Adaptive Branch-and-Bound Tree Exploration for Neural Network Verification

Kota Fukuda*, Guanqin Zhang†, Zhenya Zhang*, Yulei Sui†, Jianjun Zhao*
*Kyushu University, Fukuoka, Japan
†University of New South Wales, Sydney, Australia

***Abstract*—Formal verification is a rigorous approach that can provably ensure the quality of neural networks, and to date, Branch and Bound (BaB) is the state-of-the-art that performs verification by splitting the problem as needed and applying off-the-shelf verifiers to sub-problems for improved performance. However, existing BaB may not be efficient, due to its naive way of exploring the space of sub-problems that ignores the *importance* of different sub-problems. To bridge this gap, we first introduce a notion of "importance" that reflects how likely a counterexample can be found with a sub-problem, and then we devise a novel verification approach, called ABONN, that explores the sub-problem space of BaB adaptively, in a Monte-Carlo tree search (MCTS) style. The exploration is guided by the "importance" of different sub-problems, so it favors the sub-problems that are more likely to find counterexamples. As soon as it finds a counterexample, it can immediately terminate; even though it cannot find, after visiting all the sub-problems, it can still manage to verify the problem. We evaluate ABONN with 552 verification problems from commonly-used datasets and neural network models, and compare it with the state-of-the-art verifiers as baseline approaches. Experimental evaluation shows that ABONN demonstrates speedups of up to $15.2\times$ on MNIST and $24.7\times$ on CIFAR-10. We further study the influences of hyperparameters to the performance of ABONN, and the effectiveness of our adaptive tree exploration.**

***Index Terms*—neural network verification, branch and bound, Monte-Carlo tree search, counterexample potentiality**

I. Introduction

Recently, artificial intelligence (AI) has experienced an explosive development and pushes forward the state-of-the-art in various domains. Due to their advantages in handling complex data (e.g., images and natural languages), AI products, especially *deep neural networks*, have been deployed in different safety-critical systems, such as autonomous driving and medical devices. Despite such prosperity, a surge of concerns also arise about their safety, because misbehavior of those systems can pose severe threats to human society [1]. Given that neural networks are notoriously vulnerable to adversarial perturbations [2], it is of great significance to ensure their quality before their deployment in real world.

As a rigorous approach, formal verification has been actively studied in recent years [1], [3], which can provably ensure the quality of neural networks. A straightforward approach [4], [5] is by encoding a verification problem as logical constraints which can be solved by mixed integer linear programming (MILP). However, due to the non-linearity of activation functions, this approach is not scalable to large networks. Abstraction-based approaches [6]–[9], that over-approximate the output regions of networks, are much faster; however, they suffer from an *incompleteness issue* and may often raise *false alarms* that are misleading, with spurious counterexamples.

Branch and Bound (`BaB`) [10] is an advanced verification approach for neural networks that can mitigate the incompleteness issue of approximated verifiers. It is essentially a "*divide-and-conquer*" strategy, which splits a problem if false alarms arise and applies approximated verifiers to the sub-problems for which those verifiers suffer less from the incompleteness issue. After verifying all the sub-problems or encountering a real counterexample in a sub-problem, `BaB` can conclude the verification and return accordingly. In this way, it overcomes the incompleteness issue in directly applying approximated verifiers to the original verification problem, thus more effective.

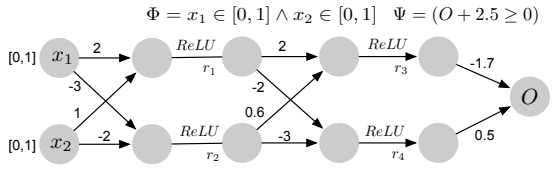
Motivation. While `BaB` has shown great promise, it may not be efficient, because it explores the space of sub-problems naively in a "*breadth-first*" manner and ignores the different "*importance*" of different sub-problems. Indeed, different sub-problems should have different priorities in verification, because with some sub-problems, it is more likely to find counterexamples, and thereby conclude the verification efficiently. By prioritizing these more important sub-problems, we can save plenty of efforts in checking unnecessary sub-problems and improve the efficiency of verification.

Contributions. In this paper, we propose a novel verification approach ABONN, that is, **A**daptive **B**aB with **O**rder for **N**eural **N**etwork verification, which incorporates the insights above.

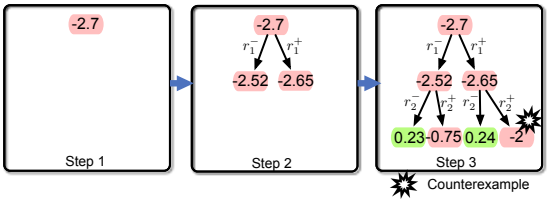
We first introduce a notion of "*importance*" over different sub-problems, called *counterexample potentiality*, used to characterize the likelihood of finding counterexamples with a sub-problem. It is defined based on two attributes of a sub-problem, including its fineness (i.e., the level of problem splitting) and a quantity returned by approximated verifiers that signify the level of specification violation of the sub-problem.

We then devise ABONN, which features an adaptive exploration in the space of sub-problems in a *Monte-Carlo tree search (MCTS)* [11] style. Guided by the counterexample potentiality of different sub-problems, ABONN favors the sub-problems that are more likely to find counterexamples, and as soon as it finds a counterexample, it can immediately terminate and report a negative result about the verification problem. Even though it cannot find such a counterexample, after visiting all the sub-problems, it can still manage to verify the problem.

We perform a comprehensive experimental evaluation for ABONN, with 552 verification problems that span over five neural network models in datasets MNIST and CIFAR-10,



(a) A neural network N and its specification



(b) A running example of BaB for verifying N

Fig. 1: Neural network verification problem and a solution via branch and bound (BaB).

and two state-of-the-art verification approaches as baselines. Our evaluation shows a great speedup of ABONN over the baseline approaches, for up to 15.2× in MNIST, and up to 24.7× in CIFAR-10. We further study the influences of hyperparameters, thereby exhibiting the necessity of striking a balance between "exploration" and "exploitation" in MCTS. Moreover, we demonstrate the effectiveness of our adaptive tree exploration strategy by separating the results for violated problems and certified problems.

II. Overview of The Proposed Approach

A. Verification Problem and BaB Approach

A neural network N and its associated specification are given in Fig. 1. The verification problem aims to determine whether the neural network N satisfies the specification, i.e., for all inputs $(x_1, x_2) \in [0,1] \times [0,1]$, whether it holds that the output O of N satisfies a logical constraint $O + 2.5 \geq 0$.

Fig. 1b illustrates how BaB [10] works to solve the problem. BaB splits the problem and applies verifiers to sub-problems to pursue better performance. It decides whether a (sub-)problem should be split, by applying an approximated verifier to it. The verifier can return a value, as depicted in each node in Fig. 1b, which signifies *how far* each problem is from being violated. If the value is positive, then the problem is verified and no need to split. Otherwise, BaB will check whether the negative value returned by the verifier is a false alarm, by validating the *counterexample* given by the verifier. If the counterexample is a real one, the verification can be concluded that the specification is violated. Otherwise, BaB will split the problem and apply verifiers to sub-problems. Problem splitting can be done in different ways; in this paper, we follow existing literature [10], [12] and impose input constraints to ReLU activation functions.

The running example in Fig. 1b works as follows:

1) By applying a verifier to the original problem, BaB obtains a negative returned value -2.7, and identifies that this is a false alarm. So, it splits the problem to two sub-problems;
2) BaB then applies the verifier to each of the two sub-problems, by which it obtains -2.52 and -2.65, respectively. Again, it identifies that both are false alarms, and so, it continues to split the sub-problems;
3) This continues in the subsequent layer of the tree, and it identifies the sub-problems that are verified (with values 0.23 and 0.24) and that need to be split further (with values -0.75). Notably, the verification can be terminated in this layer, because a real counterexample is detected in the sub-problem with -2, which can serve as an evidence that the network does not satisfy the specification.

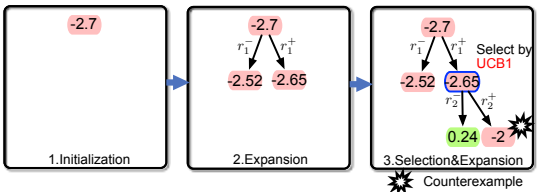


Fig. 2: ABONN's process for solving the problem.

B. The Proposed Approach

While BaB in Fig. 1b is effective to solve the problem, it may be not efficient, because it ignores the information about *how important* each sub-problem is. Indeed, different sub-problems are not equivalent in terms of their importance, and in particular, the exploration of the BaB tree should favor the sub-problems that are more likely to find a counterexample, because once that is achieved, verification can be early terminated.

Moreover, the information about the likelihood of finding counterexample has been embedded in the attributes of the nodes in the BaB tree. For instance, among the children of the root, the node with -2.65 should be prioritized than the node with -2.52, because according to such an assessment done by approximated verifiers, the former involves a sub-problem that is farther from being verified than the latter, thus more likely to contain a real counterexample.

Having these insights, we propose ABONN, that explores the space of sub-problems adaptively, in a *Monte-Carlo tree search (MCTS)* fashion. MCTS [11] is a *reinforcement learning*-based algorithm that expands a search tree guided by *rewards*, and so, it pays more efforts to the branches that are more "*promising*"; meanwhile, to avoid being too greedy, it also allocates budgets to the less "*promising*" branches to strike a balance.

For the example in Fig. 1, we differ from the naive BaB in the selection of branches. As shown in Fig. 2, ABONN selects the most "*promising*" node over the children of the root to proceed, and manages to find a real counterexample in the subsequent steps. Notably, ABONN is more efficient than the naive BaB, because it saves two visits of the sub-problems, each of which involves an expensive process of problem solving.

III. Preliminaries

Neural Networks. A (feed-forward) neural network $N : \mathbb{R}^n \to \mathbb{R}^m$ (see the example in Fig. 1a) maps an n-dim input to an m-dim output. The mapping from the input to output is conducted by alternating between affine transformations $\hat{\boldsymbol{x}}_i = W_i \boldsymbol{x}_{i-1} + B_i$ (where W_i is a matrix of *weights* and B_i is a vector of

biases) and non-linear transformations $\boldsymbol{x}_i = \sigma(\hat{\boldsymbol{x}}_i)$ (where σ is called an *activation function*). Specially, $\boldsymbol{x}_0$ is the input of N and $\boldsymbol{x}_L$ is the output of N, where L is the number of *layers* of the network. Regarding the selection of activation function, we follow many existing works (see a survey [1]), and adopt ReLU (i.e., $\sigma(x) = \max(0, x)$) as the activation function.

Verification Problem. We denote the *specification* of a neural network N as a tuple (Φ, Ψ), where Φ is a predicate over the input of N, and Ψ is a predicate over the output of N. Commonly-used properties, such as *local robustness* in image classification, can be described by such a specification. Given a reference input $\boldsymbol{x}_0$, $\Phi(\boldsymbol{x})$ requires that the input $\boldsymbol{x}$ must stay in the region $\{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x}_0\|_\infty \leq \epsilon\}$, where $\epsilon \in \mathbb{R}$ denotes a perturbation distance, and $\Psi(N(\boldsymbol{x}))$ requires that the output $N(\boldsymbol{x})$ must imply the same label as that of $\boldsymbol{x}_0$.

A verification problem concerns with a question as follows:

- *Given:* a neural network N and a specification (Φ, Ψ);
- *Return:* true if $\Psi(N(\boldsymbol{x}))$ holds, for any input $\boldsymbol{x}$ that holds $\Phi(\boldsymbol{x})$; false otherwise.

A *verifier* is implemented to answer a verification problem. In case it returns false, it will simultaneously return a *counterexample* $\hat{\boldsymbol{x}}$, which is an input that holds $\Phi(\hat{\boldsymbol{x}})$ but does not hold $\Psi(\hat{\boldsymbol{x}})$, as an evidence of the violation of specification.

Branch and Bound (`BaB`). `BaB` [10] is the state-of-the-art neural network verification approach, and has been adopted as the theoretical foundation of several verification tools, such as $\alpha\beta$-Crown [13]. While it is essentially a "divide-and-conquer" strategy, now it is often used to mitigate the *incompleteness* issue of approximated verifiers, by applying these verifiers to smaller verification problems. Below, we elaborate on how `BaB` collaboratively works with approximated verifiers.

Approximated verifiers, denoted as `AppVer`, are a class of verifiers that solve a problem by over-approximating the output region of a neural network—once the over-approximated output satisfies the specification, the original output must also satisfy it. By applying to a problem, `AppVer` can return a real value $\hat{p}$ to indicate the satisfaction of the over-approximated output: if $\hat{p}$ is positive, then the over-approximated output satisfies the specification; otherwise, it violates the specification. However, in the latter case, the result given by `AppVer` is not necessarily correct, because it is possible that the original output does not violate the specification. This can be validated by checking the counterexample $\hat{\boldsymbol{x}}$ returned by `AppVer`, and if $\Psi(N(\hat{\boldsymbol{x}}))$ is not violated, then $\hat{\boldsymbol{x}}$ is a spurious one, and so `AppVer` raises a false alarm. This is known as the *incompleteness issue* of `AppVer`.

`BaB` is used to mitigate this issue, by applying `AppVer` adaptively to smaller problems. It works as follows:

i) First, it applies `AppVer` to the original verification problem, and receives $\hat{p}$: if $\hat{p}$ is positive, or $\hat{p}$ is negative with a valid counterexample $\hat{\boldsymbol{x}}$, verification can be terminated;
ii) In the case that $\hat{p}$ is negative and the counterexample $\hat{\boldsymbol{x}}$ is a spurious one, it splits the problem into two sub-problems. This can be done by adding an additional logical constraint to each of the sub-problems, which predicates over the input condition of the ReLU in a neuron selected from the network (i.e., the additional constraint is either the input of ReLU is positive, or the input of ReLU is negative);
iii) It then applies `AppVer` to each of the sub-problems, and decides whether it needs to further split the sub-problems, in the same way as that in Step ii.

`BaB` Tree. The process of problem splitting and solving in `BaB` can be characterized as a tree, as illustrated in Fig. 1b. In this tree, each node identifies a sub-problem, and each sub-problem is identified by a sequence Γ of input constraints of ReLUs (namely, the ReLU constraints from the root to the current node). Given the i-th ReLU in a network, let r_i^+ denote the condition that the input of the ReLU is positive, and r_i^- denote the condition that the input of the ReLU is negative. Specially, the root is identified by an empty sequence.

It remains a problem in `BaB` how a ReLU should be selected in the network to split a problem, when a false alarm arises with the problem. There have been various strategies devised for that purpose, such as DeepSplit [14] and FSB [15]. All these strategies can be seen as a pre-defined heuristic H that can return a ReLU given a specific (sub-)problem (namely, given a sequence of ReLUs that have already been expanded). In this work, our aim is not to optimize H, but we are orthogonal to that line of work (see §VI); we simply select the state-of-the-art ReLU selection method [14], following existing literature [12].

IV. ABONN: The Proposed Verification Approach

As introduced, `BaB` produces a large space of sub-problems and accomplishes verification by exhaustively visiting the sub-problems. However, the strategy adopted by existing `BaB` for space exploration is naive, in the sense that it ignores the importance of different sub-problems, but visits them in a naive "*first come, first serve*" manner, which is very inefficient.

In this paper, we leverage a notion of "importance" over the sub-problems (detailed in §IV-A) and devise a novel `BaB`-based verification approach that features an adaptive tree exploration in a *Monte-Carlo tree search (MCTS)* style (detailed in §IV-B). Notably, we link the notion of "importance" to the *likelihood* of finding counterexamples in different sub-problems, and the rationale behind is as follows: during verification, we prioritize those sub-problems that are more likely to find counterexamples; as soon as we find a counterexample, we can immediately terminate the verification and return false as a result; even though we cannot find such a counterexample, compared to naive `BaB`, we only differ in the order we visit the sub-problems but we can still manage to verify the problem finally.

A. Counterexample Potentiality

We elaborate on the notion of "importance" of different sub-problems, by defining an order called *counterexample potentiality*. Specifically, the potentiality of having a counterexample in a node of `BaB` tree is related to the following two attributes:

- *Node depth.* In `BaB` tree, the more a problem is split, the less over-approximation will be introduced when applying `AppVer`; therefore, under the premise that $\hat{p}$ is still negative for a node Γ, the greater its depth is, the more likely it is to find a real counterexample from Γ;

- $\hat{p}$. The value $\hat{p}$, obtained by applying `AppVer` to a sub-problem, is a quantity that reflects *how far* the sub-problem is from being violated, according to the evaluation of `AppVer` that relies on the over-approximation of the network output. Although it is not precise, under a fixed `AppVer`, this value is correlated with the real satisfaction level that can be computed by the real output region, and so it can be used to indicate the potentiality of counterexamples. Specifically, in the case where $\hat{p}$ is negative, the greater $|\hat{p}|$ is, the more likely that the sub-problem contains a counterexample.

Based on the two node attributes mentioned above, we define counterexample potentiality of a node in Def. 1.

Definition 1 (Counterexample potentiality): Let Γ be a node that has depth $\texttt{depth}(\Gamma)$ and verifier evaluation $\hat{p}$ (with a counterexample $\hat{\boldsymbol{x}}$ if $\hat{p} < 0$). The counterexample potentiality $[\![\Gamma]\!] \in [0,1] \cup \{+\infty, -\infty\}$ of Γ is computed as follows:

$$[\![\Gamma]\!] := \begin{cases} -\infty & \text{if } \hat{p} > 0 \\ +\infty & \text{if } \hat{p} < 0 \text{ and } \texttt{valid}(\hat{\boldsymbol{x}}) \\ \lambda \frac{\texttt{depth}(\Gamma)}{K} + (1-\lambda)\frac{\hat{p}}{\hat{p}_{\min}} & \text{otherwise} \end{cases}$$

where $\lambda \in [0,1]$ is a parameter that controls the weights of the two attributes, and K is the total number of neurons (i.e., ReLUs) in the network.

Def. 1 characterizes the likelihood of containing counterexamples in a node Γ, in the following way:

- If $\hat{p} > 0$, there is no chance to find a counterexample in Γ, and so $[\![\Gamma]\!] = -\infty$;
- If $\hat{p} < 0$ and $\hat{\boldsymbol{x}}$ is a valid one, it means that a counterexample is already found, so $[\![\Gamma]\!] = +\infty$;
- Otherwise, $[\![\Gamma]\!]$ is determined by the node attributes as mentioned above, and we use a hyperparameter λ to adjust the weights of the two attributes.

In §IV-B, we present our efficient `BaB`-based verification approach, by exploiting this counterexample potentiality to guide the exploration of sub-problem space.

B. MCTS-Style Tree Exploration for `BaB`-Based Verification

We present our MCTS-style `BaB` tree exploration algorithm for verification of neural networks. The idea is that, we use counterexample potentiality to guide the search towards the sub-problems that are more likely to find counterexamples, so compared to the naive `BaB`, our tree exploration is imbalanced and favors the branches that are more "*promising*".

Our verification algorithm is presented in Alg. 1, and see Fig. 2 for an illustration of the different stages of the algorithm.

Initialization. Alg. 1 starts with handling the root node, which identifies the original verification problem. It applies `AppVer` to the problem, and obtains the returned $\hat{p}$ and $\hat{\boldsymbol{x}}$ (in case $\hat{p} < 0$) (Line 1). If $\hat{p} > 0$, or $\hat{p} < 0$ and $\hat{\boldsymbol{x}}$ is valid, the verification can be concluded (Line 8); otherwise, i.e., $\hat{p} < 0$ is a false alarm, so the problem needs to be split. Here, we use $\mathsf{R}(\Gamma)$ to record the *reward* of the node Γ (Line 2), and $\mathcal{T}(\Gamma)$ to record the (sub-)tree (i.e., the set of nodes in the tree) that has Γ as its root (Line 3). Later, we will show that these two structures are helpful to explore the tree by the mechanism of MCTS.

Algorithm 1 ABONN: An MCTS-style verification algorithm

Require: A neural network N, input and output specification Φ and Ψ, an approximated verifier $\texttt{AppVer}(\cdot)$, a ReLU selection heuristic $\mathsf{H}(\cdot)$, and hyperparameters λ and c.
Ensure: A *verdict* $\in \{\texttt{true}, \texttt{false}, \texttt{timeout}\}$

1: $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle \leftarrow \texttt{AppVer}(N, \Phi, \Psi, \varepsilon)$
2: $\mathsf{R}(\varepsilon) \leftarrow [\![\varepsilon]\!]$
3: $\mathcal{T}(\varepsilon) \leftarrow \{\varepsilon\}$
4: **if** $\hat{p} < 0$ and $\texttt{not valid}(\hat{\boldsymbol{x}})$ **then**
5: **while** not reach termination condition **do**
6: MCTS-BAB$(\varepsilon, N, \Phi, \Psi)$
7: **return** $\begin{cases} \texttt{true} & \text{if } \mathsf{R}(\varepsilon) = -\infty \\ \texttt{false} & \text{if } \mathsf{R}(\varepsilon) = +\infty \\ \texttt{timeout} & \text{otherwise} \end{cases}$
8: **else**
9: **return** $\begin{cases} \texttt{true} & \text{if } \hat{p} > 0 \\ \texttt{false} & \text{if } \hat{p} < 0 \text{ and } \texttt{valid}(\hat{\boldsymbol{x}}) \end{cases}$

10: **function** MCTS-BAB(Γ, N, Φ, Ψ)
11: $r_k \leftarrow \mathsf{H}(\Gamma)$ ▷ select a ReLU
12: **if** $\Gamma \cdot r_k^+ \in \mathcal{T}$ **then**
13: $\Gamma^* \leftarrow \arg\max_{a \in \{r_k^+, r_k^-\}} \left(\mathsf{R}(\Gamma \cdot a) + c\sqrt{\frac{2 \ln |\mathcal{T}(\Gamma)|}{|\mathcal{T}(\Gamma \cdot a)|}} \right)$ ▷ Select a child by UCB1
14: MCTS-BAB$(\Gamma^*, N, \Phi, \Psi)$ ▷ recursive call
15: **else**
16: **for** $a \in \{r_k^+, r_k^-\}$ **do**
17: $\langle \hat{p}, \hat{\boldsymbol{x}} \rangle \leftarrow \texttt{AppVer}(N, \Phi, \Psi, \Gamma \cdot a)$
18: $\mathsf{R}(\Gamma \cdot a) \leftarrow [\![\Gamma \cdot a]\!]$
19: $\mathcal{T}(\Gamma \cdot a) \leftarrow \{\Gamma \cdot a\}$
20: $\mathsf{R}(\Gamma) \leftarrow \arg\max_{a \in \{r_k^+, r_k^-\}} [\![\Gamma \cdot a]\!]$ ▷ back-propagate rewards
21: $\mathcal{T}(\Gamma) \leftarrow \mathcal{T}(\Gamma) \cup \mathcal{T}(\Gamma \cdot r_k^+) \cup \mathcal{T}(\Gamma \cdot r_k^-)$ ▷ record new nodes

Expansion. The function MCTS-BAB goes through an MCTS workflow. Given a node Γ, it first checks whether the children of Γ have been expanded. If not yet, it expands the children of Γ, by splitting the problem identified by Γ into sub-problems. The two sub-problems are respectively identified by $\Gamma \cdot r_k^+$ and $\Gamma \cdot r_k^-$, where r_k is the ReLU selected by H in Line 11. The algorithm applies `AppVer` to each of the sub-problems (Line 17). Based on the results, it computes the counterexample potentiality and records it as *reward* (Line 18), and uses $\mathcal{T}$ to record the newly expanded children (Line 19).

Back-Propagation. After expansion, it propagates the rewards and visits backwards to the ancestors of the new nodes, until the root node. In terms of rewards, the reward of the parent node will be updated to be the maximal reward of its children (Line 20); moreover, the newly expanded nodes will be added to $\mathcal{T}$ of the parent node (Line 21).

Selection. If all children of a node Γ have been expanded, it needs to select a child to proceed. In line with normal MCTS, it selects by UCB1 [11], which is an algorithm that favors not only the branches that have greater rewards, but also considers the branches that are less visited, because rewards may not always be accurate to reflect the likelihood of counterexample existence in different sub-problems. UCB1 algorithm is given in Line 13 of Alg. 1: it selects a child of Γ by comparing an aggregated value of their rewards and visits. In particular, the first term is the reward of each child, and the second term is inversely proportional to the number of visits of each child; a

TABLE I: Details of the benchmarks

Model Dataset	Architecture	Dataset	#Neurons	# Instances
$\texttt{MNIST}_{\texttt{L2}}$	2 × 256 linear	MNIST	512	112
$\texttt{MNIST}_{\texttt{L4}}$	4 × 256 linear	MNIST	1024	104
CIFAR-10BASE	2 Conv, 2 linear	CIFAR-10	4852	115
CIFAR-10WIDE	2 Conv, 2 linear	CIFAR-10	6244	101
CIFAR-10DEEP	4 Conv, 2 linear	CIFAR-10	6756	120

hyperparameter c is used to strike a balance of the two terms.

Termination. The algorithm is terminated until the termination condition is reached (Line 5). In particular, there are three conditions, and it suffices to meet one of them to terminate:

- $\mathsf{R}(\epsilon) = +\infty$: it implies that a real counterexample has been found, and so verification can be terminated with false;
- $\mathsf{R}(\epsilon) = -\infty$: it implies that all of sub-problems have been verified, and so verification can be terminated with true;
- *timeout*: If the time budget is used up, verification should also be terminated without meaningful conclusion.

V. Experimental Evaluation

A. Experiment Settings

Baseline and Metrics. We compare with two state-of-the-art verification approaches, namely, BaB-baseline and $\alpha\beta$-Crown. The evaluation metrics include the number of instances solved and the average time cost of each approach. We also compute the speedup of ABONN for individual verification problems w.r.t. the BaB-baseline. Our baselines are as follows:

- BaB-baseline: The naive BaB as introduced in §III, i.e., it explores sub-problem space in a "breadth-first" manner;
- $\alpha\beta$-Crown [8], [13]: The state-of-the-art verification tool according to [3], that features various sophisticated heuristics for performance improvement.

Our implementation of ABONN adopts the same approximated verifiers [7], [16] and ReLU selection heuristic [14] as that of BaB-baseline. For the hyperparameters in Alg. 1, as a default version of our tool, we set λ as 0.5 and c as 0.2; we will study the influences of these two hyperparameters in RQ2. All the code and data are available online[1].

Benchmarks. We adopt 552 verification problems about L_∞-based local robustness for MNIST and CIFAR-10. These datasets are standard benchmarks in community, widely recognized in VNN-COMP [3], an annual competition of neural network verification. We select meaningful problems that are neither too easy nor too hard to solve, as evidenced by the distribution of the tree sizes with BaB-baseline in Fig. 3. We evaluate two fully connected networks with MNIST and three convolutional networks with CIFAR-10, as shown in Table I. Moreover, Table I also presents the number of specifications considered for each model.

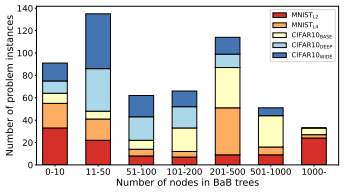


Fig. 3: The distribution of the sizes of the BaB trees used in our experiments

[1]https://github.com/DeepLearningVerification/ABONN

TABLE II: RQ1 – Overall comparison of time consumption (in seconds) and the total number of solved instances.

Model Dataset	BaB-baseline		$\alpha\beta$-Crown		ABONN	
	Solved	Time	Solved	Time	Solved	Time
$\texttt{MNIST}_{\texttt{L2}}$	95	245.11	96	19.53	92	248.29
$\texttt{MNIST}_{\texttt{L4}}$	59	200.68	43	360.97	57	270.48
$\texttt{CIFAR-10}_{\texttt{BASE}}$	27	782.31	32	699.77	106	176.87
$\texttt{CIFAR-10}_{\texttt{DEEP}}$	23	749.74	40	516.25	67	369.58
$\texttt{CIFAR-10}_{\texttt{WIDE}}$	26	706.04	38	520.3	75	246.03

Experiment Environment. Experiments ran on an AWS EC2 instance (8-core Xeon E5 2.90GHz, 16GB RAM) with 1000s timeout per problem. GUROBI 9.1.2 was used as the solver.

B. Evaluation Results

RQ1: Efficiency of ABONN compared to baselines.

Table II shows the number of solved problems across MNIST and CIFAR-10 and the average time costs of each approach. ABONN demonstrates significant advantages in the CIFAR-10 models that are relatively complex. For instance, in $\texttt{CIFAR-10}_{\texttt{BASE}}$, it solves 79 more instances than BaB-baseline, and 74 more than $\alpha\beta$-Crown within the time budget. In MNIST models that are less complex, ABONN exhibits comparable performance with the baseline approaches. These results show the efficiency of ABONN in handling complex models.

In the scatter plots of Fig. 4, x-axis denotes the time cost for individual problems of ABONN, and y-axis denotes the speedup (i.e., $\frac{T_{\text{BaB-baseline}}}{T_{\text{ABONN}}}$) of ABONN over BaB-baseline. It reveals significant performance improvement across different models, we can observe many instances for which ABONN outperforms BaB-baseline. In MNIST, the speedups are around 1-5 times, and in CIFAR-10, the speedups are around 20-80 times. In many cases, while BaB-baseline struggles, ABONN manages to solve the problems very efficiently.

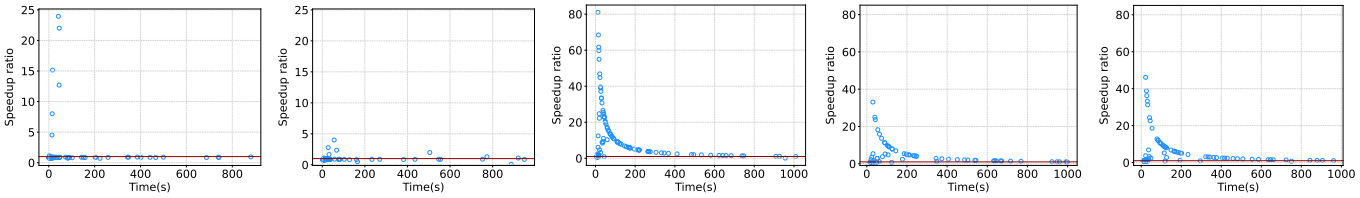
RQ2: Impacts of hyperparameter selection on ABONN.

Figures 5a, 5b, and 5c illustrate the impact of hyperparameters (λ in counterexample potentiality, Def. 1 and c in UCB1, Line 13 of Alg. 1) to the performance of ABONN.

Regarding the selection of λ, all the plots show that $\lambda = 0.5$ is the best choice. The value $\lambda = 0.5$ aims at a balance between the two node attributes in counterexample potentiality, and the results show that the likelihood of counterexample existence is closely correlated to both of the two attributes.

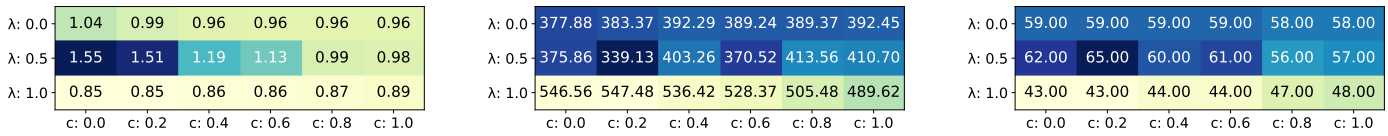
The hyperparameter c is crucial in MCTS [17], which decides the extent to which it favors "exploitation" or "exploration". While in Fig. 5a $c = 0.2$ (i.e., balance between "exploitation" and "exploration" to some extent) is a bit weaker than $c = 0$ (i.e., pure exploitation), in Fig. 5b and Fig. 5c, $c = 0.2$ is the best performer. This indicates that, 1) our counterexample potentiality is effective as a search guidance, so even pure exploitation performs well; 2) pure exploitation may be superior in individual problems, but not as good as the balanced strategy in average performance. So, it is meaningful to strike a balance between "exploitation" and "exploration" in our MCTS.

RQ3: Comparison between BaB-baseline and ABONN for violated and certified verification problems.



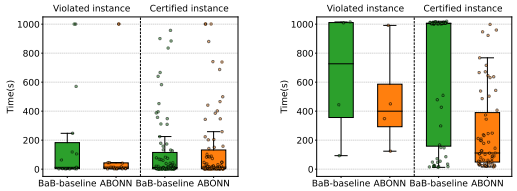
(a) $\texttt{MNIST}_{\texttt{L2}}$ (b) $\texttt{MNIST}_{\texttt{L4}}$ (c) $\texttt{CIFAR-10}_{\texttt{BASE}}$ (d) $\texttt{CIFAR-10}_{\texttt{DEEP}}$ (e) $\texttt{CIFAR-10}_{\texttt{WIDE}}$

Fig. 4: RQ1 – Comparison of ABONN over BaB-baseline in time costs and speedup. Each blue dot stands for a problem.



(a) Avg. speedup (w.r.t. BaB-baseline) (b) Avg. time (secs) (c) Number of solved problems

Fig. 5: RQ2 – Impact of hyperparameter selection across different λ and c. A darker cell implies a better performance.



(a) $\texttt{MNIST}_{\texttt{L2}}$ (b) $\texttt{CIFAR-10}_{\texttt{DEEP}}$

Fig. 6: RQ3 – Comparison between BaB-baseline and ABONN for violated and certified verification problems

The box plots in Fig. 6 shows the breakdown of the verification time, respectively for violated problems and for certified problems, of $\texttt{MNIST}_{\texttt{L2}}$ and $\texttt{CIFAR-10}_{\texttt{DEEP}}$.

In Fig. 6a, ABONN exhibits a smaller interquartile range than BaB-baseline for violated instances. For certified instances, the two approaches are almost the same. This certifies that ABONN is indeed superior in finding counterexamples compared to BaB-baseline, and so it outperforms BaB-baseline when dealing with violated verification problems. For certified instances, it performs similarly to BaB-baseline, which is also expected.

In Fig. 6b, similarly, ABONN exhibits its superiority in handling violated verification problems, evidently outperforming BaB-baseline. While BaB-baseline used up the time budget ($1000s$) for many problems, ABONN solves them much more efficiently. Surprisingly, ABONN also demonstrates outperformance in certified instances, which means that it manages to verify the problems by visiting less sub-problems. This should result from a mutual influence between the ReLU selection heuristic [14] and our adaptive tree exploration strategy, namely, our strategy manages to lead the heuristic to a better ReLU selection. In future work, we will delve more into the impact of our strategy to ReLU selection.

VI. Related Work

Neural Network Verification. Neural network verification has been extensively studied in the past decade [5], [7], [18]–[23] and approximated methods are often preferable thanks to their efficiency [24]–[29]. In particular, there is a line of work [30]–[32] that exploits information from (spurious) counterexamples to refine the abstraction (i.e., CEGAR [33]). In contrast, we do not use counterexamples to refine the approximation of verifiers, but we explore the sub-problem space of BaB guided by the possibility of finding counterexamples.

Branching Strategies in BaB. Many studies [10], [14], [15], [34]–[36] aim to optimize the branching strategy (i.e., H in Alg. 1) to pursue better abstraction refinement. In contrast, our approach is orthogonal to that line of works, and we can adopt any of those strategies in our algorithm. Essentially, we change the way of tree growth in BaB by favoring those branches that are more "promising", rather than the way of problem splitting.

Testing and Attacks. Testing and attacks aim to efficiently generate counterexamples to fool the network, and they have been extensively studied [2], [37]–[42]. Although these approaches are efficient, they cannot provide rigorous guarantee on the quality of the networks, even if they fail to detect counterexamples. In comparison, in line with BaB, our approach is a verification approach that can provide rigorous proofs about specification satisfaction, after verifying all the sub-problems.

VII. Conclusion and Future Work

We propose a neural network verification approach that features adaptive exploration of the sub-problems produced by BaB. ABONN is guided by *counterexample potentiality*, so we can efficiently find the sub-problems that contain counterexamples. Experimental evaluation demonstrates the superiority of our proposed approach in efficiency over existing baselines.

As future work, we aim to investigate how our approach can be used to improve ReLU selection heuristics, such as DeepSplit [14], such that we can further accelerate verification.

Acknowledgements

This research is supported by JSPS KAKENHI Grant No. JP23H03372, No. JP23K16865, JST-Mirai Grant No. JPMJMI20B8, and Australian Research Grant FT220100391.

REFERENCES

[1] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, M. J. Kochenderfer *et al.*, "Algorithms for verifying deep neural networks," *Foundations and Trends® in Optimization*, vol. 4, no. 3-4, pp. 244–404, 2021.

[2] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd Int. Conf. on Learning Representations (ICLR'15)*. San Diego, CA, United States: Int. Conf. on Learning Representations, ICLR, 2015.

[3] M. N. Müller, C. Brix, S. Bak, C. Liu, and T. T. Johnson, "3rd international verification of neural networks competition (VNN-COMP 2022): Summary and results," *arXiv preprint arXiv:2212.10376*, 2022.

[4] C.-H. Cheng, G. Nührenberg, and H. Ruess, "Maximum resilience of artificial neural networks," in *Automated Technology for Verification and Analysis*, D. D'Souza and K. Narayan Kumar, Eds. Springer Int. Publishing, 2017, pp. 251–268.

[5] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *Int. Conf. on Learning Representations*, 2018.

[6] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," *Advances in neural information processing systems*, vol. 31, 2018.

[7] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[8] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient neural network robustness certification with general activation functions," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[9] E. Wong and Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," in *Int. Conf. on Machine Learning*. PMLR, 2018, pp. 5286–5295.

[10] R. Bunel, P. Mudigonda, I. Turkaslan, P. Torr, J. Lu, and P. Kohli, "Branch and bound for piecewise linear neural network verification," *Journal of Machine Learning Research*, vol. 21, no. 2020, 2020.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[12] S. Ugare, D. Banerjee, S. Misailovic, and G. Singh, "Incremental verification of neural networks," *ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1920–1945, 2023.

[13] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification," *Advances in Neural Information Processing Systems*, vol. 34, pp. 29 909–29 921, 2021.

[14] P. Henriksen and A. Lomuscio, "Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis." in *IJCAI*, 2021, pp. 2549–2555.

[15] A. De Palma, R. Bunel, A. Desmaison, K. Dvijotham, P. Kohli, P. H. Torr, and M. P. Kumar, "Improved branch and bound for neural network verification via lagrangian decomposition," *arXiv preprint arXiv:2104.06718*, 2021.

[16] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[17] Z. Zhang, G. Ernst, S. Sedwards, P. Arcaini, and I. Hasuo, "Two-layered falsification of hybrid systems guided by monte carlo tree search," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2894–2905, 2018.

[18] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Springer Int. Publishing, 2017, pp. 97–117.

[19] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *Automated Technology for Verification and Analysis: 15th Int. Symp., ATVA 2017, Proceedings 15*. Springer, Oct. 2017, pp. 269–286.

[20] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *Computer Aided Verification: 29th Int. Conf., CAV 2017, Part I 30*. Springer, July 2017, pp. 3–29.

[21] C. Müller, G. Singh, M. Püschel, and M. T. Vechev, "Neural network robustness verification on gpus," *CoRR, abs/2007.10868*, 2020.

[22] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[23] Z. Shi, Y. Wang, H. Zhang, J. Z. Kolter, and C.-J. Hsieh, "Efficiently computing local lipschitz constants of neural networks via bound propagation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 2350–2364, 2022.

[24] R. Anderson, J. Huchette, C. Tjandraatmadja, and J. Vielma, "Strong convex relaxations and mixed-integer programming formulations for trained neural networks (2018)," 1811.

[25] C. Tjandraatmadja, R. Anderson, J. Huchette, W. Ma, K. K. Patel, and J. P. Vielma, "The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification," *Advances in Neural Information Processing Systems*, vol. 33, pp. 21 675–21 686, 2020.

[26] G. Singh, R. Ganvir, M. Püschel, and M. Vechev, "Beyond the single neuron convex barrier for neural network certification," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[27] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev, "Prima: general and precise neural network certification via scalable convex hull approximations," *ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–33, 2022.

[28] ——, "Precise multi-neuron abstractions for neural network certification," *arXiv preprint arXiv:2103.03638*, 2021.

[29] A. Raghunathan, J. Steinhardt, and P. S. Liang, "Semidefinite relaxations for certifying robustness to adversarial examples," *Advances in neural information processing systems*, vol. 31, 2018.

[30] P. Yang, R. Li, J. Li, C.-C. Huang, J. Wang, J. Sun, B. Xue, and L. Zhang, "Improving neural network verification through spurious region guided refinement," in *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 389–408.

[31] M. Ostrovsky, C. Barrett, and G. Katz, "An abstraction-refinement approach to verifying convolutional neural networks," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2022, pp. 391–396.

[32] Z. Zhao, Y. Zhang, G. Chen, F. Song, T. Chen, and J. Liu, "Cleverest: accelerating cegar-based neural network verification via adversarial attacks," in *International Static Analysis Symposium*. Springer, 2022, pp. 449–473.

[33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 2000, pp. 154–169.

[34] Z. Shi, Q. Jin, Z. Kolter, S. Jana, C.-J. Hsieh, and H. Zhang, "Neural network verification with branch-and-bound for general nonlinearities," *arXiv preprint arXiv:2405.21063*, 2024.

[35] J. Lu and M. P. Kumar, "Neural network branching for neural network verification," *arXiv preprint arXiv:1912.01329*, 2019.

[36] C. Ferrari, M. N. Muller, N. Jovanovic, and M. Vechev, "Complete verification via multi-neuron relaxation guided branch-and-bound," *arXiv preprint arXiv:2205.00263*, 2022.

[37] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *26th Symp. on Operating Systems Principles*, 2017, pp. 1–18.

[38] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4901–4911.

[39] M. Andriushchenko, F. Croce, N. Flammarion, and M. Hein, "Square attack: a query-efficient black-box adversarial attack via random search," in *European conference on computer vision*. Springer, 2020, pp. 484–501.

[40] C. Xie, Z. Zhang, Y. Zhou, S. Bai, J. Wang, Z. Ren, and A. L. Yuille, "Improving transferability of adversarial examples with input diversity," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2730–2739.

[41] H. Zhang, S. Wang, K. Xu, Y. Wang, S. Jana, C.-J. Hsieh, and Z. Kolter, "A branch and bound framework for stronger adversarial attacks of relu networks," in *International Conference on Machine Learning*. PMLR, 2022, pp. 26 591–26 604.

[42] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. Ieee, 2017, pp. 39–57.