Efficient Incremental Verification of Neural Networks Guided by Counterexample Potentiality

GUANQIN ZHANG, UNSW Sydney, Australia and CSIRO's Data61, Australia
ZHENYA ZHANG, Kyushu University, Japan
H.M.N. DILUM BANDARA, CSIRO's Data61, Australia and UNSW Sydney, Australia
SHIPING CHEN, CSIRO's Data61, Australia and UNSW Sydney, Australia
JIANJUN ZHAO, Kyushu University, Japan
YULEI SUI, UNSW Sydney, Australia

Incremental verification is an emerging neural network verification approach that aims to accelerate the verification of a neural network N^* by reusing the existing verification result (called a *template*) of a similar neural network N. To date, the state-of-the-art incremental verification approach leverages the problem splitting history produced by *branch and bound (BaB)* in verification of N, to select only a part of the sub-problems for verification of N^*, thus more efficient than verifying N^* from scratch. While this approach identifies *whether* each sub-problem should be re-assessed, it neglects the information of *how necessary* each sub-problem should be re-assessed, in the sense that the sub-problems that are more likely to contain counterexamples should be prioritized, in order to terminate the verification process as soon as a counterexample is detected.

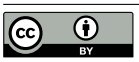
To bridge this gap, we first define a *counterexample potentiality order* over different sub-problems based on the template, and then we propose Olive, an incremental verification approach that explores the sub-problems of verifying N^* orderly guided by counterexample potentiality. Specifically, Olive has two variants, including Olive$^{\text{g}}$, a greedy strategy that always prefers to exploit the sub-problems that are more likely to contain counterexamples, and Olive$^{\text{b}}$, a balanced strategy that also explores the sub-problems that are less likely, in case the template is not sufficiently precise. We experimentally evaluate the efficiency of Olive on 1445 verification problem instances derived from 15 neural networks spanning over two datasets `MNIST` and `CIFAR-10`. Our evaluation demonstrates significant performance advantages of Olive over state-of-the-art classic verification and incremental approaches. In particular, Olive shows evident superiority on the problem instances that contain counterexamples, and performs as well as Ivan on the certified problem instances.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Software testing and debugging**.

Additional Key Words and Phrases: neural network verification, incremental verification, branch and bound, counterexample potentiality

ACM Reference Format:
Guanqin Zhang, Zhenya Zhang, H.M.N. Dilum Bandara, Shiping Chen, Jianjun Zhao, and Yulei Sui. 2025. Efficient Incremental Verification of Neural Networks Guided by Counterexample Potentiality. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 83 (April 2025), 28 pages. https://doi.org/10.1145/3720417

Authors' Contact Information: Guanqin Zhang, UNSW Sydney, Kensington, Australia and CSIRO's Data61, Sydney, Australia, guanqin.zhang@unsw.edu.au; Zhenya Zhang, Kyushu University, Fukuoka, Japan, zhang@ait.kyushu-u.ac.jp; H.M.N. Dilum Bandara, CSIRO's Data61, Sydney, Australia and UNSW Sydney, Kensington, Australia, dilum.bandara@data61.csiro.au; Shiping Chen, CSIRO's Data61, Sydney, Australia and UNSW Sydney, Kensington, Australia, shiping.chen@data61.csiro.au; Jianjun Zhao, Kyushu University, Fukuoka, Japan, zhao@ait.kyushu-u.ac.jp; Yulei Sui, UNSW Sydney, Kensington, Australia, y.sui@unsw.edu.au.



ACM 2475-1421/2025/4-ART83
https://doi.org/10.1145/3720417

1 Introduction

The prevalence of *deep neural networks (DNNs)* in the era of artificial intelligence has brought revolutionary impacts to numerous domains, such as transportation, healthcare, robotics, and energy. Notably, their exceptional capacity to learn from extensive datasets empowers them to deliver outstanding performance across diverse applications, e.g. image recognition, semantic segmentation, natural language processing and strategic decision-making.

Despite the prevalence of DNNs, there are growing concerns about their safety and robustness, especially when they are deployed in safety-critical domains, e.g., *automated driving*. Indeed, existing research [Goodfellow et al. 2015; Madry et al. 2018] has shown that even small perturbations to DNN inputs can result in a significant alteration of inference results. In real-world deployments of DNNs, such perturbations are ubiquitous, ranging from naturally occurring environmental noise to deliberate adversarial attacks by hackers.

Existing approaches: Numerous efforts have been paid to enhance the safety and robustness assurance of DNNs, among which verification is a preferable technique that can provide rigorous formal guarantees. Technically, verification approaches need to combat with the non-linearity of DNN inferences brought by their activation functions, such as the piecewise-linear *ReLU* function.

While MILP [Cheng et al. 2017; Tjeng et al. 2018] or SMT solving [Katz et al. 2017] can handle this issue, these approaches suffer from a severe problem of scalability. Alternatively, approximated methods [Gehr et al. 2018; Singh et al. 2018, 2019; Wang et al. 2018b; Wong and Kolter 2018; Xue and Sun 2023; Zhang et al. 2022] use linear bounds to over-approximate the output ranges of activation functions; these approaches are thus more efficient, but not complete, i.e., they may produce false alarms with spurious counterexamples. To mitigate this issue, Bunel et al. [Bunel et al. 2020] combine approximated methods with the *branch and bound* (BaB) strategy, which first applies approximated methods to a problem, and then splits the problem into sub-ones if it confirms a violation to be a false alarm. Since approximated verifiers can achieve better precision with sub-problems, BaB can finally draw a reliable conclusion after splitting the problem for sufficiently many times. That said, the performance of BaB can be erratic – the rapid growth in the number of sub-problems can significantly diminish the efficiency of BaB.

Recently, *incremental verification* [Fischer et al. 2022; Ugare et al. 2023, 2022] emerges with the aim of accelerating the verification of a neural network N^*, by leveraging the existing verification results of a similar neural network N as a template, where N^* has an identical structure to N but slightly differs in model parameters. In practice, there could be various strategies for template defining and reusing. The state-of-the-art approach Ivan [Ugare et al. 2023] defines the template as the tree generated by BaB for tracking the problem-splitting history of N, and then it exploits the tree to skip the sub-problems identified as unnecessary to be re-assessed. As this strategy can reduce the number of sub-problems that need to be checked in the verification of N^*, it thus performs more efficiently than verifying N^* from scratch; however, its performance improvement w.r.t. BaB could be restricted by the number of sub-problems it skipped, which is very limited.

Motivation: The template, namely, the tree generated by BaB for verification of N, embeds not only *whether* a tree node should be re-assessed (as leveraged by Ivan [Ugare et al. 2023]), but also *how necessary* a tree node should be re-assessed, in the sense that different branches may have different likelihood of containing counterexamples, and the branches that are more likely to find a counterexample should be prioritized. In neural network verification, prioritizing the sub-problems that are more likely to contain counterexamples is meaningful, because verification can be terminated as soon as a counterexample is found. This strategy has been exemplified in [Guo et al. 2021], by which verification is accelerated significantly.

In a template tree, such counterexample likelihood can be signified by two node attributes: i) *the depth* of a tree node reflects the level of problem refinement, so a deeper node can introduce less approximation imprecision from the approximated verifiers, thereby reducing the possibility of producing spurious counterexamples; ii) for each node, the approximated verifier can provide a *quantitative assessment*, which can be used as a quantitative indicator for counterexample likelihood.

Contributions: In this paper, we propose to reuse the information regarding *how necessary* a tree node should be re-assessed according to the template tree of N, for accelerating the verification of N^*. We propose to do so by exploring the template tree *orderly* during the verification of N^* and prioritizing those branches that are more likely to contain counterexamples. To that end, we define a partial order called *counterexample potentiality* over tree nodes that embodies their likelihood of containing counterexamples based on the two attributes of tree nodes, i.e., their depths and their quantitative assessments by approximated verifiers.

Based on the counterexample potentiality order, we propose Olive, an **o**rder-**l**eading **i**ncremental **ve**rification approach, that explores the template tree guided by the counterexample potentiality order to terminate the verification upon encountering a counterexample. Specifically, we propose two versions of Olive, namely, Olive$^{\text{g}}$, a greedy strategy that always favors the exploitation of the tree nodes in which a counterexample is more likely to be found, and Olive$^{\text{b}}$, a balanced strategy that also explores the tree nodes that seem less likely to contain counterexamples, in case there is a considerable gap between N^* and N and thus the template is not precise enough. Our approach is helpful in the following sense: given a neural network whose specification satisfaction is unknown beforehand, it is always desired to know the answer efficiently. Our approach can terminate immediately once a counterexample is found, by which verification result is determined. By our strategy, if the network violates the specification, we can quickly reach the answer; even though it satisfies, we are still not slower than non-ordered approaches like Ivan.

We experimentally evaluate the effectiveness of our proposed approach based on 5 reference models (i.e., the original network N), 15 models as N^*, generated by model different alteration methods, including weight pruning and quantization; along with the specifications regarding local robustness properties, they lead to totally 1445 verification problem instances. Our evaluation demonstrates significant performance advantages of Olive over the baseline approaches BaB and Ivan. In particular, Olive shows evident superiority on the problem instances that contain counterexamples, and performs as well as Ivan on the certified problem instances.

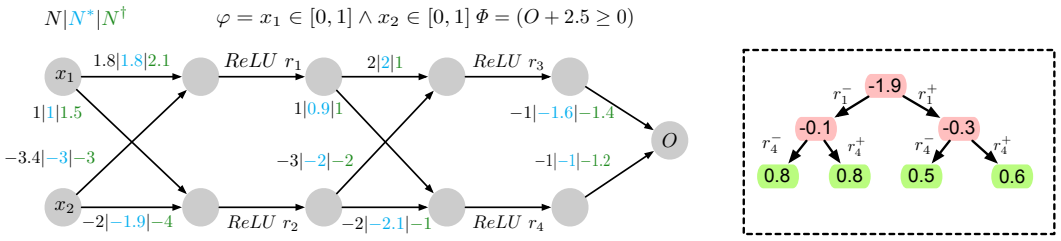
Paper structure: The rest of the paper is organized as follows: §2 overviews our approach by using an illustrative example; §3 introduces the necessary technical background; §4 presents the proposed incremental verification approach; §5 presents our experimental evaluation results; §6 discusses related work. Conclusion and future work are presented in §7.

2 Overview of Our Approach

2.1 Motivating Example

Fig. 1a depicts a group of neural networks N, N^* and $N^\dagger$, where N^* and $N^\dagger$ are generated by slightly altering the model parameters of N, respectively. The verification problem of a neural network concerns whether its output O satisfies that $O + 2.5 \geq 0$, for any input $(x_1, x_2) \in [0, 1] \times [0, 1]$.

Branch and Bound (BaB): Fig. 1b illustrates how BaB solves the problem for N. It relies on an approximated verifier, such as DeepPoly [Singh et al. 2019], that, given a verification problem possibly with extra constraints on the inputs of certain ReLUs, computes an over-approximated assessment $\hat{p} \in \mathbb{R}$ signifying *how far* the specification is from being satisfied. As $\hat{p}$ is over-approximated, it can be explained as follows: if $\hat{p}$ is positive, it is sure that the specification is satisfied; however, if $\hat{p}$ is negative, the satisfaction of specification cannot be decided. BaB follows the steps as follows:



(a) Three neural networks N, N^* and $N^\dagger$ and specification (b) BaB for N

Fig. 1. An example of neural network verification, and BaB approach [Bunel et al. 2020] for network N

1) Initially, BaB applies the approximated verifier to the original verification problem: if $\hat{p} \geq 0$, the problem is verified and the verification can be terminated; otherwise (i.e., $\hat{p} < 0$), the satisfaction of specification cannot be decided yet, which is the case in Fig. 1b where $\hat{p} = -1.9$;
2) In the case when $\hat{p} < 0$, the approximated verifier will report a counterexample, but it could possibly be spurious. BaB then validates the counterexample, by feeding it into the network and checking whether the output of the network indeed violates the specification: if it is a real one, then the verification problem is falsified and thereby terminated; otherwise, BaB splits the problem into two sub-problems, by selecting a ReLU r_i and adding a constraint (either r_i^+: input of r_i being positive, or r_i^-: input of r_i being negative) to each of the sub-problems. In Fig. 1b, the problem is split into two sub-problems by adding the constraints r_1^- and r_1^+;
3) For each of the sub-problems, BaB treats it as the target verification problem and iteratively calls Step 1 and Step 2. Now each sub-problem is a refined one since the ReLU r_i is not needed to be approximated. BaB continues this until all the sub-problems are verified, or a real counterexample is detected with a sub-problem. In Fig. 1b, although the problem is not yet solved by splitting r_1 only, it is solved after further splitting r_4, by which all the four sub-problems are verified.

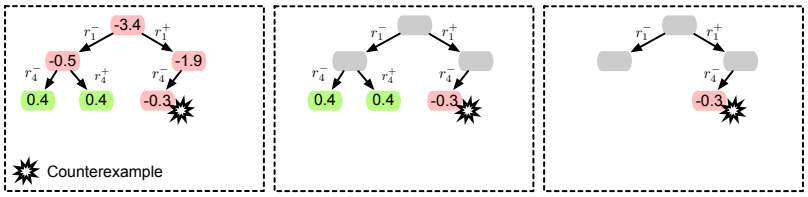
Consequently, BaB produces a tree as illustrated in Fig. 1b, where each node is a (sub-)problem and the $\hat{p}$ for each (sub-)problem is labeled in each node.

Incremental verification: Ivan [Ugare et al. 2023] is an incremental approach that leverages the tree in Fig. 1b as a template to accelerate the verification of a network N^* that has the identical structure to N but slightly differs in model parameters. Ivan is illustrated in Fig. 2b. Compared to re-applying BaB to N^* in Fig. 2a, Ivan skips the non-leaf nodes in the template (i.e., the tree in Fig. 1b) and directly starts with verification of the sub-problems identified by the leaf nodes in Fig. 1b, because during the verification of N^*, BaB may also be not able to terminate until it reaches the sub-problems identified by the leaf nodes in Fig. 1b. By this, Ivan can save the time spent for the non-lead nodes in verification of N^*. Indeed, its superiority is illustrated in Fig. 2b: compared to BaB that needs 6 calls of the approximated verifier (Fig. 2a), Ivan requires only 3 times.

2.2 Olive: **The Proposed Approach**

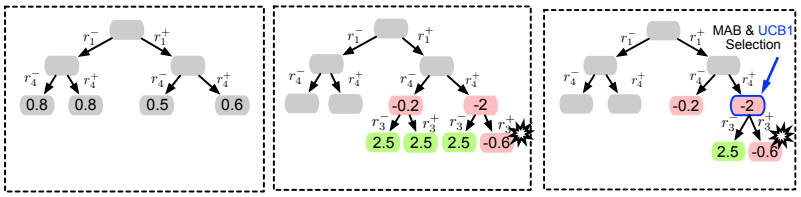
The template tree, e.g., Fig. 1b, embeds not only *whether* (as leveraged by Ivan), but also *how necessary* each node should be re-assessed; different nodes can differ in the necessity of being re-assessed, in the sense that some nodes that are more likely to contain counterexamples should be prioritized, such that verification can be terminated as soon as a counterexample is found.

Our proposed Olive^g infers the potentiality of having counterexample in each node, by leveraging the depth and the quantitative assessment $\hat{p}$ of each node in the template tree shown in Fig. 1b. Consequently, as shown in Fig. 2c, it identifies the leaf node $r_1^+ r_4^-$ (the $\hat{p}$ of which is 0.5) in Fig. 1b that has the same depth but smaller assessment than other leaf nodes, signifying that it is closest



(a) BaB for N^* (b) Ivan for N^* (c) Olive^g for N^*

Fig. 2. Comparison of Olive^g with BaB and Ivan for verification of N^*



(a) Template of N (b) Olive^g for $N^\dagger$ (c) Olive^b for $N^\dagger$

Fig. 3. Comparison between Olive^g and Olive^b for verification of $N^\dagger$

to specification violation. In Fig. 2c, Olive^g prioritizes this node, and it immediately hits the real counterexample, which costs three sub-problem attempts for Ivan (Fig. 2b).

The strategy of Olive^g is greedy, in the sense that it always prefers the nodes that are more likely to contain counterexamples according to the template. However, this strategy can fail, because it is possible that the tree of the updated network is divergent from that of the original network. This is normal, because the small alteration of the model parameters does not necessarily produce a new network that has similar behavior as the original one. In this case, we need a balanced strategy that not only favors the suspicious nodes, but also explores other nodes to avoid being misled by an imprecise template.

To address this challenge, we propose Olive^b, which strikes a balance between exploitation of suspicious nodes and exploration of the nodes that seem less suspicious. As shown in Fig. 3, for the verification of $N^\dagger$ (see Fig. 1), while Olive^g struggles in the branch of $r_1^+ r_4^-$ inspired by the template, Olive^b benefits from the exploration to the sibling of $r_1^+ r_4^-$, such that it manages to detect the branch of $r_1^+ r_4^+$ with $\hat{p} = -2$, which is smaller than that of $r_1^+ r_4^-$ and thus more promising.

Then, by expanding $r_1^+ r_4^+$, Olive^b hits a real counterexample in the node $r_1^+ r_4^+ r_3^+$, and thereby terminates the verification. In comparison with Olive^g that requires 6 calls of approximated verifiers, Olive^b only needs 4 calls, and so it can save much time to achieve the verification result.

3 Preliminaries

3.1 Neural Network Verification

In this paper, we mainly target (fully-connected) *feed-forward neural networks*, a type of neural network that has been widely adopted in literature [Müller et al. 2022a].

Definition 1 (Feed-forward neural network)**.** A *(feed-forward) neural network* is a function $N : \mathbb{R}^{n_0} \to \mathbb{R}^Y$. It consists of l hidden layers and an output layer, and the i-th hidden layer maps $\mathbf{x}_{i-1} \in \mathbb{R}^{n_{i-1}}$ to $\mathbf{x}_i \in \mathbb{R}^{n_i}$, by $\mathbf{x}_i = \sigma(W_i \mathbf{x}_{i-1} + B_i)$, where $W_i \mathbf{x}_{i-1} + B_i$ is an *affine transformation* parameterized by a matrix of weight W_i and a vector of bias B_i, and σ is a non-linear *activation*

function. Common choices of activation functions include *ReLU*, *sigmoid*, and *tanh*. In this paper, we select the widely-adopted option $ReLU(x) = \max(0, x)$. The output layer maps the output $\mathbf{x}_l$ of the l-th hidden layer to the output $N(\mathbf{x}_0) \in \mathbb{R}^Y$ of the network, by an affine transformation only.

A specification of a neural network is divided to an input specification φ and an output specification Φ, which are formally defined as follows.

Definition 2 (Input and output specification)**.** Let $N\colon \mathbb{R}^{n_0} \to \mathbb{R}^Y$ be a neural network. An input specification $\varphi \subset \mathbb{R}^{n_0}$ is a connected region in the input domain $\mathbb{R}^{n_0}$ of N. An output specification $\Phi\colon \mathbb{R}^Y \to \{\text{true}, \text{false}\}$ is a predicate over the output domain of N. In particular, we adopt the output specifications in the form $(f(\mathbf{x}_l) > 0)$, where $f\colon \mathbb{R}^Y \to \mathbb{R}$ is a linear transformation. If a given $\mathbf{x}_l \in \mathbb{R}^Y$ satisfies $f(\mathbf{x}_l) > 0$, then it holds that $\Phi(\mathbf{x}_l) = \text{true}$, also written as $\mathbf{x}_l \models \Phi$.

The definition in Def. 2 suffices to express various properties of neural networks, such as safety. Accordingly, we define the problem of neural network verification as follows.

Definition 3 (Neural network verification)**.** Let $N\colon \mathbb{R}^{n_0} \to \mathbb{R}^Y$ be a neural network, φ be an input specification, and Φ be an output specification. The neural network verification problem assesses whether it holds that $N(\mathbf{x}_0) \models \Phi$ for any $\mathbf{x}_0 \in \varphi$.

The solvers used to solve the neural network verification problem are called *verifiers*. Formally, a verifier can be described as a function that takes as input N, φ, and Φ, and outputs either true, which indicates that N satisfies the specification, or false otherwise, with a *counterexample* as a witness of specification violation, i.e., an input $\mathbf{x} \in \mathbb{R}^{n_0}$ such that $(\mathbf{x} \in \varphi) \wedge (N(\mathbf{x}) \not\models \Phi)$.

3.2 Branch and Bound: A Complete Neural Network Verification Approach

Neural network verification is a challenging problem due to the non-linearity brought by activation functions. While exact output range analysis is often not scalable, *approximated methods* [Gehr et al. 2018; Singh et al. 2018, 2019; Wang et al. 2018b; Wong and Kolter 2018; Xue and Sun 2023; Zhang et al. 2022] are more efficient via pursuing an over-approximation of the actual output range.

Approximated methods: We denote by AppVerifier a verifier that implements an approximated method. Like other verifiers, AppVerifier takes as input a neural network N, an input specification φ, and an output specification Φ; it then computes an over-approximation of the output range of N. Notably, the output of AppVerifier consists of a tuple $\langle \hat{p}, \hat{\mathbf{x}} \rangle$, where $\hat{p} \in \mathbb{R}$ is called an *approximated lower bound* that holds $\hat{p} \leq \min_{\mathbf{x} \in \varphi} \left(f(N(\mathbf{x}))\right)$ (f is as defined in Def. 2), and $\hat{\mathbf{x}} \in \mathbb{R}^{n_0}$ is a counterexample returned only when $\hat{p} < 0$. Intuitively, $\hat{p}$ is the lower bound of f derived from the approximated output range of N. Thus, $\hat{p}$ is not greater than the actual lower bound of f derived from the exact output range of N.

Consequently, $\langle \hat{p}, \hat{\mathbf{x}} \rangle$ returned by AppVerifier can be interpreted as follows: if $\hat{p} > 0$, it implies that f's actual lower bound $\min_{\mathbf{x} \in \varphi} f(N(\mathbf{x})) > 0$ and thus N satisfies the specification; otherwise (i.e., if $\hat{p} < 0$), the result of AppVerifier cannot be used to indicate the satisfaction or violation of N, and we need to check the validity of $\hat{\mathbf{x}}$ returned by AppVerifier: if $\hat{\mathbf{x}}$ is a real counterexample, then N violates the specification; otherwise if $\hat{\mathbf{x}}$ is spurious one, we cannot infer whether N satisfies or violates the specification by the result of AppVerifier. In other words, AppVerifier is *sound* in the sense that $\hat{p} > 0$ implies that N indeed satisfies the specification, but AppVerifier is *not complete* as it could possibly return a spurious counterexample.

These approximation strategies are designed to handle non-linear components in a neural network, namely, the activation functions in each neuron, and typically, they use linear approximations to bound the output range of each activation function. There have been various approximation strategies that differ in their approximation precision and computational efficiency, such as *DeepPoly* [Singh et al. 2019] and *ReluVal* [Wang et al. 2018b]. Note that our proposed technique is

orthogonal to these approaches, and it is not limited to a specific approximated verifier; nevertheless, we assume that our selected AppVerifier satisfies Assump. 1 as follows.

Assumption 1 Let Ω_1 and Ω_2 be the original output regions of two different neural networks, and they hold the relation $\Omega_1 \subset \Omega_2$. By applying the same approximated verifier AppVerifier, we obtain two approximated regions $\hat{\Omega}_1$ and $\hat{\Omega}_2$. Then, $\hat{\Omega}_1$ and $\hat{\Omega}_2$ should hold the relation $\hat{\Omega}_1 \subseteq \hat{\Omega}_2$.

Essentially, Assump. 1 states that the over-approximation produced by AppVerifier changes monotonically w.r.t. the actual region. In our context, the approximated region by AppVerifier should be correlated with the actual reachable region of the neural network.

Branch and Bound (BaB): As stated, AppVerifier is efficient but incomplete, which may produce spurious counterexamples. To ensure completeness, Bunel et al. [Bunel et al. 2020] combine these approximated methods with BaB that splits the problem into sub-problems and then applies AppVerifier to solve each of them. In this way, the imprecision brought by AppVerifier can be mitigated when it is applied to sub-problems, thus reducing the possibility of spurious counterexamples. There are different problem splitting strategies, such as input space splitting [Wang et al. 2018b] and ReLU splitting [Bunel et al. 2020; Ferrari et al. 2022; Wang et al. 2018a]. In this paper, we adopt the latter, which is scalable for verification with high-dimensional inputs.

We introduce ReLU specifications, which formalize the constraints that identify the sub-problems of neural network verification under ReLU splitting.

Definition 4 (ReLU specification)**.** Let N be a neural network with K neurons, and $\hat{x}_i \in \mathbb{R}$ ($i \in \{1, \ldots, K\}$) be the input for the ReLU function in neuron i. An *atomic proposition* AP w.r.t. neuron i is defined as either $\hat{x}_i \geq 0$ (written as r_i^+) or $\hat{x}_i < 0$ (written as r_i^-). Then, a ReLU specification Γ is defined as the conjunction of a sequence $\mathrm{Set}(\Gamma) = \{\mathsf{AP}_1, \ldots, \mathsf{AP}_{|\Gamma|}\}$ of atomic propositions, where each $\mathsf{AP} \in \mathrm{Set}(\Gamma)$ is defined w.r.t. a distinct neuron. $|\Gamma|$ is the number of atomic propositions in $\mathrm{Set}(\Gamma)$; specially if $|\Gamma| = 0$, Γ is denoted as $\top$. Moreover, we define a *refinement relation* $\prec$ over ReLU specifications, namely, given two ReLU specifications Γ_i and Γ_j, we say Γ_j refines Γ_i (denoted as $\Gamma_i \prec \Gamma_j$) if and only if $\mathrm{Set}(\Gamma_i) \subset \mathrm{Set}(\Gamma_j)$.

By selecting a neuron i, the ReLU function is split to two linear functions, identified by the two input constraints r_i^+ and r_i^-. Thereby, a neural network verification problem boils down to two sub-problems, for each of which AppVerifier does not need to perform over-approximation for the ReLU of neuron i.

BaB splits the problem if AppVerifier returns an inconclusive result with a spurious counterexample, and then applies AppVerifier to the sub-problems. Note that here AppVerifier is allowed to receive an additional argument, i.e., a ReLU specification Γ, and it holds that if $\Gamma_1 \prec \Gamma_2$, then the approximated lower bound $\hat{p}_2$ from $\mathrm{AppVerifier}(N, \varphi, \Phi, \Gamma_2)$ is tighter than $\hat{p}_1$ from $\mathrm{AppVerifier}(N, \varphi, \Phi, \Gamma_1)$ (i.e., $\hat{p}_2 > \hat{p}_1$). According to literature [Bunel et al. 2020; Henriksen and Lomuscio 2021], the neuron (thus the ReLU) selected to be split is crucial to the performance of BaB. In this paper, we encode an existing ReLU selection heuristic [Henriksen and Lomuscio 2021] as a function H that, given a ReLU specification Γ (that involves a sequence of ReLUs), decides the next ReLU to split.

Alg. 1 presents the BaB algorithm. As shown in Line 2 and 3, the neural network verification problem is solved by calling the function BaB with the initial ReLU specification $\top$. The function BaB is described in Lines 4–17. First, BaB selects the ReLU specification Γ at the head of Q and applies AppVerifier (Line 7) to assess the satisfaction of N under Γ (Line 8). If the result is true, then the problem identified by Γ is verified and BaB checks the next sub-problem (Line 17). Otherwise, BaB checks whether the counterexample $\hat{\mathbf{x}}$ returned by AppVerifier is a real one, by feeding $\hat{\mathbf{x}}$ to N and checking whether the output $N(\hat{\mathbf{x}})$ violates the output specification (Line 11). If $\hat{\mathbf{x}}$ is a real counterexample, BaB returns false (Line 12), and this can back-propagate to the call of BaB with $\top$

Algorithm 1 Branch and Bound (BaB)

Require: a neural network N, an input specification φ, an output specification Φ, an approximated verifier AppVerifier$(\cdot)$, and a ReLU selection heuristic $\mathsf{H}(\cdot)$.
Ensure: a *verdict* $\in$ {true, false} and a specification tree $\mathcal{T}$

```
1:  T ← ∅                                   ▷ initialize the specification tree T storing problem splitting history
2:  Q ← {⊤}                                 ▷ initialize a queue storing ReLU specifications to be evaluated
3:  verdict ← BaB(N, φ, Φ, Q)               ▷ call BaB with the ReLU specification ⊤
4:  function BaB(N, φ, Φ, Q)
5:      if Empty(Q) then
6:          return true                     ▷ all (sub-)problems are verified
7:      Γ ← Pop(Q)                          ▷ pop the head of Q to verify
8:      ⟨p̂, x̂⟩ ← AppVerifier(N, φ, Φ, Γ)    ▷ call the approximated verifier with Γ
9:      T ← T ∪ {⟨Γ, p̂⟩}                    ▷ add node ⟨Γ, p̂⟩ in the specification tree
10:     if p̂ < 0 then
11:         if Valid(x̂, N, Φ) then          ▷ x̂ is a real counterexample
12:             return false                ▷ return false
13:         else                            ▷ x̂ is a spurious counterexample
14:             r_k ← H(Γ)                  ▷ obtain the next ReLU to be split
15:             for a ∈ {r_k^+, r_k^-} do
16:                 Push(Q, Γ ∧ a)          ▷ push sub-problems to Q for checking later
17:     return BaB(N, φ, Φ, Q)              ▷ recursive call to check remaining sub-problems
```

in Line 3, so the algorithm returns false. Otherwise, it needs to split the problem further to refine the results. To that end, BaB applies H to obtain the next ReLU r_k to be split (Line 14), and then pushes the sub-problems of $\Gamma \wedge r_k^+$ and $\Gamma \wedge r_k^-$ to Q for checking them later (Line 17). In this case, BaB also recursively calls itself to assess the remaining sub-problems. If none of the sub-problems reports a real counterexample, finally Q will be empty and the algorithm returns true (Line 6).

Note that during the execution of BaB, Alg. 1 maintains a structure $\mathcal{T}$, called a *specification tree*, which is initialized in Line 1 and updated in Line 9 to record each time when an AppVerifier is applied. The node of the tree consists of a tuple $\langle \Gamma, \hat{p} \rangle$, where Γ identifies the path of the node and $\hat{p}$ is the approximated lower bound of the sub-problem. As shown later in §4, incremental verification reuses this tree built on a neural network N in the verification of a new neural network N^*.

Proposition 1 Let $\mathcal{T}$ be a specification tree and $\langle \Gamma_1, \hat{p}_1 \rangle, \langle \Gamma_2, \hat{p}_2 \rangle \in \mathcal{T}$ be two nodes. If Γ_2 refines Γ_1 (i.e., $|\Gamma_1| < |\Gamma_2|$), then it holds that $\hat{p}_1 \leq \hat{p}_2$.

Intuitively, if Γ_2 refines Γ_1, the related sub-problem of Γ_2 will derive an actual region Ω_2 that is a subset of the region Ω_1 derived from the related sub-problem of Γ_1. By Assump. 1, the approximated region $\hat{\Omega}_2$ during verification under Γ_2 is also a subset of the approximated region $\hat{\Omega}_1$ of Γ_1, and therefore it holds that $\hat{p}_1 \leq \hat{p}_2$.

Lemma 1 The BaB algorithm is sound and complete.

Proof sketch. The soundness of BaB relies on that: 1) AppVerifier is sound; 2) the ReLU specifications identified by the leaf nodes of the specification tree $\mathcal{T}$ cover all the cases about the input conditions of the split ReLUs in the neural network.

The completeness of BaB relies on that: a counterexample $\hat{\mathbf{x}}$ reported by BaB must be a real one because $\hat{\mathbf{x}}$ has been validated by BaB in line 11. Moreover, since the number of ReLUs in a neural network N is finite and in the worst case the sub-problem will be a linear one, BaB can eventually be terminated and a real counterexample is guaranteed to be found if it exists in a branch. □

4 The Proposed Incremental Verification Approach

4.1 Incremental Verification of Neural Networks

Incremental verification is an emerging approach that aims to accelerate verification of neural networks, and a typical usage scenario is stated as follows. Let N and N^* be two neural networks that have the same model architecture (namely, same number of layers and same number of neurons at each layer) and only differ slightly[1] in their parameters. This can happen in many cases; for instance, in model repair, N^* can be obtained by fine-tuning the parameters of N. Incremental verification aims to accelerate the verification of N^* for the same input and output specification as that of N, by reusing the results and information from the verification of N.

Template reuse in Ivan: Ivan [Ugare et al. 2023] is the state-of-the-art incremental verification approach. In Ivan, N is first verified using the BaB algorithm in Alg. 1. During this verification, a specification tree $\mathcal{T}$, consisting of a set of tuples $\langle \Gamma, \hat{p} \rangle$, is maintained to record the process of problem splitting for the verification of N. This specification tree is then used as a *template* for accelerating the verification of N^*.

Specifically, the heuristic of reusing $\mathcal{T}$ in the verification of N^* by Ivan is as shown in Fig. 2b: instead of verifying N^* from scratch, it skips the sub-problems identified by the internal nodes in $\mathcal{T}$ and directly starts with the sub-problems identified by the leaf nodes. This heuristic originates from the similarity between N and N^*, namely, the internal nodes that require further splitting in the verification of N may also need to be split in the verification of N^*; since each call of AppVerifier is costly, these internal nodes are *not necessary* to be assessed again and the verification of N^* can directly start from the sub-problems identified by those leaf nodes in $\mathcal{T}$.

Motivation: Ivan exemplifies an effective strategy of leveraging the specification tree to skip potentially unnecessary sub-problems. However, the information hidden in the specification tree of N is not sufficiently exploited. One such important information consists of the *order* over the nodes in the specification tree, and specifically, for acceleration of verification, the order can be associated with the potentiality of containing a real counterexample in a path of the tree.

Exploitation of such an order in specification tree expansion is useful for accelerating the verification of N^*, because as soon as a real counterexample is found, it is sufficient for BaB to report the verdict of the verification problem, and so BaB can be terminated. If N^* indeed contains such a counterexample, the earlier it can be found, the earlier BaB can be terminated. Even though there exists no counterexample in N^*, expanding the specification tree of N^* according to the order associated to counterexample potentiality does not hinder the verification efficiency, because in that case, verification needs to cover the same number of sub-problems as the normal BaB does, but they only differ in the order of visiting the sub-problems, and so there should be no significant difference in terms of efficiency. In §4.2, we introduce an order in that spirit, followed with two incremental verification approaches, i.e., Olive$^{\text{g}}$ in §4.3 and Olive$^{\text{b}}$ in §4.4.

4.2 Counterexample Potentiality Order in Specification Tree

We introduce the order associated with the counterexample potentiality of each node in the specification tree. Our definition of the order concerns with two reference node attributes, namely *node depth* and *approximated lower bound*.

Node depth Intuitively, in the specification tree of N, the depth $|\Gamma|$ of a node Γ indicates the necessity of problem refinement because all the sub-problems identified by its ancestors cannot be verified. Moreover, a more refined sub-problem introduces less approximation

[1] In practice, such differences often come from small tuning to neural network parameters and thus they are measured empirically by engineers. In §5, we experimentally evaluate the level of similarity within which our approach is effective.

imprecision during the application of AppVerifier. So, if a counterexample $\hat{\mathbf{x}}$ is found with such a problem, $\hat{\mathbf{x}}$ is more likely to be a real one. Hence, node depth can be considered as an indicator of the potentiality of finding real counterexamples.

Approximated lower bound The approximated lower bound $\hat{p}$ obtained by application of AppVerifier to a node can also be considered as an indicator that shows how far the sub-problem identified by the node is from violating the specification. This is because $\hat{p}$ is computed based on the approximated reachable region of a neural network, and according to Assump. 1, this approximated reachable region is correlated to the ground truth reachable region of the neural network. To that end, the less the approximated lower bound is, the more likely the sub-problem can provide a counterexample.

The relation between these two attributes has been stated in Prop. 1, i.e., the approximated lower bound monotonically increases w.r.t. the depth of tree nodes. In this regard, it is difficult to infer which node is more likely to contain a counterexample, if a node $\langle \Gamma_1, \hat{p}_1 \rangle$ is superior than another node $\langle \Gamma_2, \hat{p}_2 \rangle$ in only one of the attributes (e.g., $|\Gamma_1| > |\Gamma_2|$ but $\hat{p}_1 > \hat{p}_2$). Based on this insight, we formalize the relation over the nodes in a specification tree, as a partial order, called *counterexample potentiality order* as defined in Def. 5.

Definition 5 (Counterexample potentiality order)**.** Let $\langle \Gamma_1, \hat{p}_1 \rangle$ and $\langle \Gamma_2, \hat{p}_2 \rangle$ be two nodes in a specification tree. A *counterexample potentiality order* $\sqsubset$ between the two nodes is defined as follows: $\langle \Gamma_1, \hat{p}_1 \rangle \sqsubset \langle \Gamma_2, \hat{p}_2 \rangle$ if and only if it holds one of the following two cases: 1) $|\Gamma_1| < |\Gamma_2|$ and $\hat{p}_1 \geq \hat{p}_2$; 2) $|\Gamma_1| \leq |\Gamma_2|$ and $\hat{p}_1 > \hat{p}_2$.

Note that $\sqsubset$ is a partial order, because there exists the case when two nodes are not comparable w.r.t. $\sqsubset$, e.g., when $|\Gamma_1| > |\Gamma_2|$ but $\hat{p}_1 > \hat{p}_2$. In §4.3 and §4.4, we present two different strategies that exploit this order for achieving efficient incremental verification.

4.3 Olive^{g}: Greedy Strategy of Olive

In this section, we present Olive^{g} that adopts a greedy strategy. In line with the counterexample potentiality order, Olive^{g} prioritizes the node that is more likely to contain a counterexample to terminate as early as possible. Specifically, over the two attributes in the counterexample potentiality order, namely, node depth and approximated lower bound, Olive^{g} favors the former one, because it is more likely to find a real counterexample by solving a more refined sub-problem. To that end, we tighten the partial order $\sqsubset$ in Def. 5 to be a total order $\sqsubset_G$, called *greedy counterexample potentiality order*, as shown in Def. 6.

Definition 6 (Greedy counterexample potentiality order)**.** Let $\langle \Gamma_1, \hat{p}_1 \rangle$ and $\langle \Gamma_2, \hat{p}_2 \rangle$ be two nodes in a specification tree. A *greedy counterexample potentiality order* $\sqsubset_G$ between the two nodes is defined as follows: $\langle \Gamma_1, \hat{p}_1 \rangle \sqsubset_G \langle \Gamma_2, \hat{p}_2 \rangle$ if and only if it holds one of the following two cases: (1) $|\Gamma_1| < |\Gamma_2|$; (2) $|\Gamma_1| = |\Gamma_2|$ and $\hat{p}_1 > \hat{p}_2$.

Note that $\sqsubset_G$ is a total order[2] in the sense that given any two nodes $\langle \Gamma_1, \hat{p}_1 \rangle$ and $\langle \Gamma_2, \hat{p}_2 \rangle$, it holds that either $\langle \Gamma_1, \hat{p}_1 \rangle \sqsubset_G \langle \Gamma_2, \hat{p}_2 \rangle$ or $\langle \Gamma_2, \hat{p}_2 \rangle \sqsubset_G \langle \Gamma_1, \hat{p}_1 \rangle$.

Algorithm of Olive^{g}**:** The proposed Olive^{g} algorithm is presented in Alg. 2. In addition to the inputs for the BaB algorithm (see Alg. 1), it also requires a neural network N^* to be verified, the total order $\sqsubset_G$ defined in Def. 6, and the set $\mathcal{L}$ of the leaf nodes in the specification tree $\mathcal{T}_N$ of N. Initially, Olive^{g} feeds all the leaf nodes of $\mathcal{T}_N$ to a set S (Line 3). Then it enters a loop that consists of the main function of Olive^{g}. In the loop, first, the greatest element w.r.t. $\sqsubset_G$ is selected and then

[2]There could exist the case where two nodes $\langle \Gamma_1, \hat{p}_1 \rangle$ and $\langle \Gamma_2, \hat{p}_2 \rangle$ are not comparable under $\sqsubset_G$ because $|\Gamma_1| = |\Gamma_2|$ and $\hat{p}_1 = \hat{p}_2$. To handle that case, we can take a trivial measure, e.g., randomly assigning a fixed order between them.

Algorithm 2 Olive^{g}: the greedy strategy

Require: two neural networks N and N^*, input and output specification φ and Φ, an approximated verifier AppVerifier$(\cdot)$, a ReLU selection heuristic $\mathsf{H}(\cdot)$, the set $\mathcal{L}$ of leaf nodes in specification tree $\mathcal{T}_N$ of N, a total order $\sqsubset_G$ over tree nodes.
Ensure: a *verdict* $\in \{\text{true}, \text{false}\}$ indicating the satisfaction of N^*

1: $S \leftarrow \emptyset$ ▷ initialize S to be an empty set
2: **for** $\langle \Gamma, \hat{p} \rangle \in \mathcal{L}$ **do**
3: $S \leftarrow S \cup \{\langle \Gamma, \hat{p} \rangle\}$ ▷ feed leaf nodes of $\mathcal{T}_N$ to S
4: **while** $S \neq \emptyset$ **do** ▷ if $S = \emptyset$, it indicates N^* is verified
5: select the maximum $\langle \Gamma, \hat{p} \rangle \in S$ as per $\sqsubset_G$
6: $S \leftarrow S \setminus \{\langle \Gamma, \hat{p} \rangle\}$ ▷ remove the selected node from S
7: **if** $\langle \Gamma, \hat{p} \rangle \in \mathcal{L}$ **then** ▷ the selected node is a leaf of $\mathcal{T}_N$
8: $\langle \hat{p}, \hat{\mathbf{x}} \rangle \leftarrow$ AppVerifier$(N^*, \varphi, \Phi, \Gamma)$ ▷ leaf nodes get assessed
9: **if** $\hat{p} < 0$ and $\textsc{Valid}(\hat{\mathbf{x}}, N^*, \Phi)$ **then** ▷ the counterexample is a real one
10: **return** false, $\hat{\mathbf{x}}$ ▷ return false verdict and counterexample
11: **if** $\hat{p} < 0$ **then** ▷ $\hat{p}$ is now a approximated lower bound for N^*
12: $r_k \leftarrow \mathsf{H}(\Gamma)$ ▷ select the next ReLU to be split
13: **for** $a \in \{r_k^+, r_k^-\}$ **do**
14: $\langle \hat{p}, \hat{\mathbf{x}} \rangle \leftarrow$ AppVerifier$(N^*, \varphi, \Phi, \Gamma \wedge a)$ ▷ apply AppVerifier with $\Gamma \wedge a$
15: **if** $\hat{p} < 0$ **then**
16: **if** $\textsc{Valid}(\hat{\mathbf{x}}, N^*, \Phi)$ **then** ▷ validate $\hat{\mathbf{x}}$
17: **return** false, $\hat{\mathbf{x}}$ ▷ return false and a counterexample
18: **else** ▷ $\hat{p} < 0$ but $\hat{\mathbf{x}}$ is spurious
19: $S \leftarrow S \cup \{\langle \Gamma \wedge a, \hat{p} \rangle\}$ ▷ add $\Gamma \wedge a$ to S

removed from S (Lines 5 and 6). Note that an element $\langle \Gamma, \hat{p} \rangle$ in S can be a leaf node from $\mathcal{T}_N$ or a node of the specification tree $\mathcal{T}_{N^*}$ of N^*. In the former case, Γ requires re-assessment by AppVerifier with N^*, and a new $\hat{p}$ is computed (Line 8). Otherwise, $\hat{p}$ is already the approximated lower bound for N^*, and thus, it requires no change. In both cases, if $\hat{p} < 0$, it implies that the problem needs to be further split, and hence Olive^{g} calls the ReLU heuristic H to select the next ReLU r_k (Line 12). Having r_k, Olive^{g} respectively applies AppVerifier to the sub-problems identified by $\Gamma \wedge r^+$ and $\Gamma \wedge r^-$ (Line 14). If any of the sub-problems returns a real counterexample, Olive^{g} can terminate the whole verification problem immediately (Line 17). If $\hat{p} < 0$, but the counterexample is a spurious one, the sub-problem is added to S (Line 19) as it is yet to be verified.

In Olive^{g}, the order $\sqsubset_G$ over the set S initially depends on the template, i.e., the specification tree of N. In each loop, it selects the node that is maximal in terms of the greedy counterexample potentiality order (thus most likely to contain a counterexample), and expands its children if the approximated lower bound $\hat{p}$ is negative and the counterexample returned by AppVerifier is spurious. Since $\sqsubset_G$ prioritizes the child that has a greater depth, Olive^{g} favors exploiting all the descendants of a node, until either a counterexample is found or the branch is verified.

Lemma 2 Olive^{g} is sound and complete.

Proof. The soundness of Olive^{g} relies on two facts: 1) the AppVerifier is sound; 2) on the termination of Olive^{g} with a true verdict, it verifies all the sub-problems in S that consists of the leaf nodes of the specification tree $\mathcal{T}_N$. According to Lemma 1, this is sufficient to verify the original problem with no ReLU splitting. The proof of the completeness of Olive^{g} is the same as the case of the original BaB as presented in Lemma 1. □

4.4 Olive^{b}: Balanced Strategy of Olive

While the greedy strategy Olive^{g} favors exploitation of suspicious sub-problems suggested by the specification tree $\mathcal{T}_N$ of N, it can fail when the suggestion given by $\mathcal{T}_N$ is not precise enough to facilitate the verification of N^*. Namely, there could exist a different branch in $\mathcal{T}_{N^*}$ other than

the one being exploited, in which counterexamples are easier to be found. This raises the classic problem of *"exploration and exploitation" trade-off* in search-based techniques. In light of this, we propose a balanced strategy, by modeling the incremental verification problem as a *multi-armed bandit (MAB)* [Slivkins et al. 2019] problem.

Multi-Armed Bandits: The MAB problem originally concerns a situation in which a gambler faces a row of slot machines (called *arms*), each of which has an unknown reward distribution. With the goal of maximizing the rewards, the gambler may either *explore*, i.e., try different arms to perceive which one is the best, or *exploit*, i.e., focus on the arm that has been observed as the best. In literature [Browne et al. 2012], *UCB1* is a recognized strategy which can achieve a balance between exploration and exploitation. It prioritizes not only the arms that have better rewards (exploitation), but also the arms that have been visited less (exploration).

In incremental verification, we treat the sub-problems identified by the leaf nodes of $\mathcal{T}_N$ as different arms, which enables solving the problem in an MAB fashion, in the sense that we play with these different sub-problems with a goal of quickly exposing a counterexample. Then, a remaining problem is how to assess the "potentiality" of each arm. In the original MAB, the gambler can try all of the arms to understand whether each arm is promising. In our context, our trial involves expanding each sub-problem (identified by a leaf node of $\mathcal{T}_N$) to a sub-tree, and we associate the concept of "rewards" in the original MAB with our defined counterexample potentiality. Specifically, our tree expansion follows *Monte Carlo tree search (MCTS)* [Browne et al. 2012] to achieve a balance between exploration and exploitation. We present our technical details in the following.

Algorithm of Olive$^{\text{b}}$**:** The proposed Olive$^{\text{b}}$ algorithm is presented in Alg. 3. It consists of two phases, namely, 1) arm expansion and selection (Lines 1–10), and 2) *arm assessment* (Lines 12–23). During the execution of the algorithm, the two phases alternate, where Phase 1 selects an arm (i.e., a leaf node in $\mathcal{T}_N$) and passes it to Phase 2 and Phase 2 exploits the arm to give an assessment; then, based on the assessment from Phase 2, Phase 1 continues arm selection. In the following, we introduce the details of the two phases:

- *Arm expansion and selection.* In this phase, Alg. 3 expands the arms by adding them into a work set W, and at each loop, selects an arm to assess. A notable difference of our problem from the original MAB consists in that, although the reward distributions of our arms are still unknown, we have an initial order of counterexample potentiality (e.g., the greedy one in Def. 6) over the arms, suggested by the specification tree $\mathcal{T}_N$ of N, to allow us to expand new arms progressively and orderly. This insight is integrated with the *progressive widening* [Browne et al. 2012] technique, which involves a condition (as in Line 3) that decides the timing of adding new arms into W:
  - If arm expansion is necessary (Line 3), a new node Γ from the leaf set $\mathcal{L}$ of $\mathcal{T}_N$ is selected and added to W (Line 4). Then, AppVerifier is applied to the problem identified by Γ (Line 5), and a reward is computed accordingly (Line 6). The nodes of the tree with root Γ is recorded in $\mathsf{T}(\Gamma)$ (Line 7). The computation of reward will be elaborated on later.
  - Otherwise, we select an arm Γ^* for further assessment, according to UCB1 (Line 9) that considers both the reward and the number of visits of the arms. We perform the assessment by passing the selected arm Γ^* as the argument to call the function AssessByMCTS (line 10).
- *Arm assessment.* To assess an arm Γ, we expand the sub-tree with Γ as its root, following the *Monte Carlo tree search (MCTS)* [Browne et al. 2012] policies to strike a balance between exploration and exploitation such that the assessment about the sub-tree can be more faithful. Given an arm Γ, Alg. 3 first checks if the children of Γ have been expanded (Line 14):
  - If so, it selects a child a^* of Γ by UCB1 (Line 15), and then recursively calls AssessByMCTS with $\Gamma \wedge a^*$, until it reaches a node whose children have not been expanded yet (Line 16);

Algorithm 3 Olive$^{\text{b}}$: the balanced strategy

Require: two neural networks N and N^*, input and output specification φ and Φ, an approximated verifier AppVerifier$(\cdot)$, a ReLU selection heuristic $\mathsf{H}(\cdot)$, the set $\mathcal{L}$ of the leaf nodes in specification tree $\mathcal{T}_N$ of N, hyperparameters C, α and c.
Ensure: a *verdict* $\in \{\text{true}, \text{false}\}$ indicating the satisfaction of N^*

1: $\mathsf{W} \leftarrow \emptyset$ ▷ initialize a work set W
2: **while** termination condition not reached **do**
3: **if** $|\mathsf{W}| \leq C \cdot (\sum_{\Gamma \in \mathsf{W}} |\mathsf{T}(\Gamma)|)^{\alpha}$ and $|\mathsf{W}| \neq |\mathcal{L}|$ **then** ▷ needed to try new leaf
4: add to W the maximum $\langle \Gamma, \hat{p} \rangle \in \mathcal{L}$ not in W yet, as per $\sqsubset_G$
5: $\langle \hat{p}, \hat{\mathbf{x}} \rangle \leftarrow \text{AppVerifier}(N^*, \varphi, \Phi, \Gamma)$ ▷ apply AppVerifier with Γ
6: $\mathsf{R}(\Gamma) \leftarrow \text{ComputeReward}\,(|\Gamma|, \hat{p})$ ▷ compute reward
7: $\mathsf{T}(\Gamma) \leftarrow \{\Gamma\}$ ▷ initialize the tree of Γ
8: **else**
9: $\Gamma^* \leftarrow \underset{\Gamma \in \mathsf{W}}{\arg\max} \left(\mathsf{R}(\Gamma) + c \sqrt{\frac{2 \ln \left(\sum_{\Gamma \in \mathsf{W}} \left| \mathsf{T}(\Gamma) \right| \right)}{\left| \mathsf{T}(\Gamma) \right|}} \right)$ ▷ select Γ^* by UCB1
10: AssessByMCTS(Γ^*) ▷ call MCTS to assess Γ
11: **return** $\begin{cases} \text{true} & \text{if } \forall \langle \Gamma, \hat{p} \rangle \in \mathcal{L}.\ \mathsf{R}(\Gamma) = -\infty \\ \text{false} & \text{if } \exists \langle \Gamma, \hat{p} \rangle \in \mathcal{L}.\ \mathsf{R}(\Gamma) = +\infty \end{cases}$ ▷ return a verdict
12: **function** AssessByMCTS(Γ)
13: $r_k \leftarrow \mathsf{H}(\Gamma)$ ▷ select r_k based on current ReLU spec.
14: **if** $(\Gamma \wedge r_k^+) \in \mathsf{T}(\Gamma)$ **then** ▷ children of Γ already expanded
15: $a^* \leftarrow \underset{a \in \{r_k^+, r_k^-\}}{\arg\max} \left(\mathsf{R}(\Gamma \wedge a) + c \sqrt{\frac{2 \ln \left| \mathsf{T}(\Gamma) \right|}{\left| \mathsf{T}(\Gamma \wedge a) \right|}} \right)$ ▷ select a child by UCB1
16: AssessByMCTS($\Gamma \wedge a^*$) ▷ assess the sub-tree with root $\Gamma \wedge a^*$
17: **else**
18: **for** $a \in \{r_k^+, r_k^-\}$ **do**
19: $\langle \hat{p}, \hat{\mathbf{x}} \rangle \leftarrow \text{AppVerifier}(N^*, \varphi, \Phi, \Gamma \wedge a)$ ▷ apply AppVerifier with $\Gamma \wedge a$
20: $\mathsf{R}(\Gamma \wedge a) \leftarrow \text{ComputeReward}\,(|\Gamma|, \hat{p})$ ▷ compute reward
21: $\mathsf{T}(\Gamma \wedge a) \leftarrow \{\Gamma \wedge a\}$ ▷ initialize the tree of $\Gamma \wedge a$
22: $\mathsf{R}(\Gamma) \leftarrow \max_{a \in \left\{r_k^+, r_k^-\right\}} \mathsf{R}(\Gamma \wedge a)$ ▷ back-propagate the reward of Γ
23: $\mathsf{T}(\Gamma) \leftarrow \mathsf{T}(\Gamma) \cup \mathsf{T}\left(\Gamma \wedge r_k^+\right) \cup \mathsf{T}\left(\Gamma \wedge r_k^-\right)$ ▷ back-propagate the tree

– Otherwise, it expands both children of Γ, by applying AppVerifier to the sub-problems identified by the children (Line 19), computing the rewards accordingly (Line 20), and recording the children (Line 21). Then, it propagates the rewards of the children (Line 22) and the update of tree structure (Line 23), backwards along the path until the root (i.e., the arm).

Reward computation: In Alg. 3, a key ingredient is the use of rewards to signify the *values* of exploiting a tree node. Our reward is computed as follows:

$$\text{ComputeReward}(|\Gamma|, \hat{p}) := \begin{cases} +\infty & \text{if } \hat{p} < 0 \text{ and } \textsc{Valid}(\hat{\mathbf{x}}, N^*, \Phi) \\ -\infty & \text{if } \hat{p} > 0 \\ \sigma \frac{|\Gamma|}{K} + (1 - \sigma) \frac{\hat{p}}{\hat{p}_{\min}} & \text{otherwise} \end{cases}$$

where K is the total number of neurons (i.e., ReLUs), $\hat{p}_{\min}$ is the minimal approximated lower bound over the nodes of $\mathcal{T}_N$, and σ is a hyperparameter to tune the weights between the two terms. Intuitively, the reward reflects how promising a node is to find a real counterexample: if a counterexample is found in a node Γ, then we already achieved a counterexample and so the reward is $+\infty$; if a node Γ is verified, then there is no more chance to find a counterexample with the node and so its reward is $-\infty$; otherwise, Γ still needs to be exploited, and the priority is positively correlated to $|\Gamma|$ and negatively related to $\hat{p}$ (note that $\hat{p}_{\min}$ should be a negative value).

Table 1. The details of the benchmarks adopted in our experiments

Model (N)	Architecture	Neurons	Dataset	Problem Instances
$\texttt{MNIST}_{L2}$	2× 256 fully-connected layers	512	MNIST	241
$\texttt{MNIST}_{L4}$	4× 256 fully-connected layers	1024	MNIST	675
$\texttt{OVAL21}_{\texttt{BASE}}$	2 Conv, 2 fully-connected layers	4582	CIFAR-10	173
$\texttt{OVAL21}_{\texttt{WIDE}}$	2 Conv, 2 fully-connected layers	6244	CIFAR-10	207
$\texttt{OVAL21}_{\texttt{DEEP}}$	4 Conv, 2 fully-connected layers	6756	CIFAR-10	149

Note that by Line 22 of Alg. 3, the reward of a node Γ is back-propagated to the ancestor nodes of Γ, by taking the maximum of the rewards of children. Based on this mechanism, our definition of reward can help to decide the termination condition of the verification. By our definition, if a counterexample is found with Γ, its reward $+\infty$ will be in any case propagated to the root; if Γ is verified, the reward is $-\infty$ and so this node will be ignored in the following verification phase; otherwise, the reward of ancestor nodes depend on the offspring node that has the greatest potentiality to contain a counterexample.

Termination and return: In line with the definition of rewards, the termination condition of Alg. 3 involves the following two cases: 1) $\forall\langle\Gamma,\hat{p}\rangle\in\mathcal{L}.\ \big(\mathrm{R}(\Gamma)=-\infty\big)$, it means that all of the leaf nodes in $\mathcal{T}_N$ have been verified, and so verification can be terminated; 2) $\exists\langle\Gamma,\hat{p}\rangle\in\mathcal{L}.\ \big(\mathrm{R}(\Gamma)\big)=+\infty$, it means that there must exist a leaf node in $\mathcal{T}_N$, in which a real counterexample has been detected, and so verification can be terminated. Then, Alg. 3 returns the verdict accordingly (Line 11): in Case 1), true is returned as all the sub-problems identified by the leaf node of $\mathcal{T}_N$ are verified; in Case 2), false is returned because a real counterexample is found.

Lemma 3 Olive$^{\mathrm{b}}$ is sound and complete.

Proof. The soundness of Olive$^{\mathrm{b}}$ relies on two facts: 1) the AppVerifier is sound; 2) on the termination of Olive$^{\mathrm{g}}$ with a true verdict, it verifies all the sub-problems identified by the leaf nodes of the specification tree $\mathcal{T}_N$. According to Lemma 1, this is sufficient to verify the original problem. The proof of the completeness of Olive$^{\mathrm{b}}$ is the same as the original BaB as presented in Lemma 1. □

5 Experimental Evaluation

5.1 Experiment Settings

In the section, we report the experiment settings we adopted to assess Olive. The source code, benchmarks, and results are publicly available in our repository (see §8), and many experimental details are accessible in our supplementary website [Zhang et al. 2025].

5.1.1 Benchmarks. In our experiments, we adopt the benchmarks that are widely-used in the neural network verification communities [Brix et al. 2023a]. An overview of the benchmarks is presented in Table 1. Below we give more details about the settings of these benchmarks.

Datasets: We perform evaluation with two datasets `MNIST` and `CIFAR-10`, which have been broadly adopted in neural network verification community [Liu et al. 2021] and used in VNN-COMP [Brix et al. 2023a], the annual competition for neural network verification. Both of the datasets concern with image classification. Specifically, `MNIST` deals with images of handwritten digits; `CIFAR-10` involves images of various real-world objects, such as airplanes, cars, and different animals.

Neural Network Models: For each dataset, we adopt multiple neural networks as the reference neural networks N in our experiments. These networks will be firstly verified using BaB to generate templates. Then their model parameters will be modified by different methods to obtain the variation neural network N^* for assessing the performances of different approaches.

- For MNIST, we adopt two neural networks $\texttt{MNIST}_{\texttt{L2}}$ and $\texttt{MNIST}_{\texttt{L4}}$, selected from VNN-COMP [Brix et al. 2023b]. Both of the networks consist of fully-connected layers, and specifically, $\texttt{MNIST}_{\texttt{L2}}$ involves 2 layers and $\texttt{MNIST}_{\texttt{L4}}$ involves 4 layers. Each layer of the models contains 256 neurons, and in total, $\texttt{MNIST}_{\texttt{L2}}$ has 512 neurons and $\texttt{MNIST}_{\texttt{L4}}$ has 1024 neurons;
- For OVAL21, we adopt three neural networks $\texttt{OVAL21}_{\texttt{BASE}}$, $\texttt{OVAL21}_{\texttt{WIDE}}$ and $\texttt{OVAL21}_{\texttt{DEEP}}$, which are *ReLU*-based convolutional neural networks trained on CIFAR-10 [Wong and Kolter 2018]. These models are also included in VNN-COMP [Brix et al. 2023b]. As shown in Table 1, the three networks differ in the model architecture, namely, compared with $\texttt{OVAL21}_{\texttt{BASE}}$, $\texttt{OVAL21}_{\texttt{WIDE}}$ has more neurons in a single layer, and $\texttt{OVAL21}_{\texttt{DEEP}}$ has more layers. As a result, $\texttt{OVAL21}_{\texttt{BASE}}$ has 4582 neurons, $\texttt{OVAL21}_{\texttt{WIDE}}$ has 6244 neurons, and $\texttt{OVAL21}_{\texttt{WIDE}}$ has 6756 neurons in total.

Model alteration methods: To obtain the altered neural network N^* from a reference neural network N, we apply the following alteration methods, following existing literature [Ugare et al. 2023]. Note that both of these alteration methods have been adopted in practice to achieve particular goals, which showcases the value of our approach in real-world scenarios.

- *Weight pruning* is used for improving generalization, inference speed-up, and training/fine-tuning with fewer samples, etc. It has been integrated as a standard library in mainstream machine learning platforms such as Pytorch. Technically, it simply removes a number of weights in a given neural network, by setting them to 0. It is subject to a parameter that indicates the percentage of the number of weights to be removed. This parameter decides the similarity between N and N^*, and so it possibly affects the effectiveness of our approach. In our experiments, for different models, we adopt different parameters, due to the different sizes of the neural networks. Specifically, for the models $\texttt{MNIST}_{\texttt{L4}}$ and $\texttt{MNIST}_{\texttt{L2}}$, we adopt 7% and 12%; for the models $\texttt{OVAL21}_{\texttt{BASE}}$, $\texttt{OVAL21}_{\texttt{WIDE}}$ and $\texttt{OVAL21}_{\texttt{DEEP}}$, we adopt smaller parameters 0.1%, 1% and 3% due to the large sizes of the models. In RQ5, we take this parameter as an indicator of the similarity between N and N^*, and take a specific look at the performance change of Olive under different parameters.
- *Quantization* is also a widely-recognized technique for the similar purposes as weight pruning. It has also been adopted in mainstream machine learning platforms such as Pytorch. Technically, it converts an original floating number to a different format using less number of bits, e.g., using 8-bit integers. We adopt such a conversion for the alteration of MNIST models. For the models of OVAL21, since performing quantization can cause a big change to model behavior due to their large numbers of model parameters, we thus apply weight pruning methods only.

Specifications: In our experiments, we evaluate the local robustness property of each model. To that end, an input specification is identified by an ϵ, which is a distance measure under L_∞ and identifies a square region in input space; an output specification is identified by a reference image $\mathbf{x}$ and requires that for any input $\mathbf{x}'$ such that $||\mathbf{x}' - \mathbf{x}||_\infty \leq \epsilon$, the model should classify $\mathbf{x}'$ to the same label with $\mathbf{x}$.

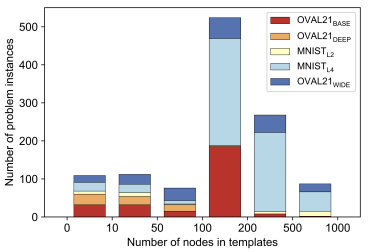


Fig. 4. The distribution of the sizes of the templates used in our experiments

To avoid meaningless specifications which can be solved by a very simple BaB tree (e.g., a tree that has only 1 node), we perform a binary search for ϵ for each reference image $\mathbf{x}$ such that the template tree produced by verification of N against $\mathbf{x}$ and ϵ has a considerable number of nodes. Note that this is necessary because, otherwise, the template can contain too few nodes, which cannot provide meaningful information for the verification of N^*. The distribution of the sizes of the template trees used in our experiments across all the benchmarks is shown in Fig. 4. By Fig. 4, we can observe

that all of the templates consist of a reasonable number of tree nodes (most of templates have tree size 100 to 500), so they can provide meaningful information for incremental verification.

5.1.2 Baseline approaches. To assess the efficiency of Olive, we compare the performance of Olive with four baseline approaches, including three classic verification approaches, namely, BaB [Bunel et al. 2020] (as introduced in Alg. 1), and other two state-of-the-art approaches Marabou [Katz et al. 2019] and $\alpha\beta$-Crown [Wang et al. 2021], which participated in VNN-COMP [Brix et al. 2023b], and one incremental verification approach Ivan [Ugare et al. 2023], as introduced in §4.1.

5.1.3 Research questions. Our evaluation aims to answer the following five research questions:

- *RQ1: Is* Olive *more efficient than existing approaches?* In this RQ, we want to compare the efficiency of Olive with baseline approaches to assess if Olive performs better than baselines.
- *RQ2: How does* Olive *perform for violated and certified verification problems respectively?* In this RQ, we want to assess the performance of Olive for the verification problem instances that violate and satisfy the specifications, respectively.
- *RQ3: Which strategy of* Olive *can achieve better performance?* In this RQ, we want to compare the performances between the two strategies of Olive, namely, Olive$^{\text{g}}$ and Olive$^{\text{b}}$.
- *RQ4: How do different hyperparameters influence the performance of* Olive$^{\text{b}}$*?* In this RQ, we aim to understand the influences of hyperparameter selections in Alg. 3 to the performance of Olive$^{\text{b}}$.
- *RQ5: How is the performance of* Olive *subject to the similarity of N^* to N?* In this RQ, we want to assess the performance of Olive under different pruning levels, to understand the necessary similarity level for Olive to be effective.

5.1.4 Evaluation metrics. For each research question, we adopt the following evaluation metrics.

- To answer RQ1, we apply all the six approaches, including two proposed approaches Olive$^{\text{g}}$ and Olive$^{\text{b}}$, and four baseline approaches, and compare their performance for verifying the problem instances presented in Table 1. Specifically, we set 1000s as timeout. Note that Ivan, Olive$^{\text{g}}$ and Olive$^{\text{b}}$ are incremental verification approaches, that require a template from N; the template is generated by running BaB, also constrained by the timeout, and all three incremental verification approaches share the same template for verification of N^*. In a comparison between two approaches, we adopt the following two evaluation metrics:
  - the number of the problems solved by one approach but not solved by another approach;
  - the time spent for solving the problem, when at least one approach manages to solve a problem instance (i.e., returns a verdict true or false). Moreover, we compute the speedup of one approach w.r.t. another approach.
- To answer RQ2, we show a breakdown result of Olive for the verification problem instances that are finally confirmed to be true and that are finally confirmed to be false. Through this RQ, we aim to understand the performance of Olive under these two different cases. As the evaluation metric, we compare the time costs of different approaches as done in RQ1.
- To answer RQ3, we adopt the same evaluation metrics as in RQ1 and RQ2, namely, we compare the time costs of the two strategies of Olive. Specifically, we check the performance differences between the two approaches on individual problem instances, and if Olive$^{\text{b}}$ is better than Olive$^{\text{g}}$, it implies that our balanced sub-problem selection strategy takes effects.
- To answer RQ4, we adopt the same metrics as RQ1, and compare the performances of Olive$^{\text{b}}$ under different settings of hyperparameters. For this RQ, we randomly select 25 instances from each of the 15 models with different alterations, leading to 375 instances in total.
- To answer RQ5, we change the parameters of weight pruning in order to change the similarity between N^* and N. Then we evaluate the performance of Olive under different similarity levels, and to which extent Olive is still useful to accelerate the verification of N^*.

Table 2. RQ1 – The performance comparison between five approaches (two classic approaches $\alpha\beta$-Crown, Marabou, and three incremental approaches Ivan, Olive^{g}, Olive^{b}), in terms of the average speedup w.r.t. BaB over the solved verification problems, and the number of the additional problems that are not solved by BaB but solved by other approaches.

Model	Alteration	$\alpha\beta$-Crown		Marabou		Ivan		Olive^{g}		Olive^{b}	
		Speedup	+Solved	Speedup	+Solved	Speedup	+Solved	Speedup	+Solved	Speedup	+Solved
MNIST_{L2}	Quant(Int8)	1.08×	0	0.93×	0	3.59×	8	3.28×	5	2.51×	3
MNIST_{L2}	Prune(7%)	1.05×	0	1.02×	0	1.60×	13	1.75×	10	1.73×	11
MNIST_{L2}	Prune(12%)	2.28×	0	1.22×	0	2.31×	0	34.89×	1	35.98×	4
MNIST_{L4}	Quant(Int8)	2.18×	0	1.28×	0	1.78×	4	2.15×	5	2.14×	6
MNIST_{L4}	Prune(7%)	1.01×	0	0.99×	0	1.59×	4	2.16×	2	2.18×	3
MNIST_{L4}	Prune(12%)	1.06×	0	1.1×	0	2.92×	2	14.74×	3	14.99×	23
OVAL21_{BASE}	Prune(0.1%)	1.01×	0	0.93×	0	1.74×	16	2.56×	11	2.58×	7
OVAL21_{BASE}	Prune(1%)	1.04×	0	0.99×	0	1.79×	17	2.37×	12	2.33×	7
OVAL21_{BASE}	Prune(3%)	1.00×	0	0.96×	0	1.77×	14	2.31×	10	2.29×	6
OVAL21_{DEEP}	Prune(0.1%)	1.02×	0	1.03×	0	1.91×	10	3.19×	8	3.21×	5
OVAL21_{DEEP}	Prune(1%)	1.56×	3	1.14×	0	1.92×	7	3.14×	5	3.18×	3
OVAL21_{DEEP}	Prune(3%)	1.03×	0	1.01×	0	1.88×	4	2.79×	2	2.78×	2
OVAL21_{WIDE}	Prune(0.1%)	1.0×	0	0.88×	0	1.98×	1	5.82×	1	5.98×	1
OVAL21_{WIDE}	Prune(1%)	0.98×	0	0.83×	0	1.74×	1	7.51×	1	7.34×	2
OVAL21_{WIDE}	Prune(3%)	1.02×	0	0.93×	0	1.88×	4	2.79×	2	2.78×	2

Software and hardware dependencies: We use Google Cloud Computing [Google LLC [n. d.]] with 60vCPU and with the memory of 240 GB running with the Ubuntu 22.04 operating system. We adopt the ReLU selection strategy (i.e., H in Alg. 1, Alg. 2 and Alg. 3) from DeepSplit [Henriksen and Lomuscio 2021]. We adopt LP-based triangle relaxation [Bunel et al. 2020] as our backend approximated verifier AppVerifier. The linearized analyzer required by the approximated verifier utilizes the Gurobi solver [Gurobi Optimization, LLC 2023].

5.2 Evaluation

RQ1 *Is* Olive *more efficient than existing approaches?*

Comparison with BaB: A performance comparison between five approaches (including two state-of-the-art classic verification approaches $\alpha\beta$-Crown and Marabou, and three incremental verification approaches Ivan, Olive^{g} and Olive^{b}) and the classic verification approach BaB, is shown in Table 2. Specifically, for approaches other than BaB, Table 2 shows the average speedup w.r.t. BaB over solved problem instances (i.e., at least one approach gives a verdict true or false) and the number of problem instances that are not solved by BaB but solved by other approaches.

The results clearly show the performance advantage of the three incremental verification approaches over BaB, in the sense that incremental verification approaches can achieve a significant speedup over BaB. Due to such speedup, for all of models, incremental approaches manage to solve more verification problem instances within the timeout. This insight is consistent with that in Ivan [Ugare et al. 2023], indicating that incremental verification that leverages existing knowledge about verification of N can indeed bring advantages in verification of a similar network N^*.

Comparison with Marabou and $\alpha\beta$-Crown: By Fig. 2, we can also see that, while $\alpha\beta$-Crown and Marabou exhibit a little performance advantage over BaB, the performances of these three approaches are comparable, and the two approaches underperform the three incremental verification approaches in general. This is expected, because although these two approaches take various strategies to improve the performance, for a given network N^*, they need to verify from scratch, which is time-consuming in any case; in contrast, incremental verification reuses the information of previous verification, which can effectively accelerate the verification of N^*. Although $\alpha\beta$-Crown

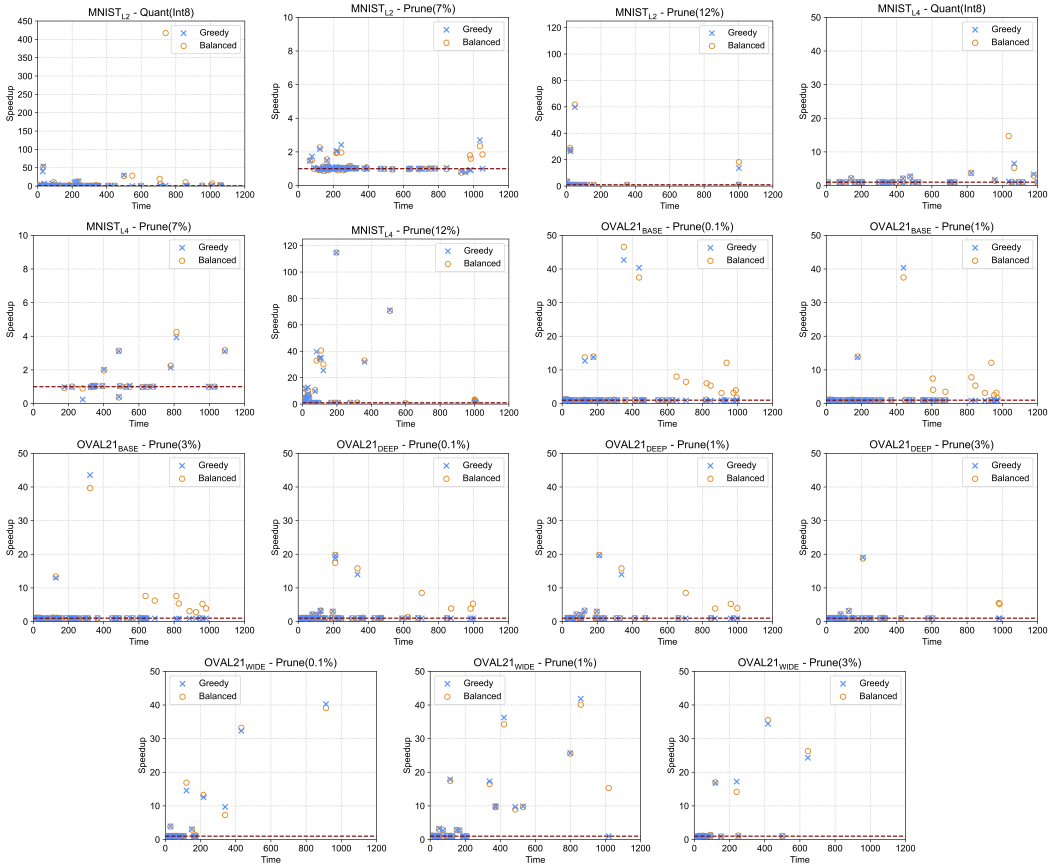


Fig. 5. RQ1&RQ3 – The comparison between our proposed approach Olive and the baseline approach Ivan, in terms of time cost and the speedup of Olive over Ivan.

indeed improves the performance of verification considerably compared to BaB, the advantage is still not significant, compared to incremental verification approaches, which highlights the usefulness of reusing information from previous verification as incremental verification does.

Comparison with Ivan: By Fig. 2, we can see that the speedup brought by our approaches Olive^{g} and Olive^{b} are mostly greater than that from Ivan. Indeed, both Olive^{g} and Olive^{b} achieve higher speedup than Ivan in 14/15 cases. Notably, in model $\texttt{MNIST}_{L2}$, Olive^{b} and Olive^{g} achieve more than 34× speedup respectively, and in model $\texttt{MNIST}_{L4}$, Olive^{b} and Olive^{g} achieve more than 14× speedup respectively, signifying a great outperformance of Olive over Ivan.

To better compare Olive and Ivan, in Fig. 5, we plot the time cost and the speedup of Olive over Ivan for each individual problem instance. In each of the sub-figures in Fig. 5, the red line signifies a speedup of 1, namely, equivalent performance between Olive and Ivan, and so the points above the red line are the problem instances in which Olive outperforms Ivan.

First, by observing Fig. 5 in most of the benchmark models, our proposed Olive achieves a higher efficiency than Ivan for many problem instances. In 13 out of 15 models, there are problem instances for which Olive achieves more than 10× speedup; specifically, in $\texttt{MNIST}_{L2}$ with `Quant(Int8)`, the speedup achieved by Olive can be up to more than 400× and in $\texttt{MNIST}_{L4}$ with `Prune(12%)`, the

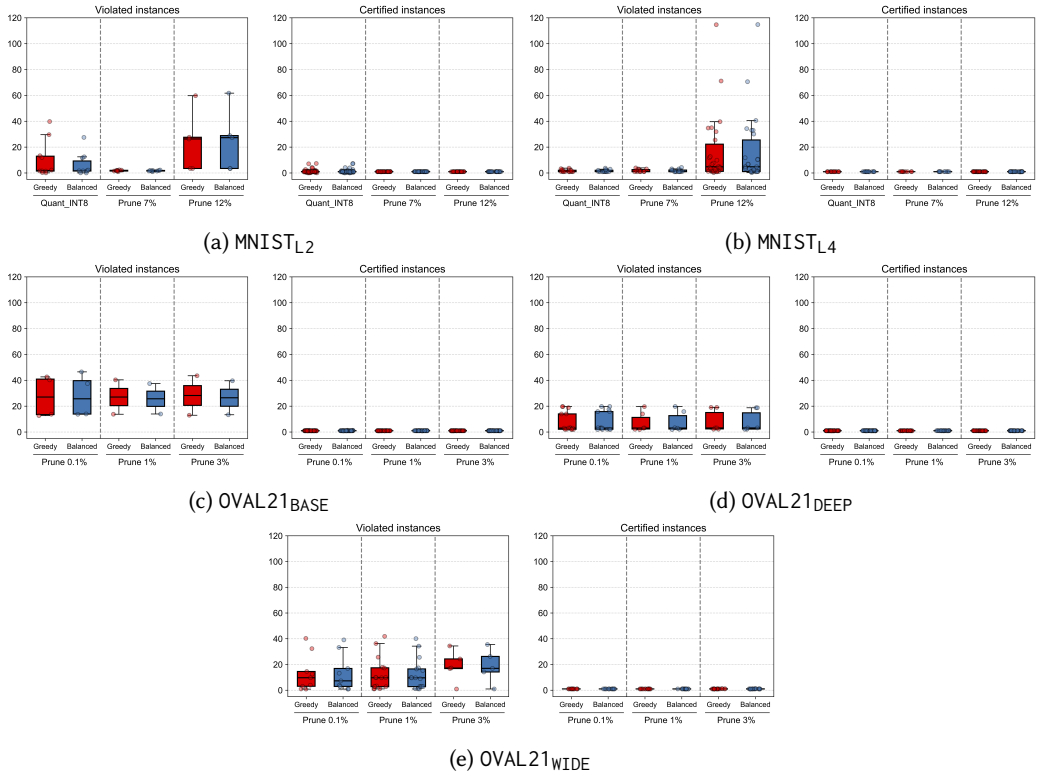


Fig. 6. RQ2&RQ3 – The breakdown result about the speedup of Olive^{g} and Olive^{b} over Ivan respectively, for violated and certified verification problem instances

speedup can be around 120×. Moreover, in MNIST$_{L2}$ with Prune(7%), MNIST$_{L4}$ with Prune(7%), OVAL21$_{BASE}$ with Prune(0.1%) and OVAL21$_{BASE}$ with Prune(1%), we can observe a significant number of problem instances for which Olive performs much faster than Ivan, and a big portion of these instances take more than 10 minutes for their verification, which signifies that they are complex verification problems. In practice, given a set of problem instances whose results are unknown beforehand, this efficiency improvement can help practitioners to quickly conclude the verification for many problem instances and certify a much greater number of problem instances, especially in the case when the problem instances are complex and so their verifications are expensive.

Second, by Fig. 5, we also find that in these problem instances, there are few instances for which Olive takes more time. Though we can observe some cases, e.g., in MNIST$_{L2}$ with Prune(7%) and MNIST$_{L4}$ with Prune(7%), the difference is not significant. There are several reasons that can cause the situation, e.g., our counterexample potentiality order does not work well due to the difference between two networks. This confirms our conjecture that while Olive can achieve better efficiency, especially when problems contain counterexamples, it can at least achieve similar performance with Ivan in general cases where there exist no counterexamples.

RQ2 *How does* Olive *perform for violated and certified verification problems respectively?*

To understand the performance of Olive under different cases, In Fig. 6, we show the breakdown results of Olive for the problem instances where there exist counterexamples (i.e., violated problem instances) and where there exists no counterexample (i.e., certified problem instances). For each

benchmark model, we produce box plots that present the speedup rates of Olive^{g} and Olive^{b}, over Ivan baseline for each individual problem instance, and the left plot in each sub-figure is the results for violated instances, and the right plot in each sub-figure is the results for certified instances.

By observing Fig. 6, we can further confirm our conjecture that Olive performs significantly better than Ivan in the problem instances where there exist counterexamples, and its performance is as good as Ivan in the certified problem instances. In Fig. 6, for all of the benchmark models, it is evident that the speedup rates for violated instances are significantly better than that of certified instances. While in most cases the distributions of speedup for certified instances are around 1, i.e., the performances of Olive for those instances are similar with Ivan, the distributions of speedup for violated instances are much greater than 1, signifying the superiority of Olive for these violated problem instances in terms of efficiency.

First, the performance advantages over the violated instances demonstrate the effectiveness of our proposed counterexample potentiality order, which manages to infer the most suspicious sub-problems and guides our search to prioritize the verification of those sub-problems. Second, even with the certified problem instances which do not contain counterexamples, our approach does not diminish the verification performance, because in this case, the only difference of Olive with Ivan involves the order they visit the necessary nodes suggested by the template tree; however, for both of the approaches, it is necessary to visit all the leaf nodes in the template tree to guarantee the soundness of the verification approach, so it does not lead to a notable performance difference.

RQ3 *Which strategy of* Olive *can achieve better performance?*

To compare the performance between Olive^{g} and Olive^{b}, we take a further look into the performance evaluation that has been done for RQ1 and RQ2.

In Fig. 5, we observe that both Olive^{g} and Olive^{b} have many points above the horizontal line of speedup 1, which indicates that both Olive^{g} and Olive^{b} hold many problem instances for which they outperform Ivan. We then count the number of the problem instances which significantly outperform Ivan for Olive^{g} and Olive^{b} respectively. We find that, first, in many benchmark models the performances between Olive^{g} and Olive^{b} are comparable, e.g., in the model $\texttt{MNIST}_{\texttt{L4}}$ with $\texttt{Prune}(12\%)$ and in the model $\texttt{OVAL21}_{\texttt{WIDE}}$ with $\texttt{Prune}(1\%)$; moreover, in general, Olive^{b} outperforms Olive^{g} in many different cases. For instance, in model $\texttt{OVAL21}_{\texttt{BASE}}$ with all different weight pruning options $\texttt{Prune}(0.1\%)$, $\texttt{Prune}(1\%)$, and $\texttt{Prune}(3\%)$, there appear a cluster of points of Olive^{b} that outperform Ivan while there is no Olive^{g} in the surroundings. These results show the performance advantages of Olive^{b} over Olive^{g}, and such superiority can be attributed to the balanced strategy of Olive^{b} that does not focus on exploitation of suspicious sub-problems only.

Fig. 6 provides another perspective to view this research question. First, we can confirm our observation that there is no evident performance difference between these two approaches in both violated problem instances and certified problem instances. Specifically for violated problem instances, such performance differences are also very small, and this can be due to that, under the specified parameters, both weight pruning and quantization do not trigger a big difference between the original model N and the updated model N^{*}. Second, the performance differences we observed in Fig. 5 are mainly from individual problem instances, while in terms of statistic metrics such as average and median of the speedup, the differences between Olive^{g} and Olive^{b} are not evident.

RQ4 *How do different hyperparameters influence the performance of* Olive^{b}*?*

Recall Alg. 3 for Olive^{b}; there are multiple hyperparameters that can influence the performance of Olive^{b}, including C, α related to progressive widening, c related to UCB1, and σ related to reward computation. To understand how these hyperparameters influence Olive^{b}, we randomly select 25 instances from each of the 15 models (including all of the models with different architectures, under

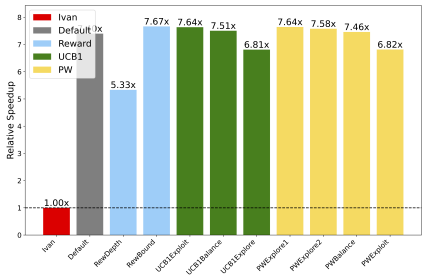


Fig. 7. RQ4 – The performance of Olive^b under different settings of hyperparameters

all different model alteration methods), leading to 375 instances in total. These instances encompass both certified and violated cases, and also some cases not solved by any approach, because they are still possible to be solved by the variant approaches.

In this study, we first have a Default configuration of Olive^b, as being used in previous RQs. The default values of hyperparameters are $(\alpha, C, \sigma, c) = (0.5, 1.2, 0.5, 0.2)$, which are obtained empirically to achieve balance between exploration and exploitation in different phases, based on existing literature [Browne et al. 2012]. Then, we vary all of the hyperparameters with respect to Default, and change one hyperparameter at a time, while keeping others at their default values, such that we can study the influence of the particular hyperparameters on the performance of Olive^b.

Table 3 summarizes the hyperparameter settings in our study, which lead to different variants of our approach. The first column gives an ID to Ivan baseline and the variants of our approach; specifically, the prefixes of the IDs indicate the components of Alg. 3 being varied in each variant, including "Rew" (reward computation), "UCB1" (the UCB1 algorithm) and "PW" (progressive widening). The second column and the third column respectively identify the hyperparameters being varied and the values of the hyperparameters in that configuration. In particular, for the Default configuration as shown in the second row of Table 3, we list out all of the four hyperparameters that can be varied by different variants and the default values for these hyperparameters; for the variants (after the second row), we only show the relevant hyperparameters being changed with respect to Default, and the value of the hyperparameters after change.

Table 3. Hyperparameter settings

Variants	Rel. params	Value
Ivan	-	-
Default	(α, C, σ, c)	(0.5, 1.2, 0.5, 0.2)
RewDepth	(σ)	0
RewBound	(σ)	1
UCB1Exploit	(c)	0
UCB1Balance	(c)	0.4
UCB1Explore	(c)	1
PWExplore1	(α, C)	(0.5, 2)
PWExplore2	(α, C)	(1, 1.2)
PWBalance	(α, C)	(0.2, 1.2)
PWExploit	(α, C)	(0.5, 0.5)

The comparison between different variants of Olive^b and Ivan is presented in Fig. 7. First, we can observe that, the range of speedup is between 5.33× (by RewDepth) and 7.67× (by RewBound), which exhibits a consistent performance advantage of Olive^b over Ivan. Specifically,

- RewBound, which defines reward purely by approximated lower bounds, outperforms the default configuration; in contrast, RewDepth, which defines reward purely by node depth, underperforms the default configuration. First, the result suggests the effectiveness of approximated lower bounds in guiding the search of counterexample. While this may suggest that defining rewards purely by approximated lower bounds is good, the merit can be attributed to a "balance" flavor of that setting, because approximated lower bounds are regardless of node depth. In general cases, the optimal ratio between these two attributes may still depend on concrete problems.
- Different settings regarding UCB1 perform very similarly, and all of them slightly are comparable with the default setting. In particular, UCB1Explore exhibits the worst performance, which indicates that too much emphasis on exploration may not bring good performance.

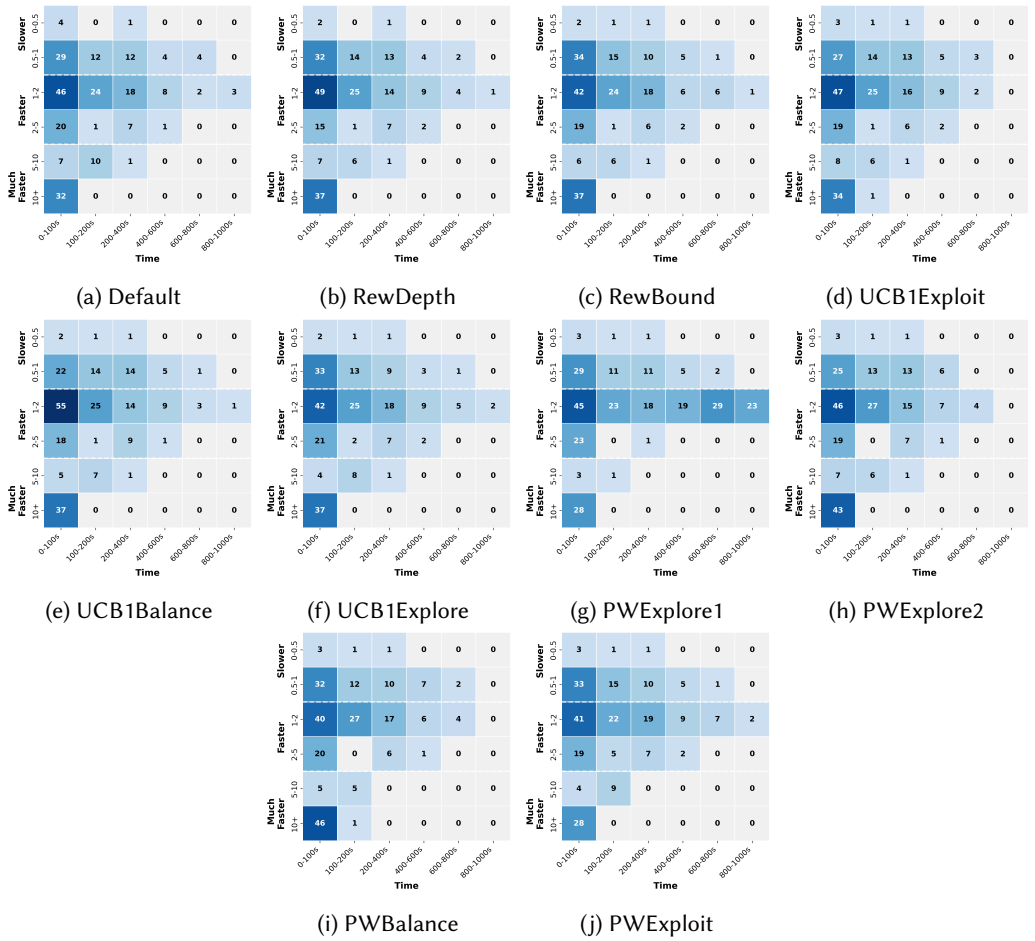


Fig. 8. RQ4 – The distributions of solved instances by variants of Olive[b], in different speedups and time costs

- Different settings regarding progressive widening are also comparable with the default setting, but PWExploit can be a bit worse. This is expected, because by PWExploit, the progressive widening is too greedy and does not often expand a new arm, thus missing the chance of finding a counterexample in other arms than the one suggested by the template.

To further study the performances of different variants, we plot a heatmap for each variant that shows the distributions of the numbers of solved instances by different variants in different ranges of speedups and time costs. The results are presented in Fig. 8. Compared to Default:

- RewDepth shows a slight improvement (49 instances in 0-100s) in quick solutions, due to its greediness, but its effectiveness diminishes for longer-running instances with only 14 solved in the 200-400s range, which suggests a limitation in capturing problem complexity.
- RewBound exhibits enhanced performance for medium-difficulty problems, solving 42 instances in 0-100s and notably 18 instances in the 200-400s range, indicating that defining reward in this way provides effective guidance for more complex problems.

Table 4. RQ5 – Performance change (in terms of average speedup over BaB) of our proposed approaches on $\texttt{OVAL21}_{\texttt{BASE}}$ and $\texttt{OVAL21}_{\texttt{DEEP}}$, with the increase of the percentage of weight pruning

	Ivan	Olive^{g}	Olive^{b}
Prune(0.1%)	1.70×	2.35×	2.95×
Prune(1%)	1.73×	2.13×	2.86×
Prune(3%)	1.69×	2.12×	2.71×
Prune(7%)	0.07×	0.07×	0.07×
Prune(12%)	0.02×	0.02×	0.02×

	Ivan	Olive^{g}	Olive^{b}
Prune(0.1%)	1.79 ×	2.53×	2.85×
Prune(1%)	1.78×	2.77×	3.40×
Prune(3%)	1.84×	2.68×	2.85×
Prune(7%)	0.13×	0.13×	0.13×
Prune(12%)	0.03×	0.03×	0.03×

- The different variants regarding UCB1 show consistent performance across different settings, which is consistent with the observation in Fig. 7. In particular, UCB1Balance outperforms Default for relatively simple problems, resulting in 55 instances solved in 0-100s.
- PWExplore1 shows remarkable efficiency for the most challenging problems by solving 52 instances in 600-1000s, while PWExplore2 excels in quicker solutions with 46 instances in 0-100s and still performs excellently for medium problems with 15 instances in 400-600s. This result indicates that the strategy that favors exploration is generally useful in progressive widening. In contrast, PWBalance performs well for quick solutions by achieving speedups over 10× for 46 instances in the 0-100s, and PWExploit solves 41 instances in 0-100s but only 26 instances in 200-400s. This stresses that different settings of progressive widening can significantly impact the efficiency of algorithm for problems of different complexities.

RQ5 *How is the performance of* Olive *subject to the similarity of N^* to N?*

In this RQ, we aim to assess the level of similarity between the original network N and the updated network N^*, within which our proposed approach Olive can still be effective. To clearly show the influence of similarity to the effectiveness of Olive, we select the problem instances w.r.t. the models $\texttt{OVAL21}_{\texttt{BASE}}$ and $\texttt{OVAL21}_{\texttt{DEEP}}$ from the dataset of CIFAR-10, because these models have more complex architectures and consist of a great number of neurons and weights, such that applying weight pruning to these models can lead to a more significant change of model behavior.

Our evaluation result is presented in Table 4. In both sub-tables of Table 4, the first column shows the parameter used in weight pruning, namely, the percentage of the weights that are pruned. The greater this value is, the less similarity between the produced network N^* and the original network N. The table shows the average speedup of the proposed approach Olive w.r.t. the BaB baseline approach, over the solved problem instances. We also present the results of the baseline incremental verification approach Ivan for a comparison.

By observing Table 4, we find that with the increase of dissimilarity between the two networks, it is evident that the performances of our approaches Olive^{g} and Olive^{b} decrease. For the problem instances of $\texttt{OVAL21}_{\texttt{BASE}}$, such decrease is monotonic, while for the problem instances of $\texttt{OVAL21}_{\texttt{DEEP}}$, there is also a same trend. Note that the Ivan baseline also demonstrates a similar phenomenon, showing that sufficient similarity between two networks is a necessary condition for incremental verification to be effective. Specifically, for the selected models $\texttt{OVAL21}_{\texttt{BASE}}$ and $\texttt{OVAL21}_{\texttt{DEEP}}$, both require a similarity such that the weight pruning percentage should be lower than 3%; otherwise, incremental verification approaches do not perform as well as BaB approach.

On the one hand, this observation confirms the usefulness of the template reuse strategy in our framework. Indeed if the two networks are not similar enough, the template cannot provide meaningful information to guide the verification of the new network N^* and so the performance of Olive can be severely diminished, as shown in Table 4. Moreover, the result also suggests that ensuring the similarity between two neural networks is a precondition for applying incremental

verification in practice. Determining the similarity between two networks may require domain expertise of practitioners, and the systematic approach for doing this is left as a future work.

6 Related Work

In this section, we discuss related works, from the aspects of classic neural network verification, testing approaches, incremental verification, and a related technique called differential verification.

Neural network verification: Neural network verification has been actively studied in the past years, as indicated by the survey papers [Liu et al. 2021] and the annual competition for comparing the performances of verification tools [Brix et al. 2023a,b]. To summarize, mainstream verification approaches can be roughly categorized to the following two classes:

- *Exact approaches* verify a neural network by performing an exact encoding or comprehensive exploration of state space. For instance, one such an approach is by encoding the neural network inference logic to be *mixed integer linear programming (MILP)* constraints [Cheng et al. 2017; Fischetti and Jo 2018; Tjeng et al. 2018]. Another line of these methods include Reluplex [Katz et al. 2017, 2019], which implements an SMT solver dedicated to neural network with ReLU activation functions. These approaches are often complete, but they may not be very efficient, and can severely suffer from the scalability issue.
- *Approximated approaches* [Singh et al. 2018, 2019; Weng et al. 2018; Zhang et al. 2018] pursue an over-approximation of the original reachable region of the network, to assess whether the approximated region violates the specification. These approaches often sacrifice completeness, but since the computation of over-approximation is much faster, they are superior in scalability. There are many works [De Palma et al. 2021a; Lan et al. 2022; Ma et al. 2024; Müller et al. 2022b; Wu and Zhang 2021; Zhang et al. 2022] that aim to improve the precision of over-approximation to achieve better completeness. There are also works [Elboher et al. 2020; Yang et al. 2021] that adopt CEGAR to refine the over-approximation to improve verification precision.

Recently, the BaB paradigm [Bunel et al. 2020; De Palma et al. 2021b; Isac et al. 2022; Kouvaros and Lomuscio 2021] has been widely adopted as a complement for the incomplete but fast approximated approaches. It decomposes the verification problem into sub-problems whenever an approximated verifier reports a spurious counterexample, and so it can guarantee both soundness and completeness of the verification approach. There are various approaches that aim to improve ReLU selection in BaB, e.g., DeepSplit [Henriksen and Lomuscio 2021], FSB [De Palma et al. 2021b]. As mentioned in §3.2, our aim is to change how BaB explores the tree, thus orthogonal to that line of works.

Although approximated approaches are efficient for neural network verification, their results can remain inconclusive once they cannot find a real counterexample. While verification can be terminated by finding a counterexample, in practice it is often not easy to achieve this. In [Guo et al. 2021], *Guo et al.* decomposes the problem and identifies the most suspicious sub-problems where counterexamples may exist, based on the affinity between different labels. Intuitively, the more affinitive two labels are, the more misleading they are to classification. This notion of affinity is then used in their technique as a guidance to eagerly falsify the specification. While the spirit in that work is similar to ours, we target a different context of verification, i.e., incremental verification, and our notion of guidance exactly benefits from the specific context we consider, namely, our guidance comes from the existing verification results for a similar neural network.

Testing approaches: Testing approaches refer to those techniques that are designed to search for counterexamples. There have been extensive research in this direction, including gradient-based attacks [Goodfellow et al. 2015; Madry et al. 2018], and search-based approaches [Eniser et al. 2019; Kim et al. 2019; Pei et al. 2017; Tian et al. 2018]. While our approach also pursues the detection of counterexamples, the most significant difference from those testing approaches is that our

approach is sound, i.e., when our approach does not detect a counterexample, it indicates that no counterexample exists. In contrast, testing or attack approaches cannot provide such guarantee. When testing or attack fails to find counterexamples, the counterexample may still possibly exist in the neural network, namely, those approaches are possible to raise false positives.

Incremental verification: Incremental verification emerges as a technique for verification acceleration, and recently it has been increasingly investigated. In [Fischer et al. 2022; Wei and Liu 2021], shared certificates are used as the proof templates to speed up verification process. In FANC [Ugare et al. 2022], the proof via linear relaxation for a network is incrementally reused as a template for accelerating the proof for a similar neural network. DeepInc [Yang et al. 2023] targets incremental verification by employing SMT-based constraint solving for incrementally searching for counterexamples. Ivan [Ugare et al. 2023], the baseline approach used in this paper, is proposed to reuse the tree generated by BaB during the verification of a similar neural network.

Differential verification of neural networks: A related line of work to incremental verification is *differential verification*, which aims to prove the accuracy of a compressed neural network w.r.t. to the original neural network. Representative works such as ReluDiff [Paulsen et al. 2020a] and Neurodiff [Paulsen et al. 2020b] rely on approximated methods, such as symbolic propagation [Wang et al. 2018a] to characterize the difference between two neural networks. Recently, SMT-based method is also adopted by [Eleftheriadis et al. 2022] to pursue a more precise equivalence relation between two neural networks.

7 Conclusion and Future Work

In this paper, we propose Olive, an incremental verification approach for neural networks, that aims to accelerate the verification of a neural network N^*, based on the existing verification results of a similar neural network N. Compared with the existing incremental verification approach, Olive leverages the information regarding the priority order over the sub-problems, in the sense that the sub-problems that are more likely to produce counterexamples should be prioritized in order to terminate the verification process as soon as a real counterexample is detected. Our experimental results, based on the comparison with two baseline approaches, show the performance advantages of Olive over the baseline approaches. In particular, Olive demonstrates excellent efficiency improvement on the verification problem instances that violate specifications, thanks to its ability of identifying critical sub-problems.

As future work, we plan to further exploit the acceleration of incremental verification, in order to impel the deployment of neural network verification in industrial practice. One possibility lies in refining the order of ReLU specification selection, as explored in [Ugare et al. 2023]. Our direction is to incorporate runtime information to derive a better ReLU specification heuristic to further improve the efficiency of neural network verification. Moreover, to apply incremental verification in practice, we plan to investigate the conditions about the similarity between networks, such that we can understand when incremental verification can indeed improve verification efficiency.

8 Data-Availability Statement

All relevant data that support the findings of this paper are available in the GitHub repository https://github.com/DeepLearningVerification/Olive.

9 Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This research is supported by JST BOOST Grant No. JPMJBY24D7, JSPS KAKENHI Grant No. JP23H03372, JST-Mirai Grant No. JPMJMI20B8, and Australian Research Council Grants No. FT220100391 and No. DP250101396.

References

Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T Johnson. 2023a. The Fourth International Verification of Neural Networks Competition (VNN-COMP 2023): Summary and Results. *arXiv preprint arXiv:2312.16760* (2023).

Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T Johnson, and Changliu Liu. 2023b. First three years of the international verification of neural networks competition (VNN-COMP). *International Journal on Software Tools for Technology Transfer* 25, 3 (2023), 329–339.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21, 42 (2020), 1–39.

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*. Springer, 251–268.

Alessandro De Palma, Harkirat S Behl, Rudy Bunel, Philip Torr, and M Pawan Kumar. 2021a. Scaling the convex barrier with active sets. In *ICLR 2021 Conf.* Open Review.

Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip HS Torr, and M Pawan Kumar. 2021b. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718* (2021).

Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. 2020. An abstraction-based framework for neural network verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer, 43–65.

Charis Eleftheriadis, Nikolaos Kekatos, Panagiotis Katsaros, and Stavros Tripakis. 2022. On neural network equivalence checking using SMT solvers. In *Int. Conf. on Formal Modeling and Analysis of Timed Systems*. Springer, 237–257.

Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. Deepfault: Fault localization for deep neural networks. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 171–191.

Claudio Ferrari, Mark Niklas Muller, Nikola Jovanovic, and Martin Vechev. 2022. Complete verification via multi-neuron relaxation guided branch-and-bound. *arXiv preprint arXiv:2205.00263* (2022).

Marc Fischer, Christian Sprecher, Dimitar Iliev Dimitrov, Gagandeep Singh, and Martin Vechev. 2022. Shared certificates for neural network verification. In *Int. Conf. on Computer Aided Verification*. Springer, 127–148.

Matteo Fischetti and Jason Jo. 2018. Deep neural networks and mixed integer linear optimization. *Constraints* 23, 3 (2018), 296–309.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symp. on Security and Privacy (SP)*. IEEE, 3–18.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd Int. Conf. on Learning Representations (ICLR'15)*. Int. Conf. on Learning Representations, ICLR, San Diego, CA, United States, 11 pages.

Google LLC . [n. d.]. *G Suite*. https://gsuite.google.com

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 *2021 IEEE 32nd Int. Symp. on Software Reliability Engineering (ISSRE)*. IEEE, 345–356.

Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

Patrick Henriksen and Alessio Lomuscio. 2021. DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis.. In *IJCAI*. 2549–2555.

Omri Isac, Clark Barrett, Min Zhang, and Guy Katz. 2022. Neural network verification with proof production. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*. 38–48.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer, 97–117.

Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. 2019. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer Int. Publishing, 443–452.

Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

Panagiotis Kouvaros and Alessio Lomuscio. 2021. Towards Scalable Complete Verification of Relu Neural Networks via Dependency-based Branching.. In *IJCAI*. 2643–2650.

Jianglin Lan, Yang Zheng, and Alessio Lomuscio. 2022. Tight neural network verification via semidefinite relaxations and linear reformulations. In *AAAI Conf. on Artificial Intelligence*, Vol. 36. 7272–7280.

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

Zhongkui Ma, Jiaying Li, and Guangdong Bai. 2024. ReLU Hull Approximation. *ACM on Programming Languages* 8, POPL (2024), 2260–2287.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th Int. Conf. on Learning Representations (ICLR'18)*. Vancouver, Canada, 27 pages.

Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T Johnson. 2022a. The third international verification of neural networks competition (VNN-COMP 2022): summary and results. *arXiv preprint arXiv:2212.10376* (2022).

Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2022b. PRIMA: general and precise neural network certification via scalable convex hull approximations. *ACM on Programming Languages* 6, POPL (2022), 1–33.

Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020a. Reludiff: Differential verification of deep neural networks. In *ACM/IEEE 42nd Int. Conf. on Software Engineering*. 714–726.

Brandon Paulsen, Jingbo Wang, Jiawei Wang, and Chao Wang. 2020b. Neurodiff: scalable differential verification of neural networks using fine-grained approximation. In *35th IEEE/ACM Int. Conf. on Automated Software Engineering*. 784–796.

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.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and Effective Robustness Certification. In *Advances in Neural Information Processing Systems*, Vol. 31. 12 pages.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *ACM on Programming Languages* 3, POPL (2019), 1–30.

Aleksandrs Slivkins et al. 2019. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning* 12, 1-2 (2019), 1–286.

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.

Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. 2018. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *Int. Conf. on Learning Representations*.

Shubham Ugare, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. 2023. Incremental Verification of Neural Networks. *ACM on Programming Languages* 7, PLDI (2023), 1920–1945.

Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. 2022. Proof transfer for fast certification of multiple approximate neural networks. *ACM on Programming Languages* 6, OOPSLA1 (2022), 1–29.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018a. Efficient formal safety analysis of neural networks. *Advances in neural information processing systems* 31 (2018).

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018b. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symp. (USENIX Security 18)*. 1599–1614.

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. *Advances in Neural Information Processing Systems* 34 (2021), 29909–29921.

Tianhao Wei and Changliu Liu. 2021. Online Verification of Deep Neural Networks under Domain or Weight Shift. *arXiv preprint arXiv:2106.12732* (2021).

Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. 2018. Towards fast computation of certified robustness for relu networks. In *Int. Conf. on Machine Learning*. PMLR, 5276–5285.

Eric Wong and Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Int. Conf. on Machine Learning*. PMLR, 5286–5295.

Yiting Wu and Min Zhang. 2021. Tightening robustness verification of convolutional neural networks with fine-grained linear approximation. In *AAAI Conf. on Artificial Intelligence*, Vol. 35. 11674–11681.

Xiaoyong Xue and Meng Sun. 2023. Branch and Bound for Sigmoid-Like Neural Network Verification. In *Int. Conf. on Formal Engineering Methods*. Springer, 137–155.

Pengfei Yang, Zhiming Chi, Zongxin Liu, Mengyu Zhao, Cheng-Chao Huang, Shaowei Cai, and Lijun Zhang. 2023. Incremental Satisfiability Modulo Theory for Verification of Deep Neural Networks. *arXiv preprint arXiv:2302.06455* (2023).

Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. 2021. Improving neural network verification through spurious region guided refinement. In *Int. Conf. on Tools and Algorithms for the*

Construction and Analysis of Systems. Springer, 389–408.

Guanqin Zhang, Zhenya Zhang, Dilum Bandara, Shiping Chen, Jianjun Zhao, and Yulei Sui. 2025. Supplementary material for the paper "Efficient Incremental Verification of Neural Networks Guided by Counterexample Potentiality". https://sites.google.com/view/olive-nnv

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems* 31 (2018).

Zhaodi Zhang, Yiting Wu, Si Liu, Jing Liu, and Min Zhang. 2022. Provably tightest linear approximation for robustness verification of sigmoid-like neural networks. In *IEEE/ACM Int. Conf. on Automated Software Engineering*. 1–13.

Received 2024-10-16; accepted 2025-02-18