{:,j}$,
$$\sum_{k\in\mathbf{W}^-_{:,j}}\mathbf{x}_k - \sum_{k\in\mathbf{W}^+_{:,j}}\mathbf{x}_k \geq \frac{1}{2} \cdot (|\mathbf{W}^-_{k,j}| - |\mathbf{W}^+_{k,j}| - \mu_j + \mathbf{b}_j + \frac{\gamma_j \cdot \sigma_j}{\alpha_j}). \tag{3}$$
After replacing $-\mathbf{x}_k$ by $\neg\mathbf{x}_k - 1$ for every $k \in \mathbf{W}^+_{:,j}$, the subtraction operation $-$ is eliminated from $\sum_{k\in\mathbf{W}^-_{:,j}}\mathbf{x}_k - \sum_{k\in\mathbf{W}^+_{:,j}}\mathbf{x}_k$ using the negation operation $\neg$ and the constraint in Equation (3) is reformulated into the constraint
$$\sum_{k\in\mathbf{W}^-_{:,j}}\mathbf{x}_k + \sum_{k\in\mathbf{W}^+_{:,j}}\neg\mathbf{x}_k \geq \frac{1}{2} \cdot (|\mathbf{W}^-_{k,j}| - |\mathbf{W}^+_{k,j}| - \mu_j + \mathbf{b}_j + \frac{\gamma_j \cdot \sigma_j}{\alpha_j}) + |\mathbf{W}^+_{k,j}|,$$
which is rewritten as the cardinality constraint
$$\sum_{k\in\mathbf{W}^-_{:,j}}\mathbf{x}_k + \sum_{k\in\mathbf{W}^+_{:,j}}\neg\mathbf{x}_k \geq \lceil\frac{1}{2} \cdot (n_i - \mu_j + \mathbf{b}_j + \frac{\gamma_j \cdot \sigma_j}{\alpha_j})\rceil.$$
- Case $\alpha_j = 0$. The constraint $\alpha_j \cdot (\frac{\sum_{k=1}^{n_i}(2\mathbf{x}_k-1)\cdot\mathbf{W}_{k,j}+\mathbf{b}_j-\mu_j}{\sigma_j}) + \gamma_j \geq 0$ becomes $\gamma_j \geq 0$.

Based on the above equivalent transformations, we define the cardinality constraint $C_{i,j}$ as follows:
$$C_{i,j} \triangleq \begin{cases} \sum_{k=1}^{n_i}\ell_k \geq \lceil\frac{1}{2}\cdot(n_i + \mu_j - \mathbf{b}_j - \frac{\gamma_j\cdot\sigma_j}{\alpha_j})\rceil, & \text{if } \alpha_j > 0; \\ 1, & \text{if } \alpha_j = 0 \wedge \gamma_j \geq 0; \\ 0, & \text{if } \alpha_j = 0 \wedge \gamma_j < 0; \\ \sum_{k=1}^{n_i}\neg\ell_k \geq \lceil\frac{1}{2}\cdot(n_i - \mu_j + \mathbf{b}_j + \frac{\gamma_j\cdot\sigma_j}{\alpha_j})\rceil, & \text{if } \alpha_j < 0; \end{cases}$$
where for every $k \in [n_i]$,
$$\ell_k \triangleq \begin{cases} \mathbf{x}_k, & \text{if } \mathbf{W}_{k,j} = +1; \\ \neg\mathbf{x}_k, & \text{if } \mathbf{W}_{k,j} = -1. \end{cases}$$

Proposition 3.5. $t^{(b)}_{i\downarrow j} \Leftrightarrow C_{i,j}$.

Hereafter, for each internal block $t_i : \mathbb{B}^{n_i}_{\pm1} \to \mathbb{B}^{n_{i+1}}_{\pm1}$ where $i \in [d]$, we denote by BNN2CC(t_i) the cardinality constraints $\{C_{i,1}, \cdots, C_{i,n_{i+1}}\}$.

Transformation for the output block. For the output block $t_{d+1} : \mathbb{B}^{n_{d+1}}_{\pm1} \to \mathbb{B}^s$, similar to the transformation for internal blocks, we first transform the function t_{d+1} into an equivalent cardinality constraint based on which a BDD can be built by applying Algorithm 1. Since $t_{d+1} = t^{am}_{d+1} \circ t^{lin}_{d+1}$, then for every $j \in [s]$, we can reformulate $t_{d+1\downarrow j} : \mathbb{B}^{n_{d+1}}_{\pm1} \to \mathbb{B}$ as the function $t^{(b)}_{d+1\downarrow j} : \mathbb{B}^{n_{d+1}} \to \mathbb{B}$ such that for every $\mathbf{x} \in \mathbb{B}^{n_{d+1}}$,
$$t^{(b)}_{d+1\downarrow j}(\mathbf{x}) = t_{d+1\downarrow j}(2\mathbf{x} - \mathbf{1}) = t^{am}_{d+1\downarrow j}(t^{lin}_{d+1}(2\mathbf{x} - \mathbf{1})).$$

According to the definition of the function t^{am}_{d+1}, $t^{(b)}_{d+1\downarrow j}(\mathbf{x}) = 1$ iff for every $j' \in [s]$ such that $j \neq j'$, one of the following conditions holds:

- $j' < j$ and $t^{lin}_{d+1\downarrow j}(2\mathbf{x} - \mathbf{1}) > t^{lin}_{d+1\downarrow j'}(2\mathbf{x} - \mathbf{1})$;
- $j' > j$ and $t^{lin}_{d+1\downarrow j}(2\mathbf{x} - \mathbf{1}) \geq t^{lin}_{d+1\downarrow j'}(2\mathbf{x} - \mathbf{1})$.

By getting ride of the function $t^{lin}_{d+1\downarrow j'}$ according to its definition, the function $t^{(b)}_{d+1\downarrow j}$ can be encoded using the following constraint:

$$t^{(b)}_{d+1\downarrow j}(\mathbf{x}) = 1 \text{ iff } \left(\begin{array}{c} \forall j' \in [j-1].\ \sum_{k=1}^{n_{d+1}} (2\mathbf{x}_k - 1) \cdot (\mathbf{W}_{k,j} - \mathbf{W}_{k,j'}) > \mathbf{b}_{j'} - \mathbf{b}_j \\ \text{and} \\ \forall j' \in \{j+1, \cdots, s\}.\ \sum_{k=1}^{n_{d+1}} (2\mathbf{x}_k - 1) \cdot (\mathbf{W}_{k,j} - \mathbf{W}_{k,j'}) \geq \mathbf{b}_{j'} - \mathbf{b}_j \end{array} \right),$$

where the latter holds iff

$$\left(\begin{array}{c} \forall j' \in [j-1].\ \sum_{k=1}^{n_{d+1}} \mathbf{x}_k \cdot \frac{\mathbf{W}_{k,j}-\mathbf{W}_{k,j'}}{2} > \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + \sum_{k=1}^{n_{d+1}} (\mathbf{W}_{k,j} - \mathbf{W}_{k,j'})) \\ \text{and} \\ \forall j' \in \{j+1, \cdots, s\}.\ \sum_{k=1}^{n_{d+1}} \mathbf{x}_k \cdot \frac{\mathbf{W}_{k,j}-\mathbf{W}_{k,j'}}{2} \geq \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + \sum_{k=1}^{n_{d+1}} (\mathbf{W}_{k,j} - \mathbf{W}_{k,j'})) \end{array} \right). \tag{4}$$

For each pair (j, j') of indices, we partition the set $[n_{d+1}]$ into three subsets

- $\mathtt{Pos}_{j,j'} = \{k \in [n_{d+1}] \mid \mathbf{W}_{k,j} - \mathbf{W}_{k,j'} = 2\}$,
- $\mathtt{Neg}_{j,j'} = \{k \in [n_{d+1}] \mid \mathbf{W}_{k,j} - \mathbf{W}_{k,j'} = -2\}$ and
- $\mathtt{Zero}_{j,j'} = \{k \in [n_{d+1}] \mid \mathbf{W}_{k,j} = \mathbf{W}_{k,j'}\}$.

Let $\sharp\mathtt{Pos}_{j,j'}$ and $\sharp\mathtt{Neg}_{j,j'}$ denote the size of the subsets $\mathtt{Pos}_{j,j'}$ and $\mathtt{Neg}_{j,j'}$, respectively. We will convert the constraints in Equation (4) to an equivalent cardinality constraint $C^{j,j'}_{d+1}$ by distinguishing the cases $j' \in [j-1]$ or $j' \in \{j+1, \cdots, s\}$.

- If $j' \in [j-1]$, using the subsets $\mathtt{Pos}_{j,j'}$ and $\mathtt{Neg}_{j,j'}$, the constraint $\sum_{k=1}^{n_{d+1}} \mathbf{x}_k \cdot \frac{\mathbf{W}_{k,j}-\mathbf{W}_{k,j'}}{2} > \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + \sum_{k=1}^{n_{d+1}} (\mathbf{W}_{k,j} - \mathbf{W}_{k,j'}))$ can be written as

$$\sum_{k \in \mathtt{Pos}_{j,j'}} \mathbf{x}_k - \sum_{k \in \mathtt{Neg}_{j,j'}} \mathbf{x}_k > \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} - 2\sharp\mathtt{Neg}_{j,j'}). \tag{5}$$

After replacing $-\mathbf{x}_k$ by $\neg\mathbf{x}_k - 1$ for every $k \in \mathtt{Neg}_{j,j'}$, the constraint in Equation (5) is reformulated as the constraint

$$\sum_{k \in \mathtt{Pos}_{j,j'}} \mathbf{x}_k + \sum_{k \in \mathtt{Neg}_{j,j'}} \neg\mathbf{x}_k > \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'}). \tag{6}$$

Now, we transform the strict inequlity > of the constraint in Equation (6) into inequlity ≥ by distinguishing if $\frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'})$ is an integer or not.

  – If $\frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'})$ is an integer, the constraint in Equation (6) is the same as

$$\sum_{k \in \mathtt{Pos}_{j,j'}} \mathbf{x}_k + \sum_{k \in \mathtt{Neg}_{j,j'}} \neg\mathbf{x}_k \geq \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'}) + 1.$$

  – If $\frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'})$ is not an integer, the constraint in Equation (6) is the same as

$$\sum_{k \in \mathtt{Pos}_{j,j'}} \mathbf{x}_k + \sum_{k \in \mathtt{Neg}_{j,j'}} \neg\mathbf{x}_k \geq \lceil \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'}) \rceil,$$

  as $\lceil \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'}) \rceil > \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathtt{Pos}_{j,j'} + 2\sharp\mathtt{Neg}_{j,j'})$.

Therefore, we define the cardinality constraint $C_{d+1}^{j,j'}$ for $j' \in [j-1]$ as follows

$$C_{d+1}^{j,j'} \triangleq \begin{cases} \sum_{k=1}^{n_{d+1}} \ell_k \geq \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} + 2\sharp\mathsf{Neg}_{j,j'}) + 1, & \text{if } \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} + 2\sharp\mathsf{Neg}_{j,j'}) \text{ is an integer;} \\ \sum_{k=1}^{n_{d+1}} \ell_k \geq \lceil \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} + 2\sharp\mathsf{Neg}_{j,j'}) \rceil, & \text{otherwise;} \end{cases}$$

where for every $k \in [n_{d+1}]$,

$$\ell_k \triangleq \begin{cases} \mathbf{x}_k, & \text{if } \mathbf{W}_{k,j} - \mathbf{W}_{k,j'} = +2; \\ \neg\mathbf{x}_k, & \text{if } \mathbf{W}_{k,j} - \mathbf{W}_{k,j'} = -2; \\ 0, & \text{if } \mathbf{W}_{k,j} - \mathbf{W}_{k,j'} = 0. \end{cases}$$

- If $j' \in \{j+1, \cdots, s\}$, using the subsets $\mathsf{Pos}_{j,j'}$ and $\mathsf{Neg}_{j,j'}$, the constraint $\sum_{k=1}^{n_{d+1}} \mathbf{x}_k \cdot \frac{\mathbf{W}_{k,j} - \mathbf{W}_{k,j'}}{2} \geq \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + \sum_{k=1}^{n_{d+1}}(\mathbf{W}_{k,j} - \mathbf{W}_{k,j'}))$ can be rewritten as

$$\sum_{k \in \mathsf{Pos}_{j,j'}} \mathbf{x}_k - \sum_{k \in \mathsf{Neg}_{j,j'}} \mathbf{x}_k \geq \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} - 2\sharp\mathsf{Neg}_{j,j'}). \tag{7}$$

After replacing $-\mathbf{x}_k$ by $\neg\mathbf{x}_k - 1$ for every $k \in \mathsf{Neg}_{j,j'}$, the constraint in Equation (7) is reformulated into the constraint

$$\sum_{k \in \mathsf{Pos}_{j,j'}} \mathbf{x}_k + \sum_{k \in \mathsf{Neg}_{j,j'}} \neg\mathbf{x}_k \geq \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} + 2\sharp\mathsf{Neg}_{j,j'}),$$

which can be rewritten as the following cardinality constraint

$$\sum_{k \in \mathsf{Pos}_{j,j'}} \mathbf{x}_k + \sum_{k \in \mathsf{Neg}_{j,j'}} \neg\mathbf{x}_k \geq \lceil \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} + 2\sharp\mathsf{Neg}_{j,j'}) \rceil.$$

Therefore, we define the cardinality constraint $C_{d+1}^{j,j'}$ for $j' \in \{j+1, \cdots, s\}$ as follows:

$$C_{d+1}^{j,j'} \triangleq \sum_{k=1}^{n_{d+1}} \ell_k \geq \lceil \frac{1}{4}(\mathbf{b}_{j'} - \mathbf{b}_j + 2\sharp\mathsf{Pos}_{j,j'} + 2\sharp\mathsf{Neg}_{j,j'}) \rceil.$$

Let C_{d+1}^{j} denote the constraint $\bigwedge_{j' \in [s].j' \neq j} C_{d+1}^{j,j'}$.

Proposition 3.6. $t_{d+1\downarrow j}^{(b)} \Leftrightarrow C_{d+1}^{j}$.

Hereafter, for each output class $j \in [s]$, we denote by $\text{BNN2CC}^j(t_{d+1})$ the set of cardinality constraints $\{C_{d+1}^{j,1}, \cdots C_{d+1}^{j,j-1}, C_{d+1}^{j,j+1}, \cdots, C_{d+1}^{j,s}\}$.

BNNs in cardinality constraint form. By applying the above transformation to all the blocks of the BNN $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$, we obtain its cardinality constraint form $\mathcal{N}^{(b)} = (t_1^{(b)}, \cdots, t_d^{(b)}, t_{d+1}^{(b)})$ such that

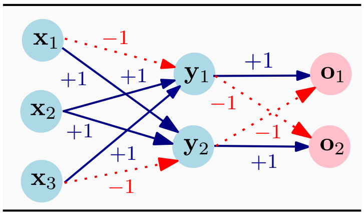
$$\mathcal{N}^{(b)} = t_{d+1}^{(b)} \circ t_d^{(b)} \circ \cdots \circ t_1^{(b)},$$

where

- for each $i \in [d]$, $t_i^{(b)}$ is (symbolically) represented by cardinality constraints $\text{BNN2CC}(t_i)$, and
- $t_{d+1}^{(b)}$ is represented by sets of cardinality constraints $(\text{BNN2CC}^1(t_{d+1}), \cdots, \text{BNN2CC}^s(t_{d+1}))$.

Theorem 3.7. *For every input* $\mathbf{u} \in \mathbb{B}_{\pm 1}^{n_1}$, $\mathcal{N}(\mathbf{u}) = \mathcal{N}^{(b)}(\mathbf{u}^{(b)})$, *where* $\mathbf{u} = 2\mathbf{u}^{(b)} - 1$.

$\mathbf{b}_1$	α_1	μ_1	γ_1	σ_1	$\mathbf{b}_2$
0.2	0.02	−0.5	0.02	2	−0.8
−0.5	−0.03	1.5	−0.03	3	0.6



$\mathcal{N}(\mathbf{x})$
$\mathbf{y}_1 = \text{sign}(0.01 \times (-\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + 2.7))$
$\mathbf{y}_2 = \text{sign}(0.01 \times (-\mathbf{x}_1 - \mathbf{x}_2 + \mathbf{x}_3 - 1))$
$\mathbf{o}_1 \Leftrightarrow \mathbf{y}_1 - \mathbf{y}_2 \geq 0.7$
$\mathbf{o}_2 \Leftrightarrow \mathbf{y}_1 - \mathbf{y}_2 < 0.7$

Cardinality Constraints
$\mathbf{y}_1^{(b)} \Leftrightarrow C_{1,1} = \neg\mathbf{x}_1^{(b)} + \mathbf{x}_2^{(b)} + \mathbf{x}_3^{(b)} \geq 1$
$\mathbf{y}_2^{(b)} \Leftrightarrow C_{1,2} = \neg\mathbf{x}_1^{(b)} + \neg\mathbf{x}_2^{(b)} + \mathbf{x}_3^{(b)} \geq 2$
$\mathbf{o}_1 \Leftrightarrow C_2^{1,2} = \mathbf{y}_1^{(b)} + \neg\mathbf{y}_2^{(b)} \geq 2$
$\mathbf{o}_2 \Leftrightarrow C_2^{2,1} = \neg\mathbf{y}_1^{(b)} + \mathbf{y}_2^{(b)} \geq 1$

Fig. 7. An illustrating example

Proof. We first show that $t_{i+1} \circ t_i(2\mathbf{x} - \mathbf{1}) = 2t_{i+1}^{(b)} \circ t_i^{(b)}(\mathbf{x}) - \mathbf{1}$ for each $i \in [d-1]$ and each input $x \in \mathbb{B}^{n_i}$. Recall that $t_{i\downarrow j}(2\mathbf{x} - \mathbf{1}) = 2t_{i\downarrow j}^{(b)}(\mathbf{x}) - 1$ for $j \in [n_{i+1}]$. Thus, we have $t_i(2\mathbf{x} - \mathbf{1}) = 2t_i^{(b)}(\mathbf{x}) - \mathbf{1}$, which implies that $t_{i+1}(t_i(2\mathbf{x} - \mathbf{1})) = t_{i+1}(2t_i^{(b)}(\mathbf{x}) - \mathbf{1})$. Let $\mathbf{y} = t_i^{(b)}(\mathbf{x}) \in \mathbb{B}^{n_{i+1}}$. Then $t_{i+1}(2t_i^{(b)}(\mathbf{x}) - \mathbf{1})$ is $t_{i+1}(2\mathbf{y} - \mathbf{1})$. Since $t_{i+1}(2\mathbf{y} - \mathbf{1}) = 2t_{i+1}^{(b)}(\mathbf{y}) - \mathbf{1}$, we get that $t_{i+1}(t_i(2\mathbf{x} - \mathbf{1})) = 2t_{i+1}^{(b)}(t_i^{(b)}(\mathbf{x})) - \mathbf{1}$.

Let $\mathcal{N}_{\leq d} = t_d \circ \cdots \circ t_1$ and $\mathcal{N}_{\leq d}^{(b)} = t_d^{(b)} \circ \cdots \circ t_1^{(b)}$. Then, for any input $\mathbf{u} \in \mathbb{B}_{\pm 1}^{n_1}$, we have: $\mathcal{N}_{\leq d}(\mathbf{u}) = \mathcal{N}_{\leq d}(2\mathbf{u}^{(b)} - 1) = 2\mathcal{N}_{\leq d}^{b}(\mathbf{u}^{(b)}) - \mathbf{1}$.

Recall that $t_{d+1\downarrow j}^{(b)}(\mathbf{x}) = t_{d+1\downarrow j}(2\mathbf{x} - \mathbf{1})$ for every $\mathbf{x} \in \mathbb{B}^{n_{d+1}}$, implying that $t_{d+1}^{(b)}(\mathbf{x}) = t_{d+1}(2\mathbf{x} - \mathbf{1})$. Thus, we get that $\mathcal{N}^{(b)}(\mathbf{u}^{(b)}) = t_{d+1}^{(b)}(\mathcal{N}_{\leq d}^{b}(\mathbf{u}^{(b)})) = t_{d+1}(2\mathcal{N}_{\leq d}^{b}(\mathbf{u}^{(b)}) - \mathbf{1}) = t_{d+1}(\mathcal{N}_{\leq d}(\mathbf{u})) = \mathcal{N}(\mathbf{u})$. □

Example 3.8. Consider the BNN $\mathcal{N} = (t_1, t_2)$ with one internal block t_1 and one output block t_2 as shown in Figure 7 (bottom left), where the entries of the weight matrix $\mathbf{W}$ are associated to the edges, and the other parameters are given in the top left table. The input-output relation of the blocks t_1 and t_2 are given in the top right table. The cardinality constraints are given in the bottom right table.

Consider an input $\mathbf{x} \in \mathbb{B}_{\pm 1}^3$, we have: $\mathbf{y}_1 = \text{sign}(0.01 \times (-\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + 2.7))$, namely, $\mathbf{y}_1 = +1$ iff $-\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + 2.7 \geq 0$. By replacing $\mathbf{x}_i$ with $2 \times \mathbf{x}_i^{(b)} - 1$ for $i \in [3]$ and $\mathbf{x}_1^{(b)}$ with $1 - \neg\mathbf{x}_1^{(b)}$, we get that $\mathbf{y}_1 = +1$ iff $-\mathbf{x}_1^{(b)} + \mathbf{x}_2^{(b)} + \mathbf{x}_3^{(b)} + 0.85 \geq 0$, which is equivalent to $\neg\mathbf{x}_1^{(b)} + \mathbf{x}_2^{(b)} + \mathbf{x}_3^{(b)} \geq 1$. Thus, we get that $\mathbf{y}_1^{(b)} \Leftrightarrow \neg\mathbf{x}_1^{(b)} + \mathbf{x}_2^{(b)} + \mathbf{x}_3^{(b)} \geq 1$. Similarly, we can deduce that $\mathbf{o}_1 = 1$ iff $\mathbf{y}_1 - \mathbf{y}_2 \geq 0.7$ and hence $\mathbf{o}_1 \Leftrightarrow \mathbf{y}_1^{(b)} - \mathbf{y}_2^{(b)} \geq 0.35$, which is equivalent to $\mathbf{y}_1^{(b)} + \neg\mathbf{y}_2^{(b)} \geq 2$.

3.5 BDD Model Builder

In general, the construction of the BDDs $(G_i^{out})_{i \in [s]}$ from the BNN $\mathcal{N}^{(b)}$ and the input region R is done iteratively throughout the blocks. Initially, the BDD for the first block is built, which can be seen as the input-output relation $t_1^{(b)}$ for the first internal block. In the i-th iteration, as the input-output relation $t_{i-1}^{(b)} \circ \cdots \circ t_1^{(b)}$ of the first $(i-1)$ internal blocks has been encoded into the BDD, we compose this BDD with the BDD for the block t_i which is built from its cardinality constraints $t_i^{(b)}$, resulting in the BDD for the first i internal blocks $t_i^{(b)} \circ t_{i-1}^{(b)} \circ \cdots \circ t_1^{(b)}$. Finally, we obtain the BDDs $(G_i^{out})_{i \in [s]}$ of the BNN $\mathcal{N}$, with respect to the input region R, where for each $i \in [s]$, the BDD G_i^{out} encodes the input-output relation $t_{d+1\downarrow i}^{(b)} \circ t_d^{(b)} \circ \cdots \circ t_1^{(b)}$.

Algorithm 2: BDD Construction of BNNs with Input Propagation

```
1  Procedure BNN2BDD(BNN : $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$, Region : $R(\mathbf{u}, \tau)$)
2      $G^{in} = G^{in}_{\mathbf{u},\tau}$ (cf. Section 3.3);
3      $\mathcal{N}^{(b)} = (t^{(b)}_1, \cdots, t^{(b)}_d, t^{(b)}_{d+1})$ (cf. Section 3.4);
4      for $(i = 1;\ i \leq d;\ i++)$ do
5          $G'$ =InterBlk2BDD($t^{(b)}_i, G^{in}$);
6          $G^{in}$ = Exists($G', \mathbf{x}^i$) ;                // $\mathbf{x}^i$ denote input variables of $t^{(b)}_i$
7          $G = (i == 1)\ ?\ G' :$ RelProd($G, G', \mathbf{x}^i$);
8      $(G_i)_{i \in [s]}$ = OutBlk2BDD($t^{(b)}_{d+1}, G^{in}$);
9      for $(i = 1;\ i \leq s;\ i++)$ do
10         $G^{out}_i$ = RelProd($G, G_i, \mathbf{x}^{d+1}$);
11     return $(G^{out}_i)_{i \in [s]}$
```

To improve the efficiency of BDD encoding, we propose two strategies, i.e., divide-and-conquer and input propagation.

3.5.1 Divide-and-Conquer Strategy. To encode the input-output relation of an internal block t_i into BDD from its cardinality constraints $t^{(b)}_i = \{C_{i,1}, \cdots, C_{i,n_{i+1}}\}$, it amounts to computing the following function $\textsc{And}_{j \in [n_{i+1}]}\text{CC2BDD}(C_{i,j})$ which requires $(n_{i+1} - 1)$ AND-operations. A simple and straightforward approach is to initially compute a BDD G =CC2BDD($C_{i,1}$) and then iteratively computes the conjunction $G = \textsc{And}(G,\text{C2BDD}(C_{i,j}))$ of G and CC2BDD($C_{i,j}$) for $2 \leq j \leq n_{i+1}$. Alternatively, we use a divide-and-conquer (D&C) strategy to recursively compute the BDDs for the first half and the second half of the cardinality constraints respectively, and then apply the AND-operation to merge the results. The divide-and-conquer strategy does not reduce the number of AND-operations, but can reduce the sizes of the intermediate BDDs, and thus improve the efficiency of BDD encoding for the entire block. This has been confirmed by the experiments (cf. Section 6.1.2).

3.5.2 Input Propagation Strategy. It becomes prohibitively costly to construct the BDD directly from the cardinality constraints $t^{(b)}_i = \{C_{i,1}, \cdots, C_{i,n_{i+1}}\}$ when n_i and n_{i+1} are large, as the BDDs constructed via the procedure CC2BDD($C_{i,j}$) (cf. Algorithm 1) for $j \in [n_{i+1}]$ need to consider all the inputs in $\mathbb{B}^{n_i}$. To improve efficiency of BDD encoding, we apply feasible input propagation (IP) which propagates a given input region block-by-block, resulting in the feasible inputs of each block, with respect to the output of its preceding block t_{i-1}. When we construct the BDD for the block t_i, we only consider its feasible inputs. Although it introduces additional BDD operations (e.g., Exists and And), the feasible inputs of each block can significantly reduce the number of BDD nodes when the feasible outputs of the preceding block t_{i-1} is relatively smaller compared with the full input space $\mathbb{B}^{n_i}$ that the block t_i needs to consider. The effectiveness of this strategy has been confirmed in our experiments (cf. Section 6.1.3).

3.5.3 Algorithmic BDD Encoding of BNNs. Algorithm 2 shows the overall BDD encoding procedure. Given a BNN $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$ with s output classes and an input region $R(\mathbf{u}, \tau)$, the algorithm outputs the BDDs $(G^{out}_i)_{i \in [s]}$, encoding the input-output relation of the BNN $\mathcal{N}$ with respect to the input region $R(\mathbf{u}, \tau)$, where G^{out}_i for the output class $i \in [s]$ encodes the function $t^{(b)}_{d+1\downarrow i} \circ t^{(b)}_d \circ \cdots \circ t^{(b)}_1$.

In details, it first builds the BDD representation $G^{in}_{\mathbf{u},\tau}$ of the input region $R(\mathbf{u}, \tau)$ (Line 2) and the cardinality constraints from BNN $\mathcal{N}^{(b)}$ (Line 3).

The first loop (Lines 4–7) builds a BDD encoding the input-output relation of the entire internal blocks w.r.t. $G^{in}_{\mathbf{u},\tau}$. It first invokes the procedure InterBlk2BDD(t_i^b, G^{in}) to build a BDD G' encoding the input-output relation of the i-th block t_i^b w.r.t. the feasible inputs $\mathcal{L}(G^{in})$ (Line 5). Remark that the feasible inputs G^{in} of the block t_i^b is the feasible outputs of the $(i-1)$-th block t_{i-1}^b (the input region $G^{in}_{\mathbf{u},\tau}$ when $i=1$). By doing so, we have

$$\mathcal{L}(G') = \{(\mathbf{x}^i, \mathbf{x}^{i+1}) \in \mathcal{L}(G^{in}) \times \mathbb{B}^{n_i+1} \mid t_i^{(b)}(\mathbf{x}^i) = \mathbf{x}^{i+1}\}.$$

From the BDD G', we compute the feasible outputs G^{in} of the block $t_i^{(b)}$ by existentially quantifying all the input variables $\mathbf{x}^i$ of the block $t_i^{(b)}$ (Line 6). The BDD G^{in} serves as the set of feasible inputs of the block $t_{i+1}^{(b)}$ at the next iteration. We next assign G' to G if the current block is the first internal block (i.e., $i=1$), otherwise we compute the relational product of G and G', the resulting BDD G encodes the input-output relation of the first i internal blocks w.r.t. $G^{in}_{\mathbf{u},\tau}$ (Line 7). (Note that the input variables $\mathbf{x}^i$ of the block $t_i^{(b)}$, that are the output variables of the block $t_{i-1}^{(b)}$, in the relational product are existentially quantified.) Thus, we have

$$\begin{aligned}\mathcal{L}(G) =& \{(\mathbf{x}^1, \mathbf{x}^{i+1}) \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}) \times \mathbb{B}^{n_i+1} \mid (t_i^{(b)} \circ \cdots \circ t_1^{(b)})(\mathbf{x}^1) = \mathbf{x}^{i+1}\},\\ \mathcal{L}(G^{in}) =& \{\mathbf{x}^{i+1} \in \mathbb{B}^{n_i+1} \mid \exists \mathbf{x}^1 \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}).(\mathbf{x}^1, \mathbf{x}^{i+1}) \in \mathcal{L}(G)\}.\end{aligned}$$

At the end of the first for-loop, we obtain the BDD G encoding the input-output relation of the entire internal blocks and its feasible outputs G^{in} w.r.t. $G^{in}_{\mathbf{u},\tau}$, namely,

$$\begin{aligned}\mathcal{L}(G) =& \{(\mathbf{x}^1, \mathbf{x}^{d+1}) \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}) \times \mathbb{B}^{n_{d+1}} \mid (t_d^{(b)} \circ \cdots \circ t_1^{(b)})(\mathbf{x}^1) = \mathbf{x}^{d+1}\},\\ \mathcal{L}(G^{in}) =& \{\mathbf{x}^{d+1} \in \mathbb{B}^{n_{d+1}} \mid \exists \mathbf{x}^1 \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}).(\mathbf{x}^1, \mathbf{x}^{d+1}) \in \mathcal{L}(G)\}.\end{aligned}$$

At Line 8, we build the BDDs $(G_i)_{i\in[s]}$ for the output block $t_{d+1}^{(b)}$ by invoking the procedure OutBlk2BDD($t_{d+1}^{(b)}, G^{in}$), one BDD G_i per output class $i \in [s]$ such that

$$\mathcal{L}(G_i) = \{\mathbf{x}^{d+1} \in \mathcal{L}(G^{in}) \mid t_{d+1\downarrow i}^{(b)}(\mathbf{x}^{d+1}) = 1\}.$$

Finally, the second for-loop (Lines 9–10) builds the BDDs $(G_i^{out})_{i\in[s]}$, each of which encodes the input-output relation of the entire BNN and a class $i \in [s]$ w.r.t. the input region $G^{in}_{\mathbf{u},\tau}$. By computing the relational product of the BDDs G and G_i, we obtain the BDD G_i^{out} for the class $i \in [s]$. Recall that the BDD G encodes the input-output relation of the entire internal blocks w.r.t. the input region $G^{in}_{\mathbf{u},\tau}$. Thus, an input $\mathbf{x} \in R(\mathbf{u}, \tau)$ is classified into the class i by the BNN $\mathcal{N}$ iff $\mathbf{x}^{(b)} \in \mathcal{L}(G_i^{out})$.

Note that by modifying Line 2 to "$G^{in} = $ Const(1)", we can disable the feasible input propagation in Algorithm 2.

Procedure InterBlk2BDD**.** The procedure InterBlk2BDD is shown in Algorithm 3 which encodes a sequence of cardinality constraints into a BDD based on the divide-and-conquer strategy. Given a set of cardinality constraints $\{C_m, \cdots, C_n\}$ (note that indices matter, and $m=1, n=|n_{i+1}|$ for block t_i at initialization) and a BDD G^{in} encoding feasible inputs, InterBlk2BDD returns a BDD G. The BDD G encodes the input-output relation of the Boolean function $f_{m,n}$ such that for every $\mathbf{x}^i \in \mathcal{L}(G^{in})$, $f_{m,n}(\mathbf{x}^i)$ is the truth vector of the cardinality constraints $\{C_m, \cdots, C_n\}$ under the valuation $\mathbf{x}^i$. When $m=1$ and $n=n_{i+1}$, $f_{m,n}$ is the same as $t_i^{(b)}$, i.e.,

$$\mathcal{L}(G) = \{\mathbf{x}^i \times \mathbf{x}^{i+1} \in G^{in} \times \mathbb{B}^{n_{i+1}} \mid t_i^{(b)}(\mathbf{x}^i) = \mathbf{x}^{i+1}\}.$$

where $\mathbf{x}^i, \mathbf{x}^{i+1}$ denotes the input and output variables of $t_i^{(b)}$ respectively.

In detail, the procedure InterBlk2BDD computes the desired BDD in a binary search fashion.

Algorithm 3: BDD Construction of the i-th Internal Block

```
Procedure InterBlk2BDD(CCs : $\{C_m, \cdots, C_n\}$, Region : $G^{in}$)
    if $n == m$ then
        $G_1$ =CC2BDD($C_m$);
        $G$ = Xnor(New($\mathbf{x}_m^{i+1}$), $G_1$) ;  // $\mathbf{x}_m^{i+1}$ denotes the $m$-th entry of the output of $t_i^{(b)}$
    else if $n == m + 1$ then
        $G_1$ =CC2BDD($C_m$); $G_1$ = Xnor(New($\mathbf{x}_m^{i+1}$), $G_1$);
        $G_2$ =CC2BDD($C_n$); $G_2$ = Xnor(New($\mathbf{x}_n^{i+1}$), $G_2$);
        $G$ = And($G^{in}$, $G_1$); $G$ = And($G$, $G_2$);
    else
        $G_1$ =InterBlk2BDD($\{C_m, \cdots, C_{\lfloor \frac{n-m}{2} \rfloor + m}\}$, $G^{in}$);
        $G_2$ =InterBlk2BDD($\{C_{\lfloor \frac{n-m}{2} \rfloor + m+1}, \cdots, C_n\}$, $G^{in}$);
        $G$ = And($G_1$, $G_2$);
    return $G$
```

Algorithm 4: BDD Construction of Output Blocks

```
Procedure OutBlk2BDD($t_{d+1}^{(b)}$, Region : $G^{in}$)
    for ($i = 1$; $i \leq s$; $i++$) do
        $G_i = G^{in}$;
        for ($j = 1$; $j \leq s - 1$; $j++$) do
            $G_i$ =And($G_i$,CC2BDD($t_{d+1 \downarrow i,j}^{(b)}$));
    return $(G_i)_{i \in [s]}$
```

- If $n == m$, it first builds the BDD G_1 for the cardinality constraint C_m by invoking the procedure CC2BDD so that $\mathcal{L}(G_1)$ represents the solutions of C_m. Then G_1 is transformed into the BDD Xnor(New($\mathbf{x}_m^{i+1}$), G_1), encoding the input-output relation of $t_{i \downarrow m}^{(b)}$, thus $t_{i \downarrow m}^{(b)}(\mathbf{x}) = 1$ iff $\mathbf{x} \in \mathcal{L}(G_1)$. Note that we regard $\mathbf{x}_m^{i+1}$ as a newly added BDD variable when applying the Xnor-operation, thus G_1 has n_i BDD variables, the same as the length of input vectors to the i-th internal block t_i.
- If $n == m + 1$, it first builds the BDDs G_1 and G_2 for the two cardinality constraints C_m and C_n such that $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$ represent the sets of solutions of C_m and C_n. Then, the BDDs G_1 and G_2 are transformed into the BDDs Xnor(New($\mathbf{x}_m^{i+1}$), G_1) and Xnor(New($\mathbf{x}_n^{i+1}$), G_2), encoding $t_{i \downarrow m}^{(b)}$ and $t_{i \downarrow n}^{(b)}$, respectively. Then, we compute the conjunction G of the BDDs G^{in}, G_1 and G_2. Note that, here we first compute And(G^{in}, G_1), then the resulting BDD is conjuncted with G_2.
- Otherwise, we build the BDDs G_1 and G_2 for $\{C_m, \cdots, C_{\lfloor \frac{n-m}{2} \rfloor + m}\}$ and $\{C_{\lfloor \frac{n-m}{2} \rfloor + m+1}, \cdots, C_n\}$, and then compute the conjunction G of them. Thus, for every $(\mathbf{x}^i, \mathbf{x}^{i+1}) \in \mathcal{L}(G)$, $\mathbf{x}^{i+1}$ is the truth vector of the constraints $\{C_m, \cdots, C_n\}$ under the valuation $\mathbf{x}^i$.

Procedure OutBlk2BDD. The procedure OutBlk2BDD is shown in Algorithm 4 which encodes the cardinality constraints of the output block into the BDDs $(G_i)_{i \in [s]}$, one BDD G_i per class $i \in [s]$. Different from the BDD encoding of the internal blocks, for each output class $i \in [s]$ of the output block $t_{d+1}^{(b)}$, OutBlk2BDD directly conjuncts the feasible inputs G^{in} with the $(s-1)$ BDDs of the cardinality constraints $t_{d+1 \downarrow i}^{(b)} = \{C_{d+1}^{i,1}, \cdots C_{d+1}^{i,i-1}, C_{d+1}^{i,i+1}, \cdots, C_{d+1}^{i,s}\}$, which encodes the input-output relation of the output block for the class i. We could also use the divide-and-conquer strategy for the output block, but it does not improve efficiency, as the number of classes is much smaller compared to the sizes of the outputs of the internal blocks and the divide-and-conquer strategy introduces additional AND-operations.

We should emphasize that instead of encoding the input-output relation of the BNN $\mathcal{N}$ as a sole BDD or MTBDD, we opt to use a family of BDDs $(G_i^{out})_{i\in[s]}$, each of which corresponds to one output class of $\mathcal{N}$. Building a single BDD or MTBDD for the BNN is possible from $(G_i^{out})_{i\in[s]}$, but our approach gives the flexibility especially when a specific target class is interested, which is common for robustness analysis.

Theorem 3.9. *Given a BNN $\mathcal{N}$ with s output classes and an input region $R(\mathbf{u},\tau)$, we can compute BDDs $(G_i^{out})_{i\in[s]}$ such that the BNN $\mathcal{N}$ classifies an input $\mathbf{x} \in R(\mathbf{u},\tau)$ into the class $i \in [s]$ iff $\mathbf{x}^{(b)} \in \mathcal{L}(G_i^{out})$.*

Proof. By Lemma 3.3 and Lemma 3.4, Line 2 in Algorithm 2 encodes the input region $R(\mathbf{u},\tau)$ into a BDD $G^{in}_{\mathbf{u},\tau}$ such that $\mathcal{L}(G^{in}_{\mathbf{u},\tau}) = R(\mathbf{u},\tau)$. By Theorem 3.7, an input $\mathbf{x} \in R(\mathbf{u},\tau)$ is classified into class $i \in [s]$ iff $t^{(b)}_{d+1\downarrow i}((t^{(b)}_d \circ \cdots \circ t^{(b)}_1)(\mathbf{x}^{(b)})) = 1$, where $\mathbf{x} = 2\mathbf{x}^{(b)} - 1$. It remains to prove that $\mathcal{L}(G_i^{out}) = \{\mathbf{x}^{(b)} \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}) \mid t^{(b)}_{d+1\downarrow i}((t^{(b)}_d \circ \cdots \circ t^{(b)}_1)(\mathbf{x}^{(b)})) = 1\}$ for each class $i \in [s]$.

The first for loop (Lines 4–7) in Algorithm 2 builds a BDD G encoding the input-output relation of the entire internal blocks, namely,

$$\mathcal{L}(G) = \{(\mathbf{x}^1, \mathbf{x}^{d+1}) \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}) \times \mathbb{B}^{n_{d+1}} \mid (t^{(b)}_{d+1} \circ \cdots \circ t^{(b)}_1)(\mathbf{x}^1) = \mathbf{x}^{d+1}\}.$$

For each class $i \in [s]$, the BDD G_i constructed at Line 8 in Algorithm 2 for the output block $t^{(b)}_{d+1}$ satisfies that $\mathcal{L}(G_i) = \{\mathbf{x}^{d+1} \in \mathcal{L}(G^{in}) \mid t^{(b)}_{d+1\downarrow i}(\mathbf{x}^{d+1}) = 1\}$. Finally, by computing the relational product of G and G_i for each class $i \in [s]$ at Line 10 in Algorithm 2, the BDD G_i^{out} is equivalent to $\exists \mathbf{x}^{d+1}.\textsc{And}(G, G_i)$. Thus, $\mathcal{L}(G_i^{out}) = \{\mathbf{x}^{(b)} \in \mathcal{L}(G^{in}_{\mathbf{u},\tau}) \mid t^{(b)}_{d+1\downarrow i}((t^{(b)}_d \circ \cdots \circ t^{(b)}_1)(\mathbf{x}^{(b)})) = 1\}$. □

Our encoding explicitly involves $O(d+s)$ RelProd-operations, $O(s^2 + \sum_{i\in[d]} n_i)$ And-operations, $O(\sum_{i\in[d]} n_i)$ Xnor-operations and $O(d)$ Exists-operations.

4 PARALLELIZATION STRATEGIES

In this section, we investigate parallelization strategies at various levels, aiming to improve the encoding efficiency further. In general, we classify them into three levels: low-level BDD operations, high-level BDD encoding of blocks, and BDD construction of the entire BNN.

4.1 Parallel BDD operations

Designing an effective and efficient parallel decision diagram implementation for BDDs is non-trivial. We resort to Sylvan, a novel parallel decision diagram implementation, that parallelizes the most common BDD operations and features parallel garbage collection. There are other BDD implementations (e.g., BeeDeeDee [61]) supporting multi-threaded BDD manipulation. We choose Sylvan based on the comparative study of BDD implementations for probabilistic symbolic model checking [101].

As a matter of fact, the speedup of the parallel BDD operations provided by Sylvan depends on the number of workers used by Sylvan and the size of the underlying problem (i.e., BNN and input region). Increasing the number of workers does not necessarily improve the encoding efficiency, as the overhead induced by the synchronisation between workers may outweigh. Indeed, as stated in [102], the limited parallel scalability is expected when the amount of parallelism in the computation task is not sufficient. Our experiments observe that, with the increase of the number

Algorithm 5: Parallel BDD Construction of the i-th Internal Block

```
Procedure InterBlk2BDDPara(CCs : $\{C_m, \cdots, C_n\}$, InputSpace : $G^{in}$)
    if $n == m$ then
        $G_1$ =CC2BDD($C_m$);
        $G$ = Xnor(New($\mathbf{x}_m^{i+1}$), $G_1$) ;  // $\mathbf{x}_m^{i+1}$ denotes the $m$-th entry of the output of $t_i^{(b)}$
    else if $n == m + 1$ then
        $G_1$ =CC2BDD($C_m$); $G_1$ = Xnor(New($\mathbf{x}_m^{i+1}$), $G_1$);
        $G_2$ =CC2BDD($C_n$); $G_2$ = Xnor(New($\mathbf{x}_n^{i+1}$), $G_2$);
        $G$ = And($G^{in}$, $G_1$); $G$ = And($G$, $G_2$);
    else
        construct the BDDs $G_1$ and $G_2$ in parallel:
            $G_1$ =InterBlk2BDDPara($\{C_m, \cdots, C_{\lfloor \frac{n-m}{2} \rfloor + m}\}$, $G^{in}$);
            $G_2$ =InterBlk2BDDPara($\{C_{\lfloor \frac{n-m}{2} \rfloor + m + 1}, \cdots, C_n\}$, $G^{in}$);
        $G$ = And($G_1$, $G_2$);
    return $G$
```

Algorithm 6: Parallel BDD Construction for Output Blocks

```
Procedure OutBlk2BDDPara($t_{d+1}^{(b)}$, InputSpace : $G^{in}$)
    construct the BDDs $G_i$'s for $i \in [s]$ in parallel:
        $G_i$ = $G^{in}$;
        for ($j = 1;\ j \leq s - 1;\ j + +$) do
            $G_i$ =And($G_i$, CC2BDD($t_{d+1 \downarrow i,j}^{(b)}$));
    return $(G_i)_{i \in [s]}$
```

of workers, the improvement is limited for small BNNs and input regions, but significant for large BNNs and input regions.

4.2 Parallel BDD Encoding of Blocks

In order to improve the encoding efficiency further, we investigate parallelization strategies at the level of BDD encoding for both internal and output blocks. Our goal is to increase parallelism in the BDD encoding for each block so that the efficiency can be improved with the increase of the number of workers.

Parallelization for internal blocks. To improve the efficiency of BDD encoding for internal blocks, we propose a parallel divide-and-conquer strategy, as shown in Algorithm 5. It is similar to the sequential procedure InterBlk2BDD shown in Algorithm 3, except that the BDDs G_1 and G_2 for $\{C_m, \cdots, C_{\lfloor \frac{n-m}{2} \rfloor + m}\}$ and $\{C_{\lfloor \frac{n-m}{2} \rfloor + m + 1}, \cdots, C_n\}$ are constructed in parallel. In our implementation, we use the SPAWN API provided by Lace in Sylvan to spawn two tasks to construct the BDDs G_1 and G_2 simultaneously, and use the SYNC API to synchronize the two tasks. Finally, the BDDs G_1 and G_2 are conjuncted together. The key advantage of the parallel divide-and-conquer strategy is the ability to largely reduce the overall synchronization when constructing the BDD for an internal block. As a result, the encoding efficiency increases with the increase of the number of workers for small BNNs and small input regions, confirmed by our experiments (cf. Section 6.1.4). When combined with parallel BDD operations, the encoding efficiency increases with the increase of the number of workers for arbitrary BNNs and arbitrary input regions, hence features a more stable acceleration.

Algorithm 7: Parallel BDD Construction of BNNs without Input Propagation

```
Procedure BNN2BDDPara(BNN : $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$, Region : $R(\mathbf{u}, \tau)$)
    $G^{in} = G^{in}_{\mathbf{u},\tau}$;
    $\mathcal{N}^{(b)} = (t_1^{(b)}, \cdots, t_d^{(b)}, t_{d+1}^{(b)})$;
    construct the BDDs $G_i'$'s for $i \in [d]$ in parallel:
        $G^{in} = (i == 1)\ ?\ G^{in}_{\mathbf{u},\tau} : \textsc{Const}(1)$;
        $G_i' = \textsc{InterBlk2BDDPara}(t_i^{(b)}, G^{in})$ (cf. Algorithm 5);
    $G = G_1'$;
    for $(i = 2;\ i \le d;\ i{+}{+})$ do
        $G = \textsc{RelProd}(G, G_i', \mathbf{x}^i)$;
    $(G_i)_{i \in [s]} = \textsc{OutBlk2BDDPara}(t_{d+1}^{(b)}, G^{in})$;
    for $(i = 1;\ i \le s;\ i{+}{+})$ do
        $G_i^{out} = \textsc{RelProd}(G, G_i, \mathbf{x}^{d+1})$;
    return $(G_i^{out})_{i \in [s]}$
```

Parallelization for output blocks. To parallelize the BDD encoding of an output block, a straightforward approach is to construct the BDD G_i in parallel by leveraging the parallel divide-and-conquer strategy for a class $i \in [s]$, as done for internal blocks in Algorithm 5. However, such a strategy cannot improve the encoding efficiency. One possible reason is that the overhead induced by the additional And-operations still occupies a large proportion of the total encoding time, as the size of feasible input BDD G^{in} can be quite large for the output block. Alternatively, we choose to construct the BDDs G_i's for the classes $i \in [s]$ in parallel, as shown in Algorithm 6.

4.3 Parallel BDD Construction of an Entire BNN

We investigate the potential parallelization strategies at the level of BDD construction for the entire BNN. However, it is non-trivial to parallelize the composition of BDDs of the blocks, i.e., the first for-loop (Lines 4–7) in Algorithm 2, as the feasible inputs of a block is computed as the feasible outputs of its preceding block. Thus, to parallelize the BDD construction of an entire BNN, we have to disable the feasible input propagation, as shown in Algorithm 7. This definitely results in the loss of benefit induced by input propagation. Thus, we expected that this strategy is effective for large input space, e.g., the full input space $\mathbb{B}^{n_1}_{\pm 1}$, for which input propagation becomes less effective. However, our experiments show that this is not the case, as the parallelization at this level reduces the number of workers that can be used for parallel BDD encoding of blocks and BDD operations.

Remark. We have also made some other attempts to improve the overall BDD encoding efficiency, including (1) parallelize the composition of two BDDs, e.g., the second for-loop in Algorithm 2 and the for-loops in Algorithm 7; (2) move the second for-loop of Algorithm 7 into the procedure OutBlk2BDDPara which composes the BDD G of the internal blocks with the BDD G_i of the output block w.r.t. the class i. However, these strategies could not improve the overall encoding efficiency, and sometimes even incurred additional overhead.

5 APPLICATIONS: ROBUSTNESS ANALYSIS AND INTERPRETABILITY

In this section, we highlight two applications within BNNQuanalyst, i.e., robustness analysis and interpretability of BNNs.

5.1 Robustness Analysis

Definition 5.1. Given a BNN $\mathcal{N}$ and an input region $R(\mathbf{u}, \tau)$, the BNN is (locally) *robust* w.r.t. the region $R(\mathbf{u}, \tau)$ if each sample $\mathbf{x} \in R(\mathbf{u}, \tau)$ is classified into the same class of the input $\mathbf{u}$, i.e., $\forall \mathbf{x} \in R(\mathbf{u}, \tau), \mathcal{N}(\mathbf{x}) = \mathcal{N}(\mathbf{u})$.

An *adversarial example* in the region $R(\mathbf{u}, \tau)$ is a sample $\mathbf{x} \in R(\mathbf{u}, \tau)$ such that $\mathbf{x}$ is classified into a class, that differs from the predicted class of the input $\mathbf{u}$, i.e., $\mathcal{N}(\mathbf{x}) \neq \mathcal{N}(\mathbf{u})$.

As mentioned in Section 1, qualitative verification which checks whether a BNN is robust is insufficient in many practical applications. In this paper, we are interested in *quantitative* verification of robustness which asks *how many adversarial examples are there in the input region of the BNN for each class.* To answer this question, given a BNN $\mathcal{N}$ and an input region $R(\mathbf{u}, \tau)$, we first obtain the BDDs $(G_i^{out})_{i \in [s]}$ by applying Algorithm 2 (or its parallel variants) and then count the number of adversarial examples for each class in the input region $R(\mathbf{u}, \tau)$. Note that counting adversarial examples amounts to computing $|R(\mathbf{u}, \tau)| - |\mathcal{L}(G_g^{out})|$, where g denotes the predicted class of $\mathbf{u}$, and $|\mathcal{L}(G_g^{out})|$ can be computed in time $O(|G_g^{out}|)$.

We remark that the robustness and adversarial examples are defined w.r.t. the predicted class of the input $\mathbf{u}$ instead of its class given in the dataset, which is the same as NPAQ [8] but different from [118].

In many real world applications, more refined analysis is needed. For instance, it may be acceptable to misclassify a dog as a cat, but unacceptable to misclassify a tree as a car [79]. This suggests that the robustness of BNNs may depend on the classes into which samples are misclassified. To address this, we consider the notion of targeted robustness.

Definition 5.2. Given a BNN $\mathcal{N}$, an input region $R(\mathbf{u}, \tau)$ and the class t, the BNN is *t-target-robust* w.r.t. the region $R(\mathbf{u}, \tau)$ if every sample $\mathbf{x} \in R(\mathbf{u}, \tau)$ is never classified into the class t, i.e., $\mathcal{N}(\mathbf{x}) \neq t$. (Note that we assume that the predicted class of $\mathbf{u}$ differs from the class t.)

The quantitative verification problem of t-target-robustness of a BNN asks *how many adversarial examples in the input region $R(\mathbf{u}, \tau)$ are misclassified to the class t by the BNN.* To answer this question, we first obtain the BDD G_t^{out} and then count the number of adversarial examples by computing $|\mathcal{L}(G_t^{out})|$.

Note that, if one wants to compute the (locally) maximal safe Hamming distance that satisfies a robustness property for an input sample (e.g., the proportion of adversarial examples is below a given threshold), our framework can incrementally compute such a distance without constructing the BDD models of the entire BNN from scratch.

Definition 5.3. Given a BNN $\mathcal{N}$, an input region $R(\mathbf{u}, r)$ and the threshold $\epsilon \geq 0$, r_1 is the (locally) maximal safe Hamming distance of $R(\mathbf{u}, \tau)$, if one of the follows holds:

- r_1 is the maximal one such that $r_1 < r$ and $Pr(R^{\text{adv}}(\mathbf{u}, r_1)) \leq \epsilon$ if $Pr(R^{\text{adv}}(\mathbf{u}, r)) > \epsilon$;
- r_1 is the maximal one such that $r_1 \geq r$ and $Pr(R^{\text{adv}}(\mathbf{u}, r')) \leq \epsilon$ for all $r' : r \leq r' \leq r_1$ if $Pr(R^{\text{adv}}(\mathbf{u}, r)) \leq \epsilon$;

where $Pr(R^{\text{adv}}(\mathbf{u}, r))$ is the probability $\frac{\sum_{i \in [s].i \neq g} |\mathcal{L}(G_i^{out})|}{|R(\mathbf{u}, r)|}$ for g being the predicted class of $\mathbf{u}$, assuming a uniform distribution over the entire sample space.

Algorithm 8 shows the procedure to incrementally compute the maximal safe Hamming distance for a given threshold $\epsilon \geq 0$, input region $R(\mathbf{u}, r)$ and predicted class g of $\mathbf{u}$. Basically, it searches for the maximal safe Hamming distance by either increasing or decreasing the distance r depending on whether $Pr(R^{\text{adv}}(\mathbf{u}, r)) > \epsilon$.

Algorithm 8: Compute the maximal safe Hamming distance

```
Procedure MaxHD(BNN : $\mathcal{N} = (t_1, \cdots, t_d, t_{d+1})$, Region : $R(\mathbf{u}, r)$, Threshold : $\epsilon$, Class : $g$)
    $(G_i^{out})_{i \in [s]}$ =BNN2BDD($\mathcal{N}, R(\mathbf{u}, r)$);
    if $(Pr(R^{\text{adv}}(\mathbf{u}, r)) > \epsilon)$ then // decrease $r$
        while $(r > 0)$ do
            $r = r - 1$;
            $(G_i^{out})_{i \in [s]} = (\textsc{And}(G_{\mathbf{u},r}^{in}, G_i^{out}))_{i \in [s]}$;
            if $(Pr(R^{\text{adv}}(\mathbf{u}, r)) \leq \epsilon)$ then
                return $r$
    else // increase $r$
        while $(r < n_1)$ do // $n_1$ is the input size of the BNN $\mathcal{N}$
            $r = r + 1$;
            $(B_i^{out})_{i \in [s]}$ =BNN2BDD($\mathcal{N}, R(\mathbf{u}, r) \setminus R(\mathbf{u}, r-1)$);
            $(G_i^{out})_{i \in [s]} = (\textsc{Or}(B_i^{out}, G_i^{out}))_{i \in [s]}$;
            if $(Pr(R^{\text{adv}}(\mathbf{u}, r)) > \epsilon)$ then
                return $r - 1$
    return $r$
```

- If $Pr(R^{\text{adv}}(\mathbf{u}, r)) > \epsilon$, we iteratively decrease r by 1 and compute the intersection between the new input region $R(\mathbf{u}, r)$ and original BDD model G_i^{out} for each class $i \in [s]$, until $Pr(R^{\text{adv}}(\mathbf{u}, r)) \leq \epsilon$ or $r = 0$.
- If $Pr(R^{\text{adv}}(\mathbf{u}, r)) \leq \epsilon$, we iteratively increase r by 1, obtain the BDDs $(B_i^{out})_{i \in [s]}$ of the BNN w.r.t. the input region $R(\mathbf{u}, r) \setminus R(\mathbf{u}, r-1)$, and compute the union of B_i^{out} and G_i^{out} for each class $i \in [s]$, until $Pr(R^{\text{adv}}(\mathbf{u}, r)) > \epsilon$ or $r = n_1$. Recall that the input region $R(\mathbf{u}, r)$ can be expressed by a constraint $\sum_{j=1}^{n_1} \ell_j \leq r$ (cf. Section 3.3). Thus, $R(\mathbf{u}, r) \setminus R(\mathbf{u}, r-1)$ can be expressed by the conjunction of two constraints $\sum_{j=1}^{n_1} \ell_j \leq r$ and $\sum_{j=1}^{n_1} \ell_j > r - 1$, which can be further reformulated into two cardinality constraints $\sum_{j=1}^{n_1} \neg \ell_j \geq n_1 - r$ and $\sum_{j=1}^{n_1} \ell_j \geq r$. The BDD encoding of the input region $R(\mathbf{u}, r) \setminus R(\mathbf{u}, r-1)$ can be constructed by applying the And-operation to the BDDs of $\sum_{j=1}^{n_1} \neg \ell_j \geq n_1 - r$ and $\sum_{j=1}^{n_1} \ell_j \geq r$.

5.2 Interpretability

In general, interpretability addresses the question of *why some inputs in the input region are (mis)classified by the BNN into a specific class?* We consider the interpretability of BNNs using two complementary explanations, i.e., prime implicant explanations and essential features.

Definition 5.4. Given a BNN $\mathcal{N}$, an input region $R(\mathbf{u}, \tau)$ and a class g,

- a *prime implicant explanation* (PI-explanation) of the decision made by the BNN $\mathcal{N}$ on the input $\mathcal{L}(G_g^{out})$ is a minimal set of literals $\{\ell_1, \cdots, \ell_k\}$ such that for every $\mathbf{x} \in R(\mathbf{u}, \tau)$, if $\mathbf{x}$ satisfies $\ell_1 \wedge \cdots \wedge \ell_k$, then $\mathbf{x}$ is classified into the class g by $\mathcal{N}$.
- the *essential features* for the inputs $\mathcal{L}(G_g^{out})$ are literals $\{\ell_1, \cdots, \ell_k\}$ such that every $\mathbf{x} \in R(\mathbf{u}, \tau)$, if $\mathbf{x}$ is classified into the class g by $\mathcal{N}$, then $\mathbf{x}$ satisfies $\ell_1 \wedge \cdots \wedge \ell_k$.

Intuitively, a PI-explanation $\{\ell_1, \cdots, \ell_k\}$ indicates that $\{\text{var}(\ell_1), \cdots, \text{var}(\ell_k)\}$ are the key such that, when fixed, the prediction is guaranteed no matter how the remaining features change. Thus, a PI-explanation can be seen as a sufficient condition to be classified into the class. Remark that there may be more than one PI-explanation for a set of inputs. When g is set to be the predicted

Table 3. BNN benchmarks based on MNIST and Fashion-MNIST (F-MNIST), where $n_1 : \cdots : n_{d+1} : s$ in the column (Architecture) denotes that the BNN model has $d + 1$ blocks, n_1 inputs and s outputs; the i-th block for $i \in [d + 1]$ has n_i inputs and n_{i+1} outputs with $n_{d+2} = s$.

Dataset	Name	Architecture	Accuracy	Name	Architecture	Accuracy
MNIST	P1	16:25:20:10	14.88 %	P7	100:100:10	75.16%
	P2	16:64:32:20:10	25.14%	P8	100:50:20:10	71.1%
	P3	25:25:25:20:10	33.67%	P9	100:100:50:10	77.37%
	P4	36:15:10:10	27.12%	P10	400:100:10	83.4%
	P5	64:10:10	49.16%	P11	784:100:10	85.13%
	P6	100:50:10	73.25%	P12	784:50:50:50:50:10	86.95%
F-MNIST	P13	100:100:10	50.01%	P15	100:100:50:10	50.40%
	P14	100:50:20:10	39.42%	P16	784:100:10	50.25%

class of the benign input $\mathbf{u}$, a PI-explanation on the input region G_g^{out} suggests why these samples are correctly classified into g.

The essential features $\{\ell_1, \cdots, \ell_k\}$ denote the key features such that all samples $\mathbf{x} \in R(\mathbf{u}, \tau)$ that are classified into the class g by the BNN $\mathcal{N}$ must agree on these features. Essential features differ from PI-explanations, where the former can be seen as a necessary condition, while the latter can be seen as a sufficient condition.

The CUDD package provides APIs to identify prime implicants (e.g., Cudd_bddPrintCover and Cudd_FirstPrime) and essential variables (e.g., Cudd_FindEssential). Therefore, prime implicants and essential features can be computed via queries on the BDDs $(G_i^{out})_{i\in[s]}$. (Note that the Sylvan package does not provide such APIs, but this problem could be solved by storing the BDDs $(G_i^{out})_{i\in[s]}$ constructed by Sylvan in a file, which is then loaded by CUDD.)

6 EVALUATION

We have implemented the framework, giving rise to a new tool BNNQuanalyst. The implementation is based on CUDD [90] as the default BDD package and Sylvan [102] for parallel computing. We use Python as the front-end to process BNNs and C++ as the back-end to perform the BDD encoding and analysis. In the rest of this section, we report the experimental results on the BNNs trained using the MNIST dataset, including the performance of BDD encoding, robustness analysis based on both Hamming distance and fixed indices, and interpretability.

Benchmarks on MNIST and Fashion-MNIST. We use the PyTorch deep learning platform provided by NPAQ [8] to train and test 12 BNNs (P1-P12) with varying sizes based on the MNIST dataset, and 4 BNNs (P13-P16) on the Fashion-MNIST dataset. Similar to NPAQ [8], we first resize the original 28 × 28 images to the input size n_1 of the BNN (i.e., the corresponding image is of the dimension $\sqrt{n_1} \times \sqrt{n_1}$) via Bilinear interpolation [11] and then binarize the normalized pixels of the images. Table 3 gives the details of the 16 BNN models based on two datasets, each of which has 10 classes (i.e., $s = 10$). Column 1 shows the dataset used for training and evaluation. Columns 2 and 5 give the name of the BNN model. Columns 3 and 6 show the architecture of the BNN model, where $n_1 : \cdots : n_{d+1} : s$ denotes that the BNN model has $d + 1$ blocks, n_1 inputs and s outputs; the i-th block for $i \in [d + 1]$ has n_i inputs and n_{i+1} outputs with $n_{d+2} = s$. Therefore, the number of the internal blocks ranges from 1 to 4 (i.e., 3 to 12 layers), the dimension of inputs ranges from 25 to 784, and the number of hidden neurons per LIN layer in each block ranges from 10 to 100. Columns 4 and 7 give the accuracy of the BNN model on the test set of the two datasets. We can observe that the accuracy of the BNN models increases with the size of the inputs, the number of layers and the number of hidden neurons per layer for each dataset. We remark that the small BNNs

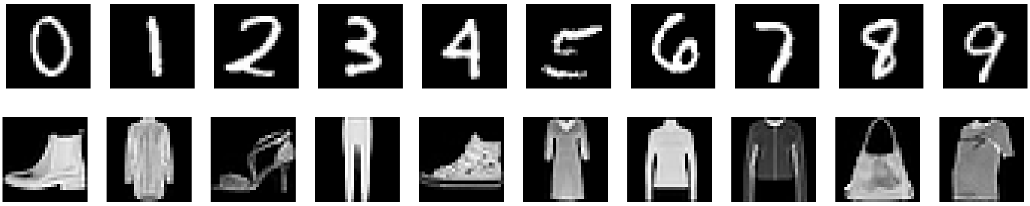


Fig. 8. Images from MNIST (top) and Fashion-MNIST (bottom) used to evaluate our approach.

(e.g., P1–P5) with low accuracy are used to understand (i) the effectiveness of input propagation strategy under full input space (cf. Section 6.1.3), (ii) the effectiveness of different parallelization strategies under full input space (cf. Section 6.1.4), and (iii) the efficiency of our BDD encoding under full input space (cf. Section 6.2.1). Figure 8 shows 20 images from the test sets of two datasets (10 images from MNIST and 10 images from Fashion-MNIST, where one image per class) to evaluate our approach unless explicitly stated. Similar to prior work [8, 15, 71, 105, 106], those images are chosen randomly. Specifically, the image of digit i from the test set of MNIST is called by i-image.

Experimental setup. The experiments were conducted on a 20-core machine (with two-way hyper-threading) with 2×2.2GHz Intel 10-core Xeon Silver 4114 processors and 376 GB of main memory of which 256 GB are used for evaluation. We kept the default values for the BDD packages except that: (1) the initial cache size is set to 2^{18} entries for both CUDD and Sylvan, (2) the maximum sizes of the BDD node hash table and operation cache for CUDD are set to 0 (i.e., no limitation), and (3) the maximum cache size, initial and maximum size of the BDD node hash table for Sylvan are set to 2^{30}, 2^{22} and 2^{33} entries (called buckets in Sylvan). The variable ordering of BDDs used for input regions is the natural row-by-row, left-to-right of the pixels in the images, the same as the prior work [83]; the variable ordering of BDDs used for each internal block and composition of internal blocks follow the orders of inputs and outputs, where the input variables are smaller than output variables; and the variable ordering of BDDs used for the output block follows the orders of inputs. The time limit for each experiment is set to be 8 hours. For parallel computing, the maximum number of workers is limited up to 39 (out of 40) where the remaining one is reserved for the other computation tasks and system processes.

6.1 Effectiveness of Strategies

We first compare the performance of our graph-based algorithm (i.e., Algorithm 1) and the DP-based algorithm [27] for the BDD encoding of cardinality constraints. We then evaluate the effectiveness of input propagation, divide-and-conquer and parallelization strategies for BDD encoding of BNNs. We repeated each experiment 3 times and report the rounded average time in seconds of 3 runs. For the sake of presentation, we define abbreviated names of sequential and parallel algorithms with various strategies in Table 4.

6.1.1 Graph-based Alg. vs. DP-based Alg. We compare the performance of our graph-based algorithm and the DP-based algorithm using the cardinality constraints obtained from the input regions. The input regions use the 0-image with the input sizes 100 and 784, where the Hamming distance ranges from 10 to 40 with step 10 for the input size 100, and ranges from 50 to 200 with step 50 for the input size 784.

The results are reported in Table 5 using CUDD. We can observe that our graph-based algorithm is at least 20 times faster than the DP-based algorithm [27]. The improvement is much significant for large cardinality constraints, e.g., for the instances 784-HD-150 and 784-HD-200.

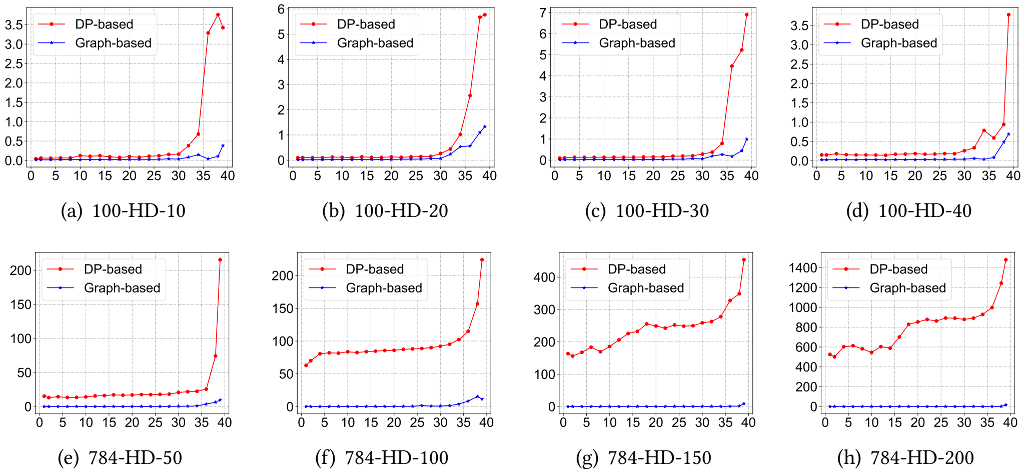
Table 4. Abbreviated names of sequential and parallel algorithms with various strategies.

Abbreviation	Description
DP-based Alg.	Dynamic programming based algorithm for BDD encoding of cardinality constraints [27]
Graph-based Alg.	Our algorithm for BDD encoding of cardinality constraints, i.e., Algorithm 1
L0-D&C	Algorithm 2 with CUDD
L0-Iteration	Algorithm 2 with CUDD, but without the divide-and-conquer strategy
L1-D&C	Algorithm 2 with Sylvan
L1-Iteration	Algorithm 2 with Sylvan, but without the divide-and-conquer strategy
L2-D&C	Algorithm 2 with Sylvan, where InterBlk2BDD and OutBlk2BDD are replaced by InterBlk2BDDPara (cf. Algorithm 5) and OutBlk2BDDPara (cf. Algorithm 6)
L2-D&C-NoIP	L2-D&C but without input propagation
L3-D&C	Algorithm 7 with Sylvan

Table 5. Execution time (in seconds) of our graph-based algorithm and the DP-based algorithm for BDD encoding of cardinality constraints using CUDD, where x-HD-y denotes the cardinality constraint obtained from the input region with the input size x and Hamming distance y.

	100-HD-10	100-HD-20	100-HD-30	100-HD-40
DP-based Alg.	0.021	0.037	0.059	0.059
Graph-based Alg.	0.001	0.001	0.001	0.002

	784-HD-50	784-HD-100	784-HD-150	784-HD-200
DP-based Alg.	11.05	37.45	617.3	1123.45
Graph-based Alg.	0.023	0.041	0.052	0.061



(a) 100-HD-10 (b) 100-HD-20 (c) 100-HD-30 (d) 100-HD-40

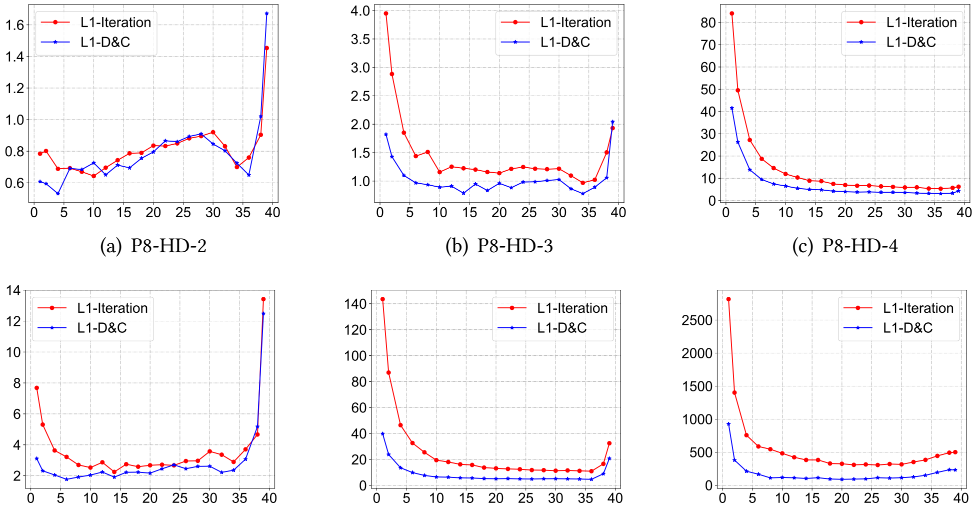
(e) 784-HD-50 (f) 784-HD-100 (g) 784-HD-150 (h) 784-HD-200

Fig. 9. Execution time (in seconds) of our graph-based algorithm and the DP-based algorithm for BDD encoding of cardinality constraints using Sylvan, where the numbers of workers (x-axis) are 1, 2, 4, 6, 8, ⋯ , 36, 38, 39.

We also conduct the same experiments using Sylvan, where the number of workers is set to 1, 2, 4, 6, 8, ⋯ , 36, 38, 39. The results are depicted in Figure 9. We can observe that our graph-based algorithm also performs significantly better than the DP-based algorithm [27]. However, with the increased number of workers, the performance of both algorithms downgraded, in particular, for

Table 6. Execution time (in seconds) of L0-Iteration and L0-D&C for BDD encoding of BNNs, where Px-HD-y denotes the BNN model Px and Hamming distance y.

	P8-HD-2	P8-HD-3	P8-HD-4	P9-HD-2	P9-HD-3	P9-HD-4
L0-Iteration	0.44	6.67	165.5	10.14	347.59	6318.7
L0-D&C	0.17	1.88	71.4	2.66	86.13	1598.6



(a) P8-HD-2 (b) P8-HD-3 (c) P8-HD-4

(d) P9-HD-2 (e) P9-HD-3 (f) P9-HD-4

Fig. 10. Execution time (in seconds) of L1-Iteration and L1-D&C for BDD encoding of BNNs, where the numbers of workers (x-axis) are $1, 2, 4, 6, 8, \cdots, 36, 38, 39$.

small cardinality constraints (cf. Figures 9(a)—9(d).) This is mainly caused by the overhead induced by the synchronisation between workers on the parallel BDD operations mentioned previously.

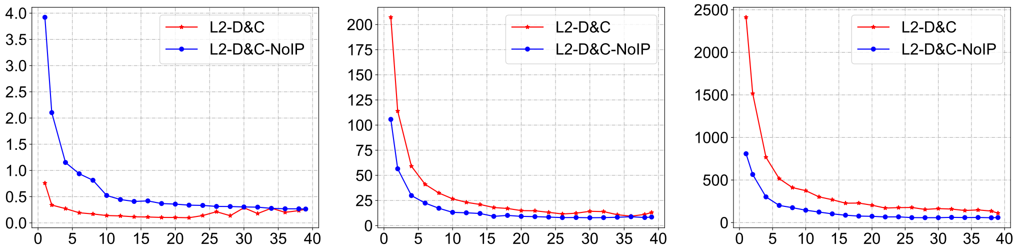
6.1.2 Effectiveness of the Divide-and-Conquer Strategy. To study the effectiveness of the divide-and-conquer strategy, we compare the performance between L0-Iteration and L0-D&C both of which use sequential BDD operations (i.e., CUDD), and between L1-Iteration and L1-D&C both of which use parallel BDD operations (i.e., Sylvan), on the BNN models P8 and P9 using the 0-image under Hamming distance $r = 2, 3, 4$. Note that r is limited up to 4, because when it is larger than 4, (i) the BDD encoding often runs out of time for both strategies, and (ii) for the BDD encoding that terminates within the time limit, the result of the comparison is similar to that of $r = 4$. Similarly, we skip $r = 1$ because the result of the comparison is similar to that of $r = 2$. On the other hand, we choose P8 and P9 as the subjects for evaluation, because (i) they are relatively larger and all the BDD encoding tasks with different number of workers terminate within the time limit (8 hours) when $r \leq 4$, and (ii) the experimental results on P8 and P9 can demonstrate the effectiveness of the divide-and-conquer strategy for different number of workers and input regions.

Table 6 reports the average execution time (in seconds) of L0-Iteration and L0-D&C, where the latter uses the divide-and-conquer strategy (cf. Table 4). We can observe that in this setting, our divide-and-conquer strategy is very effective, with more than 2.5 times speedup.

Figure 10 reports the average execution time (in seconds) of L1-Iteration and L1-D&C, where the numbers of workers are $1, 2, 4, 6, 8, \cdots, 36, 38, 39$. We can observe that our divide-and-conquer strategy is often very effective when parallel BDD operations are used, in particular for large

Table 7. Execution time (in seconds) of L2-D&C and L2-D&C-NoIP for BDD encoding of BNNs using the 0-image under Hamming distance r=2,3,4, where -TO- denotes time out (8 hours), -MO- denotes out of memory (256 GB) and s1, s10, s20, s39 indicate the number of workers.

Name	r	L2-D&C				L2-D&C-NoIP			
		s1	s10	s20	s39	s1	s10	s20	s39
P8	2	**0.42**	0.57	0.74	0.66	-TO-	-TO-	-MO-	-MO-
	3	1.17	**0.80**	0.98	0.83	-TO-	-TO-	-MO-	-MO-
	4	39.90	6.27	4.18	**2.87**	-TO-	-TO-	-MO-	-MO-
P9	2	**1.74**	1.77	1.83	2.09	-TO-	-TO-	-TO-	-TO-
	3	35.97	6.72	5.51	**4.04**	-TO-	-TO-	-TO-	-TO-
	4	603.6	79.76	52.97	**36.79**	-TO-	-TO-	-TO-	-TO-



(a) P1 with full input space (b) P3 with full input space (c) P4 with full input space

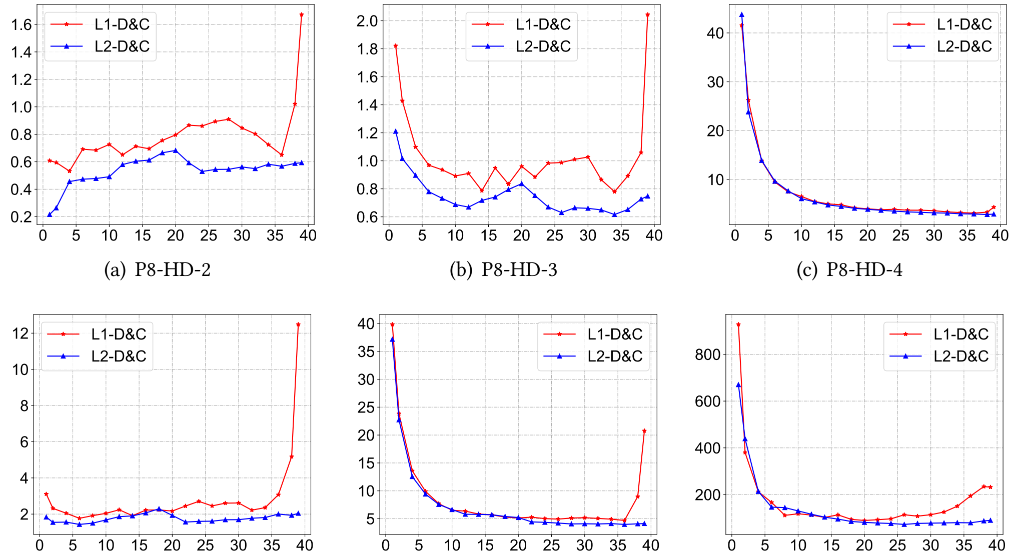
Fig. 11. Execution time (in seconds) of L2-D&C and L2-D&C-NoIP for BDD encoding of BNNs under full input space, where the numbers of workers (x-axis) are $1, 2, 4, 6, 8, \cdots, 36, 38, 39$.

Hamming distances (i.e., 3 and 4). However, with the increased number of workers, the performance of parallel BDD encoding downgraded, e.g., Figure 10(a), 10(b) and 10(d), due to the overhead induced by the synchronisation between workers. The divide-and-conquer strategy even becomes worse than the iteration-based one when the number of worker is very large (e.g., 39) in Figure 10(a) and 10(b), because the divide-and-conquer strategy requires additional AND-operations that induce synchronisation between workers.

6.1.3 Effectiveness of Input Propagation. To study the effectiveness of input propagation, we conduct experiments using L2-D&C and L2-D&C-NoIP on the BNN models P8 and P9 with the 0-image under the Hamming distances 2, 3 and 4, and on the BNN models P1, P3 and P4 with the 0-image under full input space. Recall that both L2-D&C and L2-D&C-NoIP use parallel BDD encoding of blocks and BDD operations, but only L2-D&C uses input propagation. Note that we evaluate on BNNs of varied sizes and input regions to thoroughly understand the effectiveness of the input propagation strategy which may depend upon both input regions and BNN sizes. Other BNN models are not considered due to the huge computational cost and current results are able to demonstrate the effectiveness of input propagation.

Table 7 shows the results of L2-D&C and L2-D&C-NoIP on the BNN models P8 and P9 under the Hamming distances 2, 3 and 4, where the number of workers is set to 1, 10, 20 and 39, and the best ones are highlighted in **boldface**. Note that -TO- denotes time out (8 hours) and -MO- denotes out of memory (256 GB). It is easy to see that our input propagation is very effective.

Figure 11 shows the results of L2-D&C and L2-D&C-NoIP on the small BNN models P1, P3 and P4 under the full input space, where the number of workers (x-axis) is set to $1, 2, 4, 6, 8, \cdots, 36, 38, 39$. Unsurprisingly, the effectiveness of input propagation varies with BNN models, due to the difference



(a) P8-HD-2

(b) P8-HD-3

(c) P8-HD-4

(d) P9-HD-2

(e) P9-HD-3

(f) P9-HD-4

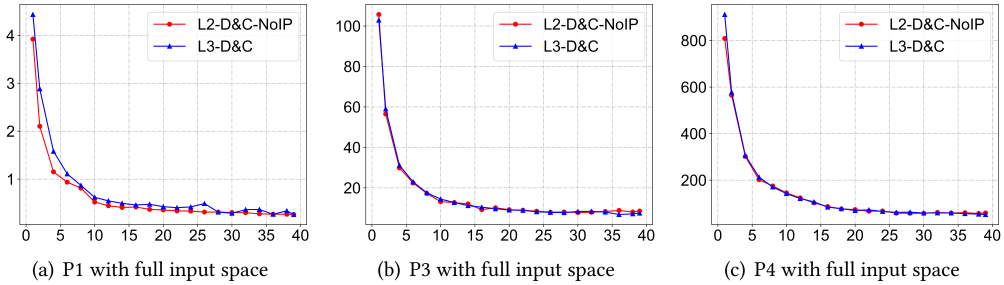
Fig. 12. Execution time (in seconds) of L1-D&C and L2-D&C for BDD encoding of BNNs, where the numbers of workers are 1, 2, 4, 6, 8, $\cdots$, 36, 38, 39.

of the architectures in P1, P3 and P4. Recall that the input propagation improves the encoding efficiency when the feasible outputs of the preceding block t_{i-1} (the input region when $i = 1$) is relatively small compared with the full input space $\mathbb{B}^{n_i}$ that the block t_i needs to consider (cf. Section 3.5). On P3 and P4, the feasible inputs of the internal blocks are close to their full input space, thus the input propagation does not make sense, and actually incurs overhead due to additional BDD operations (i.e., And and Exists).

In contrast to P3 and P4, P1 has an increase and then decrease in the number of hidden neurons of the layers, thus the input propagation is still effective on P1. Indeed, the input propagation reduces the full output space 2^{25} of the first block (i.e., the input space of the second block) to the feasible outputs 2^{16} (i.e., the input region). We remark that L2-D&C-NoIP quickly runs out of time on P2 when encoding the second block even with 39 workers due to 64 hidden neurons in its LIN layer, while it can be done in seconds when input propagation is enabled, i.e., L2-D&C. Thus, results on P2 are not depicted in Figure 11.

6.1.4 Effectiveness of parallelization strategies. By comparing the results between L0-Iteration and L1-Iteration (resp. L0-D&C and L1-D&C) on the BNN models P8 and P9 using the 0-image under Hamming distances 2, 3 and 4 (cf. Table 6 and Figure 10 in Section 6.1.2), we can observe that parallel BDD operations (i.e., Sylvan) is very effective for large input regions, (i.e., Hamming distance $r = 3, 4$). However, the improvement of sole parallel BDD operations may downgrade with the increased number of workers. Below we study the effectiveness of the other two parallelization strategies, namely, parallel BDD encoding of blocks (i.e., Algorithms 5 and 6) on P8 and P9, and parallel BDD encoding of an entire BNN (i.e., Algorithm 7) on P1, P3 and P4. Due to the similar reasons as mentioned above, we do not consider other BNN models.

To understand the effectiveness of parallel BDD encoding of blocks, we compare L1-D&C and L2-D&C on the BNN models P8 and P9 under Hamming distance $r = 2, 3, 4$. Recall that both L1-D&C and L2-D&C use parallel BDD operations, but only L2-D&C uses parallel BDD encoding of blocks.



(a) P1 with full input space (b) P3 with full input space (c) P4 with full input space

Fig. 13. BDD encoding under full input space with L2-D&C-NoIP and L3-D&C, where on the x-axis $(1, 2, \cdots, 40)$ denotes the number of workers, and y-axis denotes the computation time (seconds).

The results are shown in Figure 12. We can observe that our parallel BDD encoding of blocks is often very effective. Furthermore, increasing the number of workers in L2-D&C always improves the overall encoding efficiency for large BNNs and Hamming distance, e.g., Figure 12(c), Figure 12(e) and Figure 12(f).

To understand the effectiveness of parallel BDD encoding of entire BNNs, we compare L2-D&C-NoIP and L3-D&C on the relatively small BNN models P1, P3 and P4 under the full input region, Recall that both L2-D&C-NoIP and L3-D&C use parallel BDD operations and BDD encoding of blocks without input propagation, but only L3-D&C uses parallel BDD encoding of entire BNNs. The results are shown in Figure 13. The overall BDD encoding efficiency can be improved with the increased number of workers due to parallel BDD operations and BDD encoding of blocks, but the parallel BDD encoding of the entire BNNs did not improve the overall performance.

Summary of effectiveness of strategies. While L2-D&C with the largest number of workers (i.e., 39) does not always outputform the others, L2 (parallel BDD operations + parallel BDD encoding of blocks) always performs better than L1 (sole parallel BDD operations) and L3 (parallel BDD operations + parallel BDD encoding of blocks + parallel BDD construction of an entire BNN) (cf. Figure 12 and Figure 13 in Section 6.1.4). The divide-and-conquer strategy (i.e., D&C) performs better than the iteration-based one in most cases (cf. Table 6 and Figure 10 in Section 6.1.2), while it performs worse than the iteration-based one when the number of workers is very large (e.g., 39) and the input region is quite small (e.g., $r = 2$). On the other hand, the input propagation strategy can improve the encoding efficiency when the feasible output of the preceding block (or input region) is relatively small compared with the full input space of the encoding block (cf. Table 7 and Figure 11 in Section 6.1.3). Furthermore, the optimal number of workers for L2-D&C increases with the size of input region (cf. Table 7, Figure 11 and Figure 12). We summarize the findings as follows.

- When encoding an internal block, the divide-and-conquer strategy can improve the encoding efficiency in most cases;
- When encoding a block, the input propagation can improve the encoding efficiency when the feasible output of the preceding block (or input region) is relatively small compared with the full input space of the encoding block;
- L2 (i.e., parallel BDD operations + parallel BDD encoding of blocks) performs better than L1 (i.e., sole parallel BDD operations) and L3 (i.e., parallel BDD operations + parallel BDD encoding of blocks + parallel BDD construction of an entire BNN);
- The optimal number of workers for L2-D&C (i.e., L2 with both divide-and-conquer strategy and input propagation strategy) increases with the size of input region of interest.

Table 8. Execution time (in seconds) of L0-D&C and L2-D&C for BDD encoding of entire BNNs using full input space and number of BDD nodes, where sx denotes that number of workers is x.

Name	L0-D&C	L2-D&C		$\sum_{i\in[s]} \lvert G_i^{out}\rvert$
		s1	s39	
P1	1.20	0.76	**0.26**	16,834
P2	31.12	13.21	**0.96**	19,535
P3	425.2	207.2	**13.06**	5,071,376
P4	10,947	2,412	**110.8**	153,448,311
P5	-TO-	-TO-	-TO-	–

6.2 Performance of BDD Encoding for Entire BNNs

We evaluate BNNQuanalyst for BDD encoding of entire BNNs using three types of input regions: full input space, input regions based on Hamming distances and fixed indices. According to the results in Section 6.1, we only consider the sequential BDD encoding using CUDD (i.e., L0-D&C) and parallel BDD encoding using Sylvan (i.e., L2-D&C) with 1 worker (s1) and 39 workers (s39).

6.2.1 BDD Encoding using Full Input Space. We evaluate BNNQuanalyst on the BNNs (P1–P5) with the full input space. We remark that the other large BNNs (P6–P16) cannot be handled by BNNQuanalyst when the full input space is considered.

The average results of 10 images are reported in Table 8, where the best ones are highlighted in **boldface** and $\sum_{i\in[s]} |G_i^{out}|$ denotes the total number of nodes in BDDs $(G_i^{out})_{i\in[s]}$. BNNQuanalyst can construct the BDD models of P1–P4, but fails on P5 due to its large input size (i.e., 64). We can observe that both the execution time and the number of BDD nodes increase quickly with the size of BNNs. By comparing the results between L0-D&C and L2-D&C (s39), we confirm the effectiveness of our parallelization strategies for BDD operations and BDD encoding of internal and output blocks. Interestingly, L2-D&C (s1) also performs better than L0-D&C that uses CUDD.

6.2.2 BDD Encoding under Hamming Distance. We evaluate BNNQuanalyst on the relatively larger BNNs (P5–P16). For each BNN model, we consider all the 10 MNIST (resp. Fashion-MNIST) images shown in Figure 8 for P5–P12 (resp. P13–P16), and report average result of 10 images, excluding the computations that run out of time or memory. For each BNN model and image, the input regions are given by the image and Hamming distance $r = 2, 3, 4, 5$.

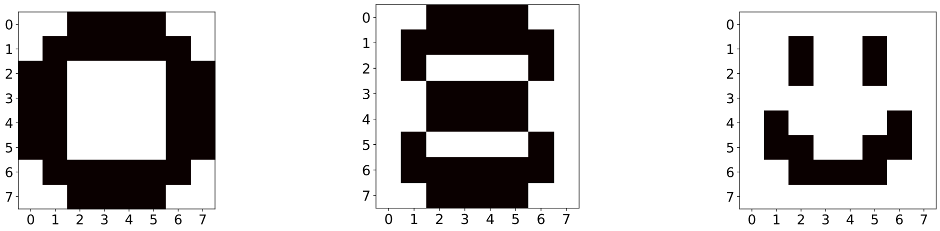
The results are shown in Table 9, where the best ones are highlighted in **boldface**. Overall, the execution time increases with the Hamming distance r. The comparison between L0-D&C and L2-D&C with 1 worker (s1) and 39 workers (s39) is summarized as follows:

- Both L0-D&C (i.e., columns cudd) and L2-D&C succeeded on all the cases when $r \leq 3$;
- L0-D&C succeeded on 108 out of 120 cases when $r = 4$, and 69 out of 120 cases when $r = 5$;
- L2-D&C (s1) succeeded on 112 out of 120 cases when $r = 4$, and 86 of 120 cases when $r = 5$;
- L2-D&C (s39) succeeded on 119 out of 120 cases when $r = 4$, and 95 of 120 cases when $r = 5$;
- For small-scale problems (i.e., P5 or r=2 or P12 with r=3), in most cases, L0-D&C and L2-D&C (s1) are almost comparable, but are better than L2-D&C (s39);
- For other problems, L2-D&C (s39) is much better than the other two.

Regarding the architecture of BNNs, we observe that the execution time increases with the number of hidden neurons (P6 vs. P7, P8 vs. P9 and P14 vs P15), while the effect of the number of layers is diverse (P6 vs. P8, P7 vs. P9 and P13 vs P15). From P11 and P12, we note that the number of hidden neurons per layer is likely the key impact factor of the BDD encoding efficiency. Interestingly, BNNQuanalyst works well on BNNs with large input sizes and large number of hidden layers (i.e., on P12).

Table 9. Execution time (in seconds) of L0-D&C and L2-D&C for BDD encoding of entire BNNs under Hamming distance, where (#*i*) (resp. [#*i*]) indicates the number of computations that run out of time in 8 hours (resp. memory in 256 GB).

Name	r=2			r=3			r=4			r=5		
	cudd	s1	s39	cudd	s1	s39	cudd	s1	s39	cudd	s1	s39
P5	**0.02**	0.04	0.41	**0.03**	0.05	0.63	**0.16**	0.17	0.69	1.03	**0.88**	0.98
P6	**0.35**	0.41	1.25	6.18	5.43	**2.01**	144.6	121.1	**9.43**	2,635	1,606	**84.48**
P7	0.80	**0.70**	1.17	29.35	32.75	**3.26**	964.0	778.0	**63.96**	(#2) 17,156	13,952	**733.8**
P8	0.27	**0.26**	1.11	3.47	2.75	**1.71**	94.80	61.44	**5.62**	1,715	821.6	**50.05**
P9	2.92	**2.11**	2.77	83.90	42.85	**5.94**	1,747	670.9	**44.66**	(#7) 20,851	(#3) 17,256	**[#2] 1,086**
P10	5.47	**5.19**	18.99	205.5	177.8	**30.26**	(#2) 10,791	(#2) 6,414	**401.3**	(#10)	(#4)[#6]	**[#8] 8,938**
P11	**22.14**	22.89	70.06	809.9	515.4	**102.2**	(#6) 11,203	(#4) 8,964	**[#1] 1,962**	(#10)	(#3)[#7]	**(#1)[#7] 5,702**
P12	9.82	**9.48**	35.73	10.92	**10.26**	44.65	73.81	40.49	**37.93**	2,659	746.8	**272.1**
P13	1.05	**0.84**	1.07	64.20	29.48	**2.18**	1,645	789.7	**36.56**	(#2) 24,501	15,648	**733.7**
P14	0.32	**0.29**	0.39	34.00	4.42	**0.56**	614.5	108.1	**7.00**	13,904	1,456	**97.24**
P15	1.82	**1.20**	1.23	345.3	35.47	**4.66**	10,283	686.2	**60.24**	[#10]	(#1) 11,761	**(#1) 1,621**
P16	24.46	**15.18**	43.28	929.8	429.4	**56.88**	(#4) 11,403	(#2) 8,268	**782.3**	(#10)	(#9)[#1]	**(#2)[#4] 14,561**



(a) A digit 0 image classified as '0' by the BNN P0

(b) A digit 8 image classified as '8' by the BNN P0

(c) A smile image classified as '0' by the BNN P0

Fig. 14. The three images of size 8 × 8 from [83].

Compared with BDD-learning-based method [83]. The results in Table 9 demonstrate the efficiency and scalability of BNNQuanalyst on BDD encoding of BNNs. Compared with the BDD-learning-based approach [83], our approach is considerably more efficient and scalable, since the BDD-learning-based approach takes 403 seconds to encode a BNN with input size 64, 5 hidden neurons, and output size 2 when r=6, while ours takes about 2 seconds (not list in the table) even for a larger BNN P5. To directly and fairly compare with this approach, we also conduct experiments on the BNN P0 provided in [83]. P0 is a binary classifier using the architecture 64:5:2, trained on the USPS digits dataset [46] for distinguishing digit 0 images (class 0) and digit 8 images (class 1) with 94% accuracy. The input regions of interest are given by three images and Hamming distance $r = 1, ..., 7$. The three input images of size 8×8 are provided by [83] and shown in Figure 14, where the smile image is classified as digit 0 by P0. We use L0-D&C and L2-D&C in BNNQuanalyst.

The results are reported in Table 10, where the BDD G produced by [83] encodes the input-output relation of P0 on the input region and both output classes, and the BDD G_i^{out} produced by BNNQuanalyst encodes the input-output relation of P0 on the input region and the class i for $i = 0, 1$. More specifically,

- any image $\mathbf{u}$ in the given input region is classified into the class 0 (resp. 1) by P0 iff $\mathbf{u} \notin \mathcal{L}(G)$ (resp. $\mathbf{u} \in \mathcal{L}(G)$), however, it is *not* guaranteed that images $\mathbf{u}$ outside of the given input region are classified into class 0 (resp. 1) by P0 iff $\mathbf{u} \notin \mathcal{L}(G)$ (resp. $\mathbf{u} \in \mathcal{L}(G)$).
- any image $\mathbf{u}$ in the given input region is classified into the class i by P0 iff $\mathbf{u} \in \mathcal{L}(G_i^{out})$, while for any image $\mathbf{u}$ outside the given input region, $\mathbf{u} \notin \mathcal{L}(G_i^{out})$.

Table 10. BDD encoding results of BNN P0 under different Hamming distances r for 3 input images.

r	Digit 0						Digit 8						Smile					
	[83]		BNNQuanalyst				[83]		BNNQuanalyst				[83]		BNNQuanalyst			
				Time(s)						Time(s)						Time(s)		
	$\|G\|$	Time(s)	$\|G_1^{out}\|$	cudd	s1	s39	$\|G\|$	Time(s)	$\|G_0^{out}\|$	cudd	s1	s39	$\|G\|$	Time(s)	$\|G_1^{out}\|$	cudd	s1	s39
1	1	0.18	1	**<0.01**	0.04	0.17	1	0.27	1	**<0.01**	0.04	0.13	1	0.19	1	**<0.01**	0.05	0.07
2	1	0.30	1	**<0.01**	0.04	0.14	1	0.38	1	**<0.01**	0.04	0.11	258	29.09	499	**0.01**	0.06	0.10
3	1	0.48	1	**<0.01**	0.05	0.16	1	0.63	1	**0.01**	0.04	0.16	1,437	430.4	1,621	**0.02**	0.07	0.14
4	1	1.54	1	**<0.01**	0.05	0.19	1	1.30	1	**0.01**	0.04	0.17	6,048	3,481	5,875	**0.04**	0.10	0.12
5	1	1.66	1	**<0.01**	0.05	0.19	243	120.8	546	**0.01**	0.06	0.12	12,297	25,645	13,127	0.09	0.13	**0.08**
6	509	436.7	718	**<0.01**	0.06	0.17	765	606.1	1,221	**0.02**	0.07	0.12	–	-TO-	31,067	0.25	0.22	**0.09**
7	2,202	2,210	2,437	**0.01**	0.06	0.21	2,431	3,231	2,907	**0.03**	0.08	0.10	–	-TO-	55,898	0.52	0.22	**0.19**

Therefore, the number of nodes in G may differ from one in G_i^{out} even for the same input region. We can observe that BNNQuanalyst significantly outperforms the BDD-learning-based method [83]. Interestingly, [83] takes more time on the smile image than the other two. It may be due to that the smile image is far away from the real digit 0.

6.2.3 BDD encoding under Fixed Indices. Similar to Section 6.2.2, we evaluate BNNQuanalyst on the relatively large BNNs (P5–P16) using 10 images for each BNN and report the average result of 10 images, excluding those that run out of time or memory. For each model and image, the input regions are given by the randomly chosen index set I whose size ranges from 10 to 25.

The results are shown in Table 11. We can observe similar results to the BDD encoding under Hamming distance. For instance, the execution time increases with the number of indices. For small-scale problems (i.e., $|I| = 10$) or P10–P12 and P16 with $|I| = 15$, L2-D&C (s1) often performs better than the other two. For other problems, L2-D&C (s39) often performs much better than the other two. The execution time increases with the number of hidden neurons (P6 vs. P7, P8 vs. P9, and P14 vs. P15). Remarkably, the execution time also increases with the number of layers (P6 vs. P8, P7 vs. P9, and P13 vs. P15), which is different from the results on the BDD encoding under Hamming distance.

6.3 Robustness Analysis

We evaluate BNNQuanalyst on the robustness of BNNs, including robustness verification under different Hamming distances and maximal safe Hamming distance computing.

6.3.1 Robustness verification with Hamming distance. We evaluate BNNQuanalyst on BNNs (P7, P8, P9, and P12) using 30 images from the MNIST dataset. Note that, besides the 10 digit images shown in Figure 8, we randomly choose another 20 images from the MNIST dataset. Other BNN models are not considered due to the huge computational cost and furthermore we will see later that the most computational cost of BNNQuanalyst is BDD encoding which has been extensively evaluated in Section 6.2. The input regions are given by the Hamming distance r ranging from 2 to 4, resulting in totally 360 verification tasks. We use L0-D&C, L2-D&C (s1) and L2-D&C (s39) in BNNQuanalyst.

Since the BDD-learning-based method [83] is significantly less efficient than our approach, we only compare with NPAQ [8]. The main difference between NPAQ and BNNQuanalyst is that NPAQ encodes a verification task as a Boolean formula without input propagation and uses an approximate SAT model-counting solver to answer the quantitative verification query with PAC-style guarantees while BNNQuanalyst encodes a verification task as BDDs with input propagation and uses the BDD operation SatCount to exactly answer the quantitative verification query. Namely, NPAQ sets a tolerable error ε and a confidence parameter δ. The final estimated results of NPAQ have the

Table 11. Execution time (in seconds) of L0-D&C and L2-D&C for BDD encoding of entire BNNs under fixed indices, where (#i) (resp. [#i]) indicates the number of computations that run out of time (resp. memory).

Name	$\|I\|$=10			$\|I\|$=15			$\|I\|$=20			$\|I\|$=25		
	cudd	s1	s39	cudd	s1	s39	cudd	s1	s39	cudd	s1	s39
P5	**0.03**	0.04	0.43	**0.05**	0.06	0.11	0.31	0.28	**0.16**	4.87	5.39	**0.59**
P6	0.24	**0.20**	0.67	1.20	0.79	**0.54**	43.38	27.68	**1.86**	844.0	577.4	**28.05**
P7	0.64	**0.43**	1.57	6.08	4.81	**2.21**	291.8	211.8	**12.07**	7,916	4,742	**219.4**
P8	0.25	**0.19**	0.81	1.51	0.96	**0.80**	37.56	21.99	**2.30**	1,074	818.3	**43.51**
P9	1.42	**0.85**	2.03	21.68	9.93	**2.52**	716.4	267.7	**17.49**	(#1) 18,143	7,739	**378.6**
P10	5.43	**4.04**	16.16	8.09	**5.92**	16.57	85.55	94.59	**21.31**	2,198	1,666	**99.86**
P11	21.62	**19.75**	60.08	24.03	**21.41**	60.67	73.40	65.85	**63.02**	1,460	1,120	**120.0**
P12	10.12	**8.40**	28.49	10.94	**8.73**	27.76	15.38	**11.88**	28.36	69.44	58.38	**30.71**
P13	0.57	**0.45**	1.03	7.49	3.48	**1.34**	386.43	182.9	**10.16**	15,616	5,538	**302.5**
P14	0.22	**0.21**	0.36	5.13	0.80	**0.49**	126.2	20.56	**2.30**	5,231	635.3	**48.26**
P15	1.30	**0.83**	1.38	66.60	19.06	**2.31**	1,983	426.6	**20.68**	(#9) 23,354	11,939	**485.4**
P16	26.42	**15.04**	44.75	29.27	**27.90**	43.04	166.8	73.63	**47.60**	5,564	1,951	**102.7**

bounded error ε with confidence of at least $1 - \delta$, i.e.,

$$Pr[(1+\varepsilon)^{-1} \times \texttt{RealNum} \leq \texttt{EstimatedNum} \leq (1+\varepsilon) \times \texttt{RealNum}] \geq 1 - \delta \quad (8)$$

In our experiments, we set $\varepsilon = 0.8$ and $\delta = 0.2$, as done by NPAQ [8].

Table 12 reports the results of the 294 instances (out of the 360 verification tasks) that can be solved by NPAQ, where the best ones are highlighted in **boldface**. Note that BNNQuanalyst solved all of them using either L0-D&C, L2-D&C (s1), or L2-D&C (s39). Columns (#Adv) give the average number of the adversarial examples for the 30 input images found by NPAQ and BNNQuanalyst. Columns (Time(s)) show the average execution time in seconds using NPAQ, L0-D&C, L2-D&C (s1) and L2-D&C (s39). Column (Diff #Adv) shows the error rate $\tau = \frac{\texttt{EstimatedNum}-\texttt{RealNum}}{\texttt{RealNum}}$ of NPAQ, where `RealNum` is from BNNQuanalyst, and `EstimatedNum` is from NPAQ. The last three columns show the speedups of BNNQuanalyst compared with NPAQ. Remark that the numbers of adversarial examples are 0 for P12 on input regions with $r = 3$ and $r = 4$ that can be solved by NPAQ. There do exist input regions for P12 that cannot be solved by NPAQ but have adversarial examples (see below). One may notice that L2-D&C (s1) performs better than L2-D&C (s39) on P12 for all the Hamming distances $r = 2, 3, 4$ which contradicts the results reported in Table 9. It is because Table 12 only shows the average execution time of the verification tasks that are successfully solved by NPAQ. Indeed, L2-D&C (39) performs better than L2-D&C (s1) on average when all the 30 verification tasks for P12 under $r = 4$ are considered.

The average number of BDD nodes in G_g^{out} and the average solving time by BNNQuanalyst are reported in Table 13, where g denotes the predicted class of the input image. Note that the average execution time reported in Table 12 includes the time used for BDD encoding and adversarial example counting while the average solving time reported in Table 13 only includes the time used for adversarial example counting. By comparing the execution time in Table 12 and solving time in Table 13, we can observe that most verification time of BNNQuanalyst is spent in BDD encoding. We also observe that while the number of BDD nodes blows up quickly with the input size of the BNN when the full input space is considered (cf. Table 8), it grows moderately when the input region is relatively small even for large BNNs (e.g., with Hamming distance $r = 2, 3, 4$), as shown in Table 13.

Compared with NPAQ. NPAQ runs out of time (8 hours) on 66 verification tasks (i.e., in P9 with $r = 4$ and P12 with $r = 2, 3, 4$), while BNNQuanalyst successfully verified all the 360 verification

Table 12. Robustness verification under Hamming distance, where (#*i*) indicates the number of verification tasks that run out of time whose execution time is not counted in the table.

Name	r	NPAQ [8]		BNNQuanalyst				Diff	Speedup		
					Time(s)						
		#Adv	Time(s)	#Adv	cudd	s1	s39	#Adv	cudd	s1	s39
P7	2	887	317.1	882	0.79	**0.76**	1.16	0.57%	400	**416**	272
	3	37,161	1,143	35,587	21.76	33.67	**2.43**	4.42%	52	33	**469**
	4	1,016,050	4,090	996,677	706.6	576.2	**38.05**	1.94%	5	6	**106**
P8	2	907	148.9	884	0.23	**0.21**	0.58	2.60%	646	**708**	256
	3	37,040	427.3	36,490	2.23	2.03	**0.89**	1.51%	191	209	**479**
	4	1,127,424	2,102	1,093,783	67.74	52.13	**3.82**	3.08%	30	39	**549**
P9	2	688	477.5	673	1.97	**1.63**	1.85	2.23%	241	**292**	257
	3	35,744	2,762	33,915	57.20	35.24	**4.03**	5.39%	47	77	**684**
	4	(#2) 898,528	5,197	869,715	1,285	672.4	**43.84**	3.31%	3	7	**118**
P12	2	(#14) 4,032	14,355	3,756	8.94	**8.55**	27.98	7.35%	1,605	**1,678**	512
	3	(#23) 0	20,029	0	10.73	**10.23**	44.27	0%	1,866	**1,957**	451
	4	(#27) 0	22,538	0	10.12	**9.43**	36.63	0%	2,226	**2,389**	614

Table 13. The average number of BDD nodes in G_g^{out} and average solving time of BNNQuanalyst w.r.t. Table 12.

Name	r	$\lvert G_g^{out}\rvert$	Solving Time (s)		
			cudd	s1	s39
P7	2	2,115	**<0.01**	**<0.01**	**<0.01**
	3	42,545	0.07	0.02	**<0.01**
	4	625,142	1.40	0.19	**0.08**
P8	2	2,022	**<0.01**	**<0.01**	**<0.01**
	3	40,139	0.11	**0.01**	**0.01**
	4	679,148	2.55	0.39	**0.14**
P9	2	2,345	**<0.01**	**<0.01**	**<0.01**
	3	55,208	0.16	0.03	**0.01**
	4	870,294	4.56	0.46	**0.16**
P12	2	5,793	0.01	**<0.01**	0.01
	3	3,125	**<0.01**	**<0.01**	**<0.01**
	4	3,901	**<0.01**	**<0.01**	**<0.01**

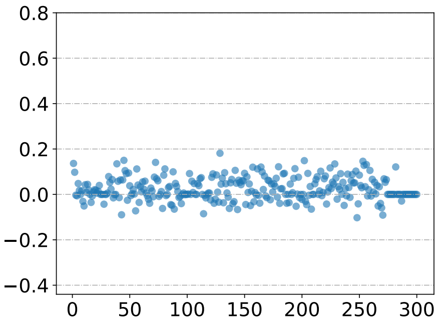
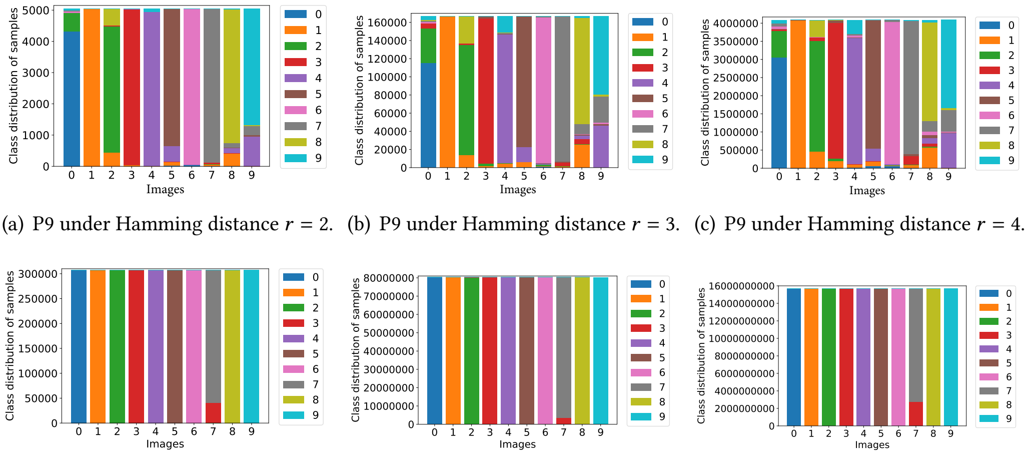


Fig. 15. Distribution of error rates of NPAQ.

tasks. On BNNs that were solved by both NPAQ and BNNQuanalyst, BNNQuanalyst is often significantly faster and more accurate than NPAQ. Specifically,

- L0-D&C (i.e., CUDD) is 3–2,226 times faster than NPAQ;
- L2-D&C (s1) is 6–2,389 times faster than NPAQ;
- L2-D&C (s39) is 106–684 times faster than NPAQ.

We remark that our input propagation plays a key role in outperforming NPAQ, otherwise the number of BDD nodes would blow up quickly with the number of hidden neurons per layer and input size (cf. Table 8). Recall that the input propagation is implemented using the Exists-operation and RelProd-operation whose worst-case time complexity is exponential in the number of Boolean variables, but they work well in our experiments. NPAQ encodes the entire BNN and input region into a Boolean formula without input propagation. However, it is non-trivial to directly perform input propagation at Boolean logic level for large Boolean formulas.



(a) P9 under Hamming distance $r = 2$. (b) P9 under Hamming distance $r = 3$. (c) P9 under Hamming distance $r = 4$.

(d) P12 under Hamming distance $r = 2$. (e) P12 under Hamming distance $r = 3$. (f) P12 under Hamming distance $r = 4$.

Fig. 16. Details of robustness verification on P9 and P12 with 10 images and Hamming distance $r = 2, 3, 4$.

Quality validation of NPAQ**.** Recall that NPAQ only supports approximate quantitative robustness verification with probably approximately correctness. Our exact approach can be used to validate the quality of such approximate approaches. Figure 15 shows the distribution of error rates of NPAQ on all the 294 instances, where the x-axis denotes the instances successfully verified by NPAQ, and the y-axis is the corresponding error rate of the verification result. According to Equation (8), the error rate τ should satisfy the following equation: $Pr(\tau \in [-0.44, 0.80]) \geq 0.8$. We can observe that the error rates of all the 294 instances fell in the range of $[-0.2, 0.2]$.

Details of robustness and targeted robustness. Figure 16 depicts the distributions of digits classified by the BNN models P9 and P12 under Hamming distance $r = 2, 3, 4$ using the ten MNIST images in Figure 8, where the x-axis $i = 0, \cdots, 9$ denotes the i-image. We can observe that P9 is robust for the 1-image when $r = 2$, but is not robust for the other images. (Note P9 is not robust for 1-image when $r \geq 3$, which is hard to be visualized in Figure 16(b) and Figure 16(c) due to the small number of adversarial examples.) Interestingly, most adversarial examples of the 0-image are misclassified into the digit 2, most adversarial examples of the 5-image and 9-image are misclassified into the digit 4, and most adversarial examples of the 8-image are misclassified into the digit 1. With the increase of the Hamming distance, more and more neighbors of the 7-image are misclassified into the digit 3. Though the BNN model P9 is not robust on most input images, indeed it is t-target-robust for many target digits t, e.g., P9 is t-target-robust for the 9-image when $t \leq 2$ and $r = 2$. We find that P12 is much more robust than P9, as P12 shows robustness for all cases except for the 7-image when $r = 2$, $\{2, 7\}$-images when $r = 3$, and $\{2, 7, 8, 9\}$-images when $r = 4$. Furthermore, we find that P12 is always t-target robust for all the images when $t \notin \{3, 4, 9\}$. (Note that similar to P9, the small number of adversarial examples of P12 for the 2-image when $r \geq 3$ and $\{8, 9\}$-images when $r = 4$ is also hard to visualize in Figures 16(e) and 16(f).)

6.3.2 Comparing with other possible approaches. Recall that NPAQ directly encodes a verification task into a Boolean formula in conjunctive normal forms (CNF) to which an approximate model-counting solver is applied, whereas BNNQuanalyst directly encodes a BNN under an input region with input propagation as a BDD to which the BDD operation SatCount is applied. Since Boolean formula and BDD are two exchangeable representations, two questions turn to be interesting: (i)

Table 14. Robustness verification under Hamming distance using different approaches, where -TO- (resp. -SO-) indicates that the computation runs out of time (8 hours) (resp. out of storage (100 GB) when saving the CNF formula $f_{G_g^{out}}$). N/A means that the data is not available.

Name	r	Encoding Results of Four Approaches										Solving Time(s)			
		BNN2CNF			CNF2BDD		BNN2BDD		BDD2CNF						
		#Vars	#Clauses	Time(s)	$\|G\|$	Time(s)	$\|G_g^{out}\|$	Time(s)	#Vars	#Clauses	Time(s)	f	G	G_g^{out}	$f_{G_g^{out}}$
P7	2	717,590	1,315,602	54.34	N/A	-TO-	2,994	0.79	100	140,753	0.42	310.5	N/A	<0.01	146.7
	3	717,623	1,315,731	56.03	N/A	-TO-	64,842	29.80	100	2,865,065	9.95	866.8	N/A	<0.01	9,182
	4	717,656	1,315,858	56.51	N/A	-TO-	1,143,994	1,024	100	44,645,415	170.3	3,899	N/A	4.92	-TO-
P8	2	337,646	601,556	44.88	N/A	-TO-	809	0.33	100	159,236	0.48	101.0	N/A	<0.01	191.0
	3	337,679	601,685	44.74	N/A	-TO-	10,613	2.16	100	3,837,306	13.77	549.6	N/A	0.02	15,892
	4	337,712	601,812	43.74	N/A	-TO-	363,343	77.58	100	68,637,805	273.5	2,755	N/A	1.11	-TO-
P9	2	908,082	1,638,938	62.04	N/A	-TO-	3,188	2.89	100	141,031	1.35	518.6	N/A	<0.01	317.1
	3	908,115	1,639,067	65.29	N/A	-TO-	83,677	82.13	100	2,897,549	16.46	1,136	N/A	0.20	-TO-
	4	908,147	1,639,194	60.12	N/A	-TO-	1,120,119	1,773	100	59,234,719	329.2	-TO-	N/A	4.18	-TO-
P12	2	5,756,197	9,533,668	78.20	N/A	-TO-	2,347	9.53	N/A	N/A	-SO-	6,169	N/A	<0.01	N/A
	3	5,756,460	9,534,711	77.13	N/A	-TO-	3,125	10.25	N/A	N/A	-SO-	17,199	N/A	<0.01	N/A
	4	5,756,751	9,535,753	73.19	N/A	-TO-	3,901	19.60	N/A	N/A	-SO-	-TO-	N/A	<0.01	N/A

how efficient if the BDD model is built from the Boolean formula generated by NPAQ and solved by the BDD operation SatCount? and (ii) how efficient if the Boolean formula is constructed from the BDD model generated by BNNQuanalyst and solved by applying an approximate model-counting solver?

To answer these questions, we implement two verification approaches, i.e., NPAQ2BDD (consisting of three sub-procedures: BNN2CNF + CNF2BDD + Solving) and BDD2NPAQ (consisting of three sub-procedures: BNN2BDD + BDD2CNF + Solving). We conduct experiments on the BNNs (P7, P8, P9 and P12) using the 0-image with Hamming distance $r = 2, 3, 4$ for computational cost consideration. The implementation details are as follows.

- BNN2CNF: We use NPAQ to encode a verification task into a Boolean formula f in CNF;
- CNF2BDD: Given a formula f produced by BNN2CNF, we encode all the clauses of f into BDDs based on which we compute the final BDD model G of f by applying the And-operations with a divide-and-conquer strategy, similar to Section 3.5.1;
- BNN2BDD: We use BNNQuanalyst with L0-D&C to build a BDD model G_g^{out} of a verification task, where g denotes the predicted class of the input image used for defining an input region;
- BDD2CNF: Given a BDD G_g^{out} produced by BNN2BDD, we encode it as a Boolean formula $f_{G_g^{out}}$ using the API (i.e., Dddmp_cuddBddStoreCnf) provided by CUDD;
- Solving: We use the approximate model-counting solver in NPAQ for solving Boolean formulas and use the BDD operation SatCount for solving all the final BDDs.

Note that we use L0-D&C (i.e., CUDD) only because Sylvan cannot produce a Boolean formula automatically, NPAQ is BNN2CNF + Solving and BNNQuanalyst is BNN2CNF + Solving.

Table 14 reports the results, where Columns (Time(s)) give the execution time for each sub-procedure in seconds, and the last four columns give the solving time. Columns 3-4 (resp. Columns 10-11) give the number of variables (#Vars) and clauses (#Clauses) of the Boolean formula f (resp. $f_{G_g^{out}}$) produced by BNN2CNF (resp. BDD2CNF), respectively. Columns ($|G|$) (resp. ($|G_g^{out}|$)) gives the number of BDD nodes of G (resp. G_g^{out}) produced by CNF2BDD (resp. BNN2BDD).

We can observe that the Boolean formulas produced by BNN2CNF are too large to be handled by CNF2BDD within the time limit (8 hours). Thus, NPAQ2BDD (i.e., BNN2BDD + BDD2CNF + Solving) is *not* realistic and building BDD model directly from BNNs (i.e., BNNQuanalyst) is better

Table 15. Maximal safe Hamming distance using Sylvan with 39 workers, where -TO- denotes time out (8 hours) and -MO- denotes out of memory (256 GB).

Image	P7				P8				P9				P12			
	$\epsilon = 0$		$\epsilon = 0.03$		$\epsilon = 0$		$\epsilon = 0.03$		$\epsilon = 0$		$\epsilon = 0.03$		$\epsilon = 0$		$\epsilon = 0.03$	
	SD	Time(s)	SD	Time(s)	SD	Time(s)	SD	Time(s)	SD	Time(s)	SD	Time(s)	SD	Time(s)	SD	Time(s)
0	0	4.11	0	4.25	1	2.00	3	9.13	0	5.97	0	6.31	6	-TO-	6	-TO-
1	1	4.32	2	3.78	0	1.93	0	2.15	2	5.80	5	-MO-	4	112.5	6	-TO-
2	0	2.74	0	3.33	0	2.14	1	3.22	0	5.75	0	5.57	2	43.77	6	-TO-
3	1	3.09	1	3.02	1	1.52	2	2.86	1	5.72	2	6.16	5	307.4	6	9,398
4	0	3.73	0	4.06	0	2.84	0	2.74	0	8.45	2	7.30	5	263.9	6	7,895
5	0	3.11	0	3.34	1	1.76	1	2.47	0	6.40	0	5.24	6	183.3	6	142.5
6	3	14.51	5	-TO-	1	2.05	2	2.34	1	7.30	2	6.76	6	625.2	6	681.4
7	0	3.83	0	4.25	1	1.57	2	2.46	1	7.66	2	7.28	1	38.87	1	30.28
8	0	4.28	0	4.82	0	3.25	0	4.14	0	8.25	0	8.98	3	100.9	6	-TO-
9	0	4.30	0	4.01	0	2.92	0	1.46	0	8.94	0	7.34	3	63.74	6	5,479

than building BDD model from Boolean formulas produced by NPAQ (i.e., NPAQ2BDD). On the other hand, though the number of Boolean variables of the Boolean formulas $f_{G_g^{out}}$ produced by BDD2CNF is fixed to the input size (100 for P7, P8 and P9, 784 for P12), the number of clauses of $f_{G_g^{out}}$ is often larger than that of the Boolean formulas f produced by BNN2CNF (except for P7, P8 and P9 with $r = 2$). In particular, BDD2CNF fails to transform BDDs into Boolean formulas on P12. Consequently, for hard verification tasks (i.e., large BNNs and input regions), building Boolean formulas directly from BNNs (i.e., NPAQ) is better than building Boolean formulas from the BDD models produced by BNN2BDD, and BDD2NPAQ performs significantly worse than NPAQ.

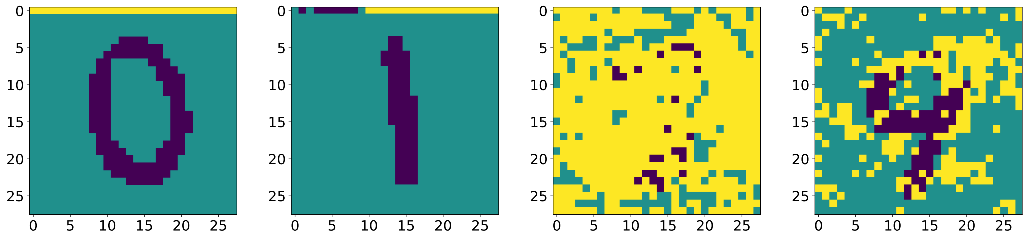
6.3.3 Maximal safe Hamming distance. As a representative of such an analysis, we evaluate our tool BNNQuanalyst on 4 BNNs (P7, P8, P9, and P12) with 10 images for 2 robustness thresholds ($\epsilon = 0$ and $\epsilon = 0.03$), and use s39 as our BDD encoding engine. The initial Hamming distance r is 3. Intuitively, $\epsilon = 0$ (resp. $\epsilon = 0.03$) means that up to 0% (resp. 3%) samples in the input region can be adversarial.

Table 15 shows the results, where columns (SD) and (Time(s)) give the maximal safe Hamming distance and the execution time, respectively. BNNQuanalyst solved 73 out of 80 instances. (For the remaining 7 instances, BNNQuanalyst ran 6 out of time and 1 out of memory, but was still able to compute a larger safe Hamming distance. For these cases, we only record the currently calculated result). We can observe that the maximal safe Hamming distance increases with the threshold ϵ on several BNNs and input regions. Moreover, P12 is more robust than the others, which is consistent with its highest accuracy (cf. Table 3).

6.4 Interpretability

We demonstrate the capability of BNNQuanalyst on interpretability using the BNN P12 and the $\{0, 1, 8, 9\}$-images because P12 has the largest input size 784 which makes the explanations more visual-friendly. We use $\mathcal{L}^x(G_y^{out})$ to denote the set of all inputs in the input region given by the x-image (based on Hamming distance or fixed indices) that are classified into the digit y.

PI-explanations. For demonstration, we assume that the input region is given by the fixed set $I = \{1, 2, \cdots, 28\}$ of indices which denotes the first row of pixels of 28×28 images. We compute two PI-explanations of the inputs $\mathcal{L}^0(G_0^{out})$ and $\mathcal{L}^1(G_4^{out})$ respectively. The PI-explanations are depicted in Figures 17(a) and 17(b), where the black (resp. blue) color denotes that the value of the corresponding pixel is 1 (resp. 0), and the yellow means that the value of the corresponding



(a) PI of 0-image to digit 0. (b) PI of 1-image to digit 4. (c) EFs of 8-image to digit 4. (d) EFs of 9-image to digit 4.

Fig. 17. Graphic representation of PI-explanations and essential features.

pixel can take arbitrary values. Figure 17(a) (resp. Figure 17(b)) suggests that, by the definition of the PI-explanation, all the images in the input region given by the 0-image (resp. 1-image) and I obtained by assigning arbitrary values to the yellow-colored pixels are always classified (resp. misclassified) into the digit 0 (resp. digit 4), while changing one black-colored or blue-colored pixel may change the predication result since a PI-explanation is a minimal set of literals. From Figure 17(a), we find that the first row of pixels has no influence on the prediction of the 0-image, which means no matter how we perturb the pixels from the first row of the 0-image, we can always get the same and correct prediction result by P12.

Essential features. For the input region given by the Hamming distance $r = 4$, we compute two sets of essential features for the inputs $\mathcal{L}^8(G_4^{out})$ and $\mathcal{L}^9(G_4^{out})$, i.e., the adversarial examples in the two input regions that are misclassified into the digit 4. The essential features are depicted in Figures 17(c) and 17(d). Recall that the black (resp. blue) color means that the value of the corresponding pixel is 1 (resp. 0), and the yellow color means that the value of the corresponding pixel can take arbitrary values. Figure 17(c) (resp. Figure 17(d)) indicates that the inputs $\mathcal{L}^8(G_4^{out})$ (resp. $\mathcal{L}^9(G_4^{out})$) must agree on these black- and blue-colored pixels.

7 RELATED WORK

In this section, we discuss the related work to BNNQuanalyst on qualitative/quantitative analysis and interpretability of DNNs. As there is a vast amount of literature regarding these topics, we will only discuss the most related ones.

Qualitative analysis of DNNs. For the verification of real-numbered DNNs, we broadly classify the existing approaches into three categories: (1) constraint solving based, (2) optimization-based, and (3) program analysis based.

The first class of approaches represents the early efforts which reduce to constraint solving. Pulina and Tacchella [78] verified whether the output of the DNN is within an interval by reducing to the satisfiability checking of a Boolean combination of linear arithmetic constraints which can be then solved via SMT solvers. However, their reduction yields an over-approximation of the original problem, thus may produce spurious adversarial examples. Spurious adversarial examples are used to trigger refinements to improve verification accuracy and retraining of DNNs to automate the correction of misbehaviors. Katz et al. [49] and Ehlers [28] independently implemented two SMT solvers, Reluplex and Planet, for exactly verifying properties of DNNs that are expressible with respective constraints. Recently, Reluplex was re-implemented in a new framework Marabou [50] with significant improvements.

For the second class of approaches which reduce to an optimization problem, Lomuscio and Maganti [60] verified whether some output is reachable from a given input region by reducing to mixed-integer linear programming (MILP) via optimization solvers. To speed up DNN verification via MILP solving, Cheng et al. [21] proposed heuristics for MILP encoding and parallelization of MILP solvers. Dutta et al. [25] proposed an algorithm to estimate the output region for a given input region which iterates between a global search with MILP solving and a local search with gradient descent. Tjeng et al. [97] proposed a tighter formulation for non-linearities in MILP and a novel presolve algorithm to improve performance. Recently, Bunel et al. [15] presented a branch and bound algorithm to verify DNNs on properties expressible in Boolean formulas over linear inequalities. They claimed that both previous SAT/SMT and MILP-based approaches are its special cases. Convex optimization has also been used to verify DNNs with over-approximations [26, 110, 112].

For the third class, researchers have adapted various methods from traditional static analysis to DNNs. A typical example is to use abstract interpretation, possibly aided with a refinement procedure to tighten approximations [2, 33, 38, 56, 57, 59, 86–88, 98, 99, 115]. These methods vary in the abstract domain (e.g., box, zonotope, polytope, and star-set), efficiency, precision, and activation functions. (Remark that [87, 88] considered floating-points instead of real numbers.) Another type is to compute convergent output bounds by exploring neural networks layer-by-layer. Huang et al. [44] proposed an exhaustive search algorithm with an SMT-based refinement. Later, the search problem was solved via Monte-Carlo tree search [109, 111]. Xiang et al. [108] proposed to approximate the bounds based on the linear approximations for the neurons and Lipschitz constants [40]. Wang et al. [104] presented symbolic interval analysis to tighten approximations. Recently, abstraction-based frameworks have been proposed [4, 29, 58, 121] which aim to reduce the size of DNNs, making them more amenable to verification.

Existing techniques for qualitative analysis of quantized DNNs are mostly based on constraint solving, in particular, SAT/SMT/MILP solving. SAT-based approaches transform BNNs into Boolean formulas, where SAT solving is harnessed [20, 52, 71, 72]. Following this line, verification of three-valued BNNs [47, 74] and quantized DNNs with multiple bits [9, 35, 41, 119] were also studied. Very recently, the SMT-based framework Marabou for real-numbered DNNs [50] has been extended to support partially or strictly binarized DNNs [1].

Quantitative analysis of DNNs. Comparing to the qualitative analysis, the quantitative analysis of neural networks is currently very limited. Two sampling-based approaches were proposed to certify the robustness of adversarial examples [7, 106], which require only blackbox access to the models, hence can be applied on both DNNs and BNNs. Yang et al. [115] proposed a spurious region-guided refinement approach for real-numbered DNN verification. The quantitative robustness verification is achieved by over-approximating the Lebesgue measure of the spurious regions. The authors claimed that it is the first work to quantitative robustness verification of DNNs with soundness guarantee.

Following the SAT-based qualitative analysis of BNNs [71, 72], SAT-based quantitative analysis approaches were proposed [8, 34, 73] for verifying robustness and fairness, and assessing heuristic-based explanations of BNNs. In particular, approximate SAT model-counting solvers are utilized. Though some of them provide probably approximately correct (PAC) style guarantees, tremendous verification cost has to pay to achieve higher precision and confidence. As demonstrated in Section 6, our BDD-based approach is considerably more accurate and efficient than the SAT-based one [8]. In general, we remark that the BDD construction is computationally expensive but the follow-up analysis is often much more efficient, while the SAT encoding is efficient (polynomial-time) but ♯SAT queries are often computationally expensive (♯P-hard). The computational cost of our approach is more dependent on the number of neurons per linear layer but less relevant to the number of layers (cf. Section 6.2.2), while the computational cost of the SAT-based approach [8] is dependent on both of them.

Shih et al. [83] proposed a BDD-based approach to tackle BNNs, similar to our work, in spirit. In this BDD-learning-based approach, membership queries are implemented by querying the BDD for each input, equivalence queries are implemented by transforming the BDD and BNN to two Boolean formulas, and checking the equivalence of two Boolean formulas under the input region (in a Boolean formula) via SAT solving. This construction requires n equivalence queries and $6n^2 + n \cdot \log(m)$ membership queries, where n (resp. m) is the number of nodes (resp. variables) in the final BDD. Due to the intractability of SAT solving (i.e., NP-complete), currently the technique is limited to quite toy BNNs.

Interpretability of DNNs. Though interpretability of DNNs is crucial for explaining predictions, it is very challenging to tackle due to the blackbox nature of DNNs. There is a large body of work on the interpretability of DNNs (cf. [43, 68] for a survey). One line of DNN interpretability is based on individual inputs which aims to give an explanation of the decision made for each given input. Bach et al. [5] proposed to compute the feature scores of an input via a layer-wise relevance backward propagation. Ribeiro et al. [81] proposed to learn a representative model locally for each input by approximating the classifier. Sundararajan et al. [93] proposed to compute an integrated gradient on each feature of the input which could be used to represent the contribution of that feature to prediction. Following this line, saliency map based methods were also proposed [22, 31, 84, 85, 89], varying in the way of computing the saliency map. Simonyan et al. [85] proposed to compute an image-specific saliency map for each class via a single gradient-based backward propagation. Later, Smilkov et al. [89] sharpened this map further by randomly perturbing the input with noises and then computing the average of resulting maps. Another line of DNN interpretability is to learn an interpretable model, such as binary decision trees [32, 117] and finite-state automata [107]. Then, an intuitive explanation could be obtained by directly querying these models.

Our interpretability analysis method is conducted by querying BDD models. In contrast to prior work which focuses on DNNs and only approximates the original model in the input region, we focus on BNNs and give a precise BDD encoding w.r.t. the given input region. The BDD encoding allows us to give a precise PI-explanation and essential feature analysis for an input region, which cannot be done on DNNs. Similar to ours, the BDD-learning based method [42] has also used PI-explanation for BNN interpretability, but the essential features were not studied therein.

8 CONCLUSION

In this article, we have proposed a novel BDD-based framework for the quantitative analysis of BNNs. The framework relies on the structure characterization of BNNs, and comprises a set of strategies such as input propagation, divide-and-conquer and parallelization at various levels to improve the overall encoding efficiency. We implemented our framework as a prototype tool BNNQuanalyst and conducted extensive experiments on BNN models with varying sizes and input regions trained on the popular dataset MNIST. Experimental results on quantitative robustness analysis demonstrated that BNNQuanalyst is more scalable than the existing BDD-learning based approach, and significantly efficient and accurate than the existing SAT-based approach NPAQ.

This work represents the first but a key step of the long-term program to develop an efficient and scalable BDD-based quantitative analysis framework for BNNs. For the future work, we plan to evaluate our framework on more applications, improve its encoding efficiency further and extend it to handle general quantized DNNs.

ACKNOWLEDGEMENT

This work is supported by the National Key Research Program (2020AAA0107800), the National Natural Science Foundation of China (NSFC) under Grants No. 62072309 and No. 61872340, an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03), and Birkbeck BEI School Project (EFFECT).

REFERENCES

[1] Guy Amir, Haoze Wu, Clark W. Barrett, and Guy Katz. 2020. An SMT-Based Approach for Verifying Binarized Neural Networks. *CoRR* abs/2011.02948 (2020).

[2] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 731–744.

[3] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106.

[4] Pranav Ashok, Vahid Hashemi, Jan Kretínský, and Stefanie Mohr. 2020. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis*. 92–107.

[5] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. 2015. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one* 10, 7 (2015), e0130140.

[6] Baidu. 2021. Apollo. https://apollo.auto.

[7] Teodora Baluta, Zheng Leong Chua, Kuldeep S. Meel, and Prateek Saxena. 2021. Scalable Quantitative Verification For Deep Neural Networks. In *Proceedings of th 43rd IEEE/ACM International Conference on Software Engineering*. 312–323.

[8] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S Meel, and Prateek Saxena. 2019. Quantitative verification of neural networks and its security applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1249–1264.

[9] Marek S. Baranowski, Shaobo He, Mathias Lechner, Thanh Son Nguyen, and Zvonimir Rakamaric. 2020. An SMT Theory of Fixed-Point Arithmetic. In *Proceedings of the 10th International Joint Conference on Automated Reasoning*. 13–31.

[10] Constantinos Bartzis and Tevfik Bultan. 2003. Construction of efficient BDDs for bounded arithmetic constraints. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 394–408.

[11] Alan C. Bovik. 2009. *The essential guide to image processing*. Elsevier.

[12] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* 35, 8 (1986), 677–691.

[13] Randal E. Bryant. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.* 24, 3 (1992), 293–318.

[14] Lei Bu, Zhe Zhao, Yuchao Duan, and Fu Song. 2021. Taking Care of The Discretization Problem: A Comprehensive Study of the Discretization Problem and A Black-Box Adversarial Attack in Discrete Integer Domain. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–18. https://doi.org/10.1109/TDSC.2021.3088661

[15] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. 2020. Branch and Bound for Piecewise Linear Neural Network Verification. *J. Mach. Learn. Res.* 21 (2020), 42:1–42:39.

[16] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. 39–57.

[17] Guangke Chen, Sen Chen, Lingling Fan, Xiaoning Du, Zhe Zhao, Fu Song, and Yang Liu. 2021. Who is Real Bob? Adversarial Attacks on Speaker Recognition Systems. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*.

[18] Guangke Chen, Zhe Zhao, Fu Song, Sen Chen, Lingling Fan, and Yang Liu. 2021. SEC4SR: A Security Analysis Platform for Speaker Recognition. *CoRR* abs/2109.01766 (2021).

[19] Guangke Chen, Zhe Zhao, Fu Song, Sen Chen, Lingling Fan, and Yang Liu. 2022. AS2T: Arbitrary Source-To-Target Adversarial Attack on Speaker Recognition Systems. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–17. https://doi.org/10.1109/TDSC.2022.3189397

[20] Chih-Hong Cheng, Georg Nührenberg, Chung-Hao Huang, and Harald Ruess. 2018. Verification of Binarized Neural Networks via Inter-neuron Factoring - (Short Paper). In *Proceedings of the 10th International Conference on Verified Software. Theories, Tools, and Experiments*. 279–290.

[21] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum Resilience of Artificial Neural Networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 251–268.

[22] Piotr Dabkowski and Yarin Gal. 2017. Real time image saliency for black box classifiers. *Advances in neural information processing systems* 30 (2017).

[23] Nilesh N. Dalvi, Pedro M. Domingos, Mausam, Sumit K. Sanghai, and Deepak Verma. 2004. Adversarial classification. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 99–108.

[24] Elvis Dohmatob. 2018. Limitations of adversarial robustness: strong No Free Lunch Theorem. *CoRR* abs/1810.04065 (2018).

[25] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *Proceedings of the 10th International Symposium NASA Formal Methods (NFM)*. 121–138.

[26] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A. Mann, and Pushmeet Kohli. 2018. A Dual Approach to Scalable Verification of Deep Networks. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence*. 550–559.

[27] Niklas Eén and Niklas Sörensson. 2006. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1-4 (2006), 1–26.

[28] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis*. 269–286.

[29] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. 2020. An Abstraction-Based Framework for Neural Network Verification. In *Proceedings of the 32nd International Conference on Computer Aided Verification*. 43–65.

[30] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proceedings of 2018 IEEE Conference on Computer Vision and Pattern Recognition*. 1625–1634.

[31] Ruth C Fong and Andrea Vedaldi. 2017. Interpretable explanations of black boxes by meaningful perturbation. In *Proceedings of the IEEE international conference on computer vision*. 3429–3437.

[32] Nicholas Frosst and Geoffrey E. Hinton. 2017. Distilling a Neural Network Into a Soft Decision Tree. In *Proceedings of the 1st International Workshop on Comprehensibility and Explanation in AI and ML*.

[33] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. 3–18.

[34] Bishwamittra Ghosh, Debabrota Basu, and Kuldeep S. Meel. 2020. Justicia: A Stochastic SAT Approach to Formally Verify Fairness. *CoRR* abs/2009.06516 (2020).

[35] Mirco Giacobbe, Thomas A. Henzinger, and Mathias Lechner. 2020. How Many Bits Does it Take to Quantize Your Neural Network?. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 79–97.

[36] Justin Gilmer, Nicolas Ford, Nicholas Carlini, and Ekin D. Cubuk. 2019. Adversarial Examples Are a Natural Consequence of Test Error in Noise. In *Proceedings of the 36th International Conference on Machine Learning*. 2280–2289.

[37] Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S. Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian J. Goodfellow. 2018. Adversarial Spheres. In *Proceedings of the 6th International Conference on Learning Representations*.

[38] Xingwu Guo, Wenjie Wan, Zhaodi Zhang, Min Zhang, Fu Song, and Xuejun Wen. 2021. Eager Falsification for Accelerating Robustness Verification of Deep Neural Networks. In *Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering*. 345–356.

[39] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning*. 1737–1746.

[40] Matthias Hein and Maksym Andriushchenko. 2017. Formal Guarantees on the Robustness of a Classifier against Adversarial Manipulation. In *Proceedings of the Annual Conference on Neural Information Processing Systems*. 2266–2276.

[41] Thomas A Henzinger, Mathias Lechner, and Dorde Žikelić. 2020. Scalable Verification of Quantized Neural Networks (Technical Report). *arXiv preprint arXiv:2012.08185* (2020).

[42] Hao Hu, Marie-José Huguet, and Mohamed Siala. 2022. Optimizing Binary Decision Diagrams with MaxSAT for classification. (2022).

[43] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. 2020. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review* 37 (2020), 100270.

[44] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*. 3–29.

[45] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems*. 4107–4115.

[46] Jonathan J. Hull. 1994. A Database for Handwritten Text Recognition Research. *IEEE Trans. Pattern Anal. Mach. Intell.* 16, 5 (1994), 550–554. https://doi.org/10.1109/34.291440

[47] Kai Jia and Martin Rinard. 2020. Efficient Exact Verification of Binarized Neural Networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems.*

[48] Nidhi Kalra and Susan M Paddock. 2016. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice* 94 (2016), 182–193.

[49] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings of the 29th International Conference on Computer Aided Verification.* 97–117.

[50] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proceedings of the 31st International Conference on Computer Aided Verification.* 443–452.

[51] Philip Koopman and Beth Osyk. 2019. Safety argument considerations for public road testing of autonomous vehicles. *SAE International Journal of Advances and Current Practices in Mobility* 1 (2019), 512–523.

[52] Svyatoslav Korneev, Nina Narodytska, Luca Pulina, Armando Tacchella, Nikolaj Bjørner, and Mooly Sagiv. 2018. Constrained Image Generation Using Binarized Neural Networks with Decision Procedures. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing.* 438–449.

[53] Jaeha Kung, David C. Zhang, Gooitzen S. van der Wal, Sek M. Chai, and Saibal Mukhopadhyay. 2018. Efficient Object Detection Using Embedded Binarized Neural Networks. *Journal of Signal Processing Systems* 90, 6 (2018), 877–890.

[54] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *Proceedings of International Conference on Learning Representations.*

[55] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database.

[56] Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. 2019. Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification. In *Proceedings of the 26th International Symposium on Static Analysis (SAS).* 296–319.

[57] Renjue Li, Jianlin Li, Cheng-Chao Huang, Pengfei Yang, Xiaowei Huang, Lijun Zhang, Bai Xue, and Holger Hermanns. 2020. PRODeep: a platform for robustness verification of deep neural networks. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1630–1634.

[58] Jiaxiang Liu, Yunhan Xing, Xiaomu Shi, Fu Song, Zhiwu Xu, and Zhong Ming. 2022. Abstraction and Refinement: Towards Scalable and Exact Verification of Neural Networks. *CoRR* abs/2207.00759 (2022). https://doi.org/10.48550/arXiv.2207.00759

[59] Wan-Wei Liu, Fu Song, Tang-Hao-Ran Zhang, and Ji Wang. 2020. Verifying ReLU Neural Networks from a Model Checking Perspective. *Journal of Computer Science and Technology* 35, 6 (2020), 1365–1381.

[60] Alessio Lomuscio and Lalit Maganti. 2017. An approach to reachability analysis for feed-forward ReLU neural networks. *CoRR* abs/1706.07351 (2017).

[61] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. 2014. A Thread-Safe Library for Binary Decision Diagrams. In *Proceedings of the 12th International Conference on Software Engineering and Formal Methods.* 35–49.

[62] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 120–131.

[63] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 175–186.

[64] Saeed Mahloujifar, Dimitrios I. Diochnos, and Mohammad Mahmoody. 2019. The Curse of Concentration in Robust Learning: Evasion and Poisoning Attacks from Concentration of Measure. In *Proceedings of the AAAI Conference on Artificial Intelligence.* 4536–4543.

[65] Bradley McDanel, Surat Teerapittayanon, and H. T. Kung. 2017. Embedded Binarized Neural Networks. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks.* 168–173.

[66] Kenneth L. McMillan. 1993. *Symbolic model checking.* Kluwer. https://doi.org/10.1007/978-1-4615-3190-6

[67] Shin-Ichi Minato and Fabio Somenzi. 1997. Arithmetic boolean expression manipulator using BDDs. *Formal methods in system design* 10, 2 (1997), 221–242.

[68] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. 2020. Interpretable Machine Learning - A Brief History, State-of-the-Art and Challenges. *CoRR* abs/2010.09337 (2020).

[69] Laurence Moroney. 2021. Introduction to tensorflow for artificial intelligence, machine learning, and deep learning. https://www.coursera.org/learn/introduction-tensorflow.

[70] Atsuyoshi Nakamura. 2005. An efficient query learning algorithm for ordered binary decision diagrams. *Information and Computation* 201, 2 (2005), 178–198.

[71] Nina Narodytska. 2018. Formal Analysis of Deep Binarized Neural Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 5692–5696.

[72] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2018. Verifying Properties of Binarized Deep Neural Networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 6615–6624.

[73] Nina Narodytska, Aditya A. Shrotri, Kuldeep S. Meel, Alexey Ignatiev, and João Marques-Silva. 2019. Assessing Heuristic Machine Learning Explanations with Model Counting. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing*. 267–278.

[74] Nina Narodytska, Hongce Zhang, Aarti Gupta, and Toby Walsh. 2020. In Search for a SAT-friendly Binarized Neural Network Architecture. In *Proceedings of the 8th International Conference on Learning Representations*.

[75] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks against Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 506–519.

[76] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of IEEE European Symposium on Security and Privacy*. 372–387.

[77] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[78] Luca Pulina and Armando Tacchella. 2010. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*. 243–257.

[79] Chongli Qin, Krishnamurthy (Dj) Dvijotham, Brendan O'Donoghue, Rudy Bunel, Robert Stanforth, Sven Gowal, Jonathan Uesato, Grzegorz Swirszcz, and Pushmeet Kohli. 2019. Verification of Non-Linear Specifications for Neural Networks. In *Proceedings of the 7th International Conference on Learning Representations*.

[80] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Proceedings of the 14th European Conference on Computer Vision*. 525–542.

[81] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.

[82] Dinggang Shen, Guorong Wu, and Heung-Il Suk. 2017. Deep learning in medical image analysis. *Annual Review of Biomedical Engineering* 19 (2017), 221–248.

[83] Andy Shih, Adnan Darwiche, and Arthur Choi. 2019. Verifying binarized neural networks by angluin-style learning. In *Proceedings of the 2019 International Conference on Theory and Applications of Satisfiability Testing*. 354–370.

[84] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *International conference on machine learning*. PMLR, 3145–3153.

[85] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *In Workshop at International Conference on Learning Representations*. Citeseer.

[86] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin T. Vechev. 2019. Beyond the Single Neuron Convex Barrier for Neural Network Certification. In *Proceedings of the Annual Conference on Neural Information Processing Systems*. 15072–15083.

[87] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. 2018. Fast and Effective Robustness Certification. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*. 10825–10836.

[88] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages (POPL)* 3 (2019), 41:1–41:30.

[89] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. 2017. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825* (2017).

[90] Fabio Somenzi. 2015. CUDD: CU Decision Diagram Package Release 3.0.0.

[91] Fu Song, Yusi Lei, Sen Chen, Lingling Fan, and Yang Liu. 2021. Advanced evasion attacks and mitigations on practical ML-based phishing website classifiers. *Int. J. Intell. Syst.* 36, 9 (2021), 5210–5240.

[92] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 109–119.

[93] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic Attribution for Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning*. 3319–3328.

[94] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *Proceedings of International Conference on Learning Representations*.

[95] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*. 6105–6114.

[96] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. 303–314.

[97] Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2019. Evaluating Robustness Of Neural Networks With Mixed Integer Programming. In *Proceedings of the 7th International Conference on Learning Representations*.

[98] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. 2020. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proceedings of the 32nd International Conference on Computer Aided Verification*. 18–42.

[99] Hoang-Dung Tran, Diego Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. 2019. Star-Based Reachability Analysis of Deep Neural Networks. In *Proceedings of the 3rd World Congress on Formal Methods*. 670–686.

[100] Jonathan Uesato, Brendan O'Donoghue, Pushmeet Kohli, and Aäron van den Oord. 2018. Adversarial Risk and the Dangers of Evaluating Against Weak Attacks. In *Proceedings of the 35th International Conference on Machine Learning*. 5032–5041.

[101] Tom van Dijk, Ernst Moritz Hahn, David N. Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. 2015. A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking. In *Proceedings of the 1st International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. 35–51.

[102] Tom van Dijk and Jaco van de Pol. 2015. Sylvan: Multi-core decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 677–691.

[103] Tom van Dijk and Jaco C. van de Pol. 2014. Lace: Non-blocking Split Deque for Work-Stealing. In *Proceedings of the International Workshops on Parallel Processing*. 206–217.

[104] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proceedings of the 27th USENIX Security Symposium*. 1599–1614.

[105] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2021. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification. In *Proceedings of the Annual Conference on Neural Information Processing Systems*. 29909–29921.

[106] Stefan Webb, Tom Rainforth, Yee Whye Teh, and M. Pawan Kumar. 2019. A Statistical Approach to Assessing Neural Network Robustness. In *Proceedings of the 7th International Conference on Learning Representations*.

[107] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *Proceedings of the 35th International Conference on Machine Learning*. 5244–5253.

[108] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. 2018. Towards Fast Computation of Certified Robustness for ReLU Networks. In *Proceedings of the 35th International Conference on Machine Learning*. 5273–5282.

[109] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 408–426.

[110] Eric Wong and J. Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning*. 5283–5292.

[111] Min Wu, Matthew Wicker, Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2020. A game-based approximate verification of deep neural networks with provable guarantees. *Theoretical Computer Science* 807 (2020), 298–329.

[112] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. 2018. Output Reachable Set Estimation and Verification for Multilayer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 29, 11 (2018), 5777–5783.

[113] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[114] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[115] Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. 2020. Improving Neural Network Verification through Spurious Region Guided Refinement. *CoRR* abs/2010.07722 (2020).

[116] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Trans. Software Eng.* 48, 2 (2022), 1–36.

[117] Quanshi Zhang, Yu Yang, Haotian Ma, and Ying Nian Wu. 2019. Interpreting CNNs via Decision Trees. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6261–6270.

[118] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, and Taolue Chen. 2021. BDD4BNN: A BDD-Based Quantitative Analysis Framework for Binarized Neural Networks. In *Proceedings of the 33rd International Conference on Computer Aided Verification*. 175–200.

[119] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, Min Zhang, and Taolue Chen. 2022. QVIP: An ILP-based Formal Verification Approach for Quantized Neural Networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.

[120] Zhe Zhao, Guangke Chen, Jingyi Wang, Yiwei Yang, Fu Song, and Jun Sun. 2021. Attack as defense: characterizing adversarial examples using robustness. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 42–55.

[121] Zhe Zhao, Yedi Zhang, Guangke Chen, Fu Song, Taolue Chen, and Jiaxiang Liu. [n.d.]. CENTRAL: Accelerating CEGAR-based Neural Network Verification via Adversarial Attacks, booktitle = Proceedings of the 29th Static Analysis Symposium, year = 2022.