Lookahead Branching for Neural Network Verification

Liam Davis[1] , **Duo Zhou**[2] , **Huan Zhang**[2] , **Guy Katz**[3] , **Clark Barrett**[4] , **Haoze Wu**[1]
[1]Amherst College
[2]University of Illinois Urbana-Champaign
[3]Hebrew University of Jerusalem
[4]Stanford University
ljdavis27@amherst.edu, duozhou2@illinois.edu, huan@huan-zhang.com, g.katz@mail.huji.ac.il, barrettc@stanford.edu, hwu@amherst.edu

Abstract

In this work, we investigate the effect of lookahead branching strategies in neural network verification. We present a general recipe to integrate lookahead into any branch-and-bound verifier and demonstrate how one of the current state-of-the-art branching heuristics, FSB, can be viewed as a special instantiation of the lookahead branching strategy. We also describe how, in addition to improving the quality of branching decisions, lookahead can generate additional lemmas that accelerate verification. We instantiate the method in two representative branch-and-bound-based verifiers (Marabou and α-β-CROWN), and demonstrate that lookahead leads to consistent speedups in verification time and up to 57% more solved instances. Code is available at https://github.com/ai-ar-research/lookahead-branching.

1 Introduction

Deep neural networks (DNNs) have become state-of-the-art solutions in various domains [Sallab *et al.*, 2017; Mnih *et al.*, 2013; He *et al.*, 2015]. To ensure the deployment of DNN-based systems in safety critical domains, there has been significant effort from the formal methods and machine learning communities on developing scalable formal verification techniques that can reason about the behaviors of a neural network [Katz *et al.*, 2017; Singh *et al.*, 2019; Zhang *et al.*, 2018; Wang *et al.*, 2021]. State-of-the-art complete neural network verifiers are based on branch-and-bound (BaB), which involves performing case splitting on non-linear activation functions (e.g., ReLU) and analyzing the cases using an incomplete verifier; if the analysis is inconclusive, further case splits are recursively performed.

The branching heuristic, the strategy to choose which case to split on, has a significant impact on verification efficiency. Ideally, the branching heuristic should lead to easier subproblems. Failing to do so, especially at the top of the search tree, can result in duplicated verification efforts and increased verification time. A number of branching heuristics have been developed for neural network verification [Katz *et al.*, 2017; Wu *et al.*, 2020; Bunel *et al.*, 2020; De Palma *et al.*, 2021; Wu *et al.*, 2022]. Most heuristics aim to make a decision *quickly* by leveraging local information (e.g., variable bounds) gathered during the solving process. This increases the risk of ineffective branching, as decisions are made without evaluating the long-term impact of splits. Recent heuristics have shown it is beneficial to spend more effort making a branching decision by simulating branching on candidate neurons and evaluating its effect [De Palma *et al.*, 2021]. This approach has culminated in Filtered Smart Branching (FSB), the standard branching heuristic in recent work.

Motivated by the success of FSB, we systematically study a general branching strategy that spends *significant* effort making branching decisions. We adopt the terminology in formal methods and call this approach *lookahead* [Heule and van Maaren, 2009]. Lookahead involves simulating potential branching decisions by performing multiple splits to measure downstream effects. By analyzing the impact of each simulated branch, lookahead uses more information to make branching decisions. We present lookahead as a template algorithm with several tunable parameters such as the preselect strategy, lookahead depth, and evaluation metric that can be instantiated in a solver-specific manner. We discuss the choices for these parameters and how FSB can be viewed as a special instantiation of the lookahead procedure. In addition to informing the branching decision, lookahead might also discover new information, such as tightened variable bounds. We show this information can be used to derive valid lemmas and potentially prune the search space. We instantiate lookahead in two distinct BaB-verifiers, Marabou [Katz *et al.*, 2019; Wu *et al.*, 2024] and α-β-CROWN [Wang *et al.*, 2021; Xu *et al.*, 2020; Zhang *et al.*, 2022a; Zhou *et al.*, 2024], and demonstrate that lookahead can improve both tools.

An overview of the lookahead workflow is presented in Figure 1. At a given search state when branching is required, a set of split candidates is pre-selected and lookahead simulates branching on each candidate up to a certain depth. The outcome of each simulation is a score along with a number of tightened bounds, which might entail that certain unstable neurons are fixed to particular activation phases. In the end, lookahead outputs the next neuron to split on and a set of lemmas discovered during the lookahead simulations.

The rest of the paper is organized as follows: Section 2 reviews related work. Section 3 provides background on neural network verification. Section 4 presents the general looka-

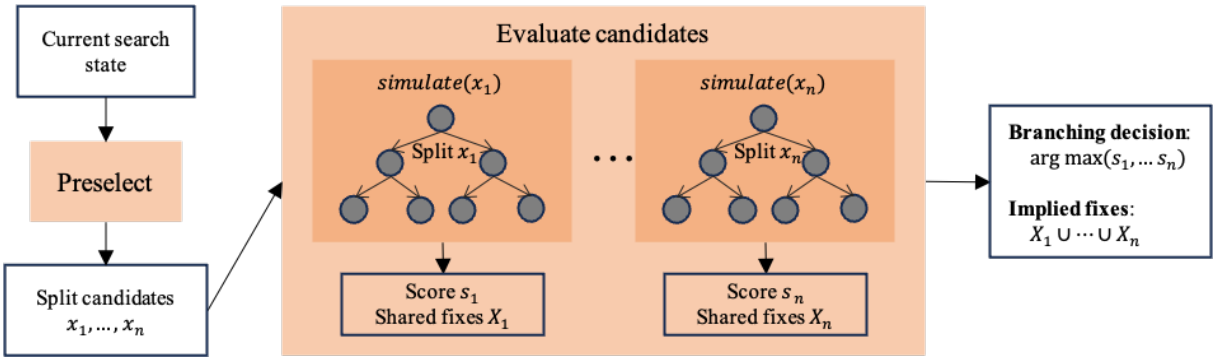


Figure 1: Visual overview of the lookahead procedure

head approach and concrete instantiations in Marabou and α-β-CROWN. Section 5 provides an evaluation of lookahead branching, comparing it to existing heuristics in Marabou and α-β-CROWN. Finally, we conclude and discuss future directions in Section 6.

2 Related Work

Early efforts for complete verification of neural networks employed SMT and MILP solvers that enumerate activation patterns [Katz *et al.*, 2017]. A unified BaB view of verification was presented by [Bunel *et al.*, 2020]. State-of-the-art complete verification tools including the SMT-based solvers [Katz *et al.*, 2019; Wu *et al.*, 2024] and GPU-accelerated bound-propagation-based tools [Wang *et al.*, 2021; Xu *et al.*, 2020; Zhang *et al.*, 2022a; Zhou *et al.*, 2024; Shi *et al.*, 2025] can be viewed as instantiations of BaB. Branching heuristics have a significant impact on the runtime of complete verification tools. BABSR introduced a "strong-branching" style score that solves a cheap surrogate relaxation for each candidate and became the default in many verifiers [Bunel *et al.*, 2020]. FSB (Filtered Smart Branching) refines this idea by first filtering candidates with BaBSR and then re-running a tighter bound computation on the shortlist [De Palma *et al.*, 2021]. There has also been work on using Graph Neural Networks to train a policy for selecting splitting neurons [Lu and Kumar, 2019], but such approaches require offline training and are typically tied to the distribution of training data, making them less directly comparable to general-purpose heuristics like ours. Our framework is complementary with learning-based heuristics, which could in principle be used as a preselect strategy within the lookahead template. Our work is inspired by *lookahead* search in SAT and SMT solvers [Heule and van Maaren, 2009] as well as MILP solvers [Glankwamdee and Linderoth, 2006], and we extend similar ideas to neural network verification.

3 Background

3.1 Neural Network Verification

For a trained deep neural network $N : \mathbb{R}^n \to \mathbb{R}^m$ with an input $x \in \mathbb{R}^n$ and output $y = N(x) \in \mathbb{R}^m$, the general DNN verification problem is whether or not there exists an input x that produces an output y that satisfies a property $\phi(y)$. If there exists an input x that leads to an output y that satisfies the property $\phi(y)$, then the problem is satisfiable (SAT); otherwise it is unsatisfiable (UNSAT).

3.2 Bound Propagation

To refine the search space of a neural network verification problem, bound propagation [Singh *et al.*, 2019] estimates activation ranges at each layer, defining upper and lower bounds for each neuron. The Rectified Linear Unit (ReLU) activation function is defined as $\text{ReLU}(x) = \max(0, x)$. A ReLU neuron is considered active if its lower bound is strictly greater than zero ($\underline{z}^{(l)} > 0$), meaning its output will always be its input. Conversely, a ReLU neuron is inactive if its upper bound is less than or equal to zero ($\overline{z}^{(l)} \leq 0$), meaning its output will always be zero. Otherwise, the activation status is not yet known and the ReLU in unstable. If a ReLU's bounds can guarantee the neuron is always active or inactive, unnecessary computations can be eliminated. Through this iterative process, the bounds on neuron activations are progressively tightened, leading the solver closer to a solution.

3.3 Branch-and-bound

The branch-and-bound (BaB) framework, illustrated by Figure 2, is an efficient approach to neural network verification. BaB systematically tightens the bounds of a neural network by splitting on unstable ReLU neurons. When an unstable ReLU is split, the problem is turned into two problems, one where the ReLU is active and one where the ReLU is inactive. To verify with BaB, ReLU splits are repeatedly applied until each subproblem can be definitively classified as either satisfying the property (SAT) or not satisfying it (UNSAT).

The choice of ReLU to split on is imperative for the efficiency of BaB, as it determines how fast a solver converges to a solution; well-selected splits make significantly more progress to a solution than poorly-chosen splits. Thus, branching heuristics are essential to efficient verification.

3.4 Branching Heuristics

Several branching heuristics have been developed for BaB-based neural network verification. BABSR is a deterministic heuristic that computes a surrogate relaxation of each neuron with the upper and lower bounds, and the biases of

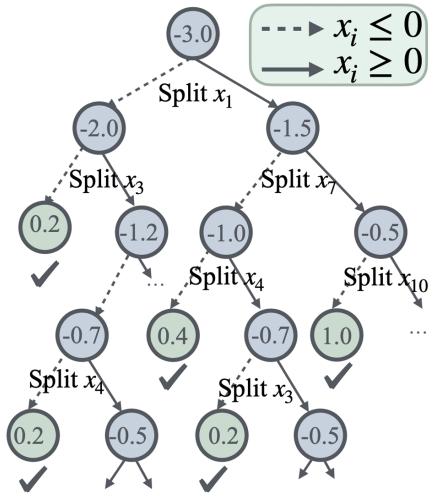


Figure 2: Each node encodes a BaB subproblem by splitting unstable ReLUs. Green nodes represent pruned nodes. [Zhou *et al.*, 2024].

previous layers [2020]. The unstable neuron with the highest score is then branched on. PSEUDO-IMPACT BRANCHING is a branching heuristic specific to Marabou. It estimates the impact a ReLU has on the Sum-of-Infeasibilities (SoI) of the network, calculated during Marabou's DeepSoI procedure [Wu *et al.*, 2022]. The ReLU with the greatest estimated impact on the SoI is branched on. Pseudo-impact is presently the state-of-the-art heuristic implemented in Marabou. FSB pre-selects candidate ReLUs with BaBSR scores, and simulates one level of branching on each candidate before making a branching decision, and can be seen as a special instantiation of lookahead branching. FSB is presently the state-of-the-art heuristic implemented in α-β-CROWN. We will discuss how lookahead extends FSB in section 4.

4 Methodology

In this section, we first present lookahead as a template algorithm and discuss its properties. We then discuss concrete strategies for instantiating the lookahead procedure. The algorithm begins by identifying a set of unstable neurons to simulate splits on. Given that performing a lookahead simulation on every unstable ReLU is computationally expensive, a preselect strategy is employed to shrink the candidate set. The preselect strategy differs based on implementation as detailed in Section 4.1.

For each preselected candidate, the SIMULATESPLIT subroutine simulates a case split on the neuron, creating one branch for each activation phase. For the ReLU activation function, there would be two phases (active and inactive). The algorithm then recursively explores the next level of the search tree by selecting another neuron to split on, using the BASESELECT heuristic. This process continues until a specified lookahead depth ($\mathbf{d}$) is reached. The goal of performing additional splits is to explore the cascading effects of each candidate split. After each split is simulated, the solver performs bound propagation. When the lookahead depth is reached, the EVALUATESTATE function is called to evaluate the current solver state and collect the current variable bounds. Importantly, the solver state is saved and restored after each simulated split to ensure that the lookahead simulation does not interfere with the actual search process. In the end, the SIMULATESPLIT function returns the score of each simulated leaf, as well as the loosest lower and upper bounds of the variables discovered during the simulation.

The scores from all simulated branches are aggregated using the specified aggregation strategy to produce a single score for each candidate neuron. The neuron with the best score is selected as the next split. Moreover, the tightest lower and upper bounds discovered during lookahead are returned, which can be used to further prune the search space.

In general, lookahead can be computationally expensive, as it requires repeated simulation of the solving process. In practice, it is beneficial to invoke Algorithm 1 at the top of the search tree, before falling back to more efficient heuristics. The key advantage of lookahead is that it considers the cascading effects of a branching decision. A neuron that appears promising based on local information may lead to poor downstream progress, while a seemingly less promising candidate can yield improvements over several branching decisions.

Algorithm 1 Lookahead Branching

Input: Set of unstable neurons $\mathcal{N}$ at the current search level
Output: $\langle n^*, \mathcal{L}, \mathcal{U} \rangle$ where n^* is the neuron to split, $\mathcal{L}$ and $\mathcal{U}$ are sound lower and upper bounds
Parameters: lookahead depth k, preselect strategy P, heuristic B, scoring function E, aggregate strategy C
$\mathcal{N}' \leftarrow P(\mathcal{N})$; $agScores \leftarrow \{\}$; $\mathcal{L}, \mathcal{U} \leftarrow [\,], [\,]$
for each $n \in \mathcal{N}'$ **do**
  $scores', \ell, u \leftarrow$ CaseSplit$(n, k, \mathcal{N} \setminus \{n\})$
  $agScores[n] \leftarrow C(scores')$; $\mathcal{L}$.append(ℓ); $\mathcal{U}$.append(u)
end for
return $\arg\max(agScores)$,
    elementwise-max$(\mathcal{L})$, elementwise-min$(\mathcal{U})$
Procedure CaseSplit$(n, d, \mathcal{N})$
if $d = 0$ **then**
  $score, \ell, u \leftarrow E()$; **return** $[score], \ell, u$
else
  storeSolverState(); $scores, L, U \leftarrow [\,], [\,], [\,]$
  for each $p \in$ phases(n) **do**
    applySplit(p); propagateBounds()
    if $d = 1$ **then**
      $n' \leftarrow B(\mathcal{N})$
    else
      $n' \leftarrow nil$
    end if
    $scores', \ell, u \leftarrow$ CaseSplit$(n', d-1, \mathcal{N} \setminus \{n'\})$
    $scores \leftarrow scores :: scores'$; L.append(ℓ); U.append(u); restoreSolverState()
  end for
  return $scores$,
      elementwise-min(L), elementwise-max(U)
end if

FSB can be viewed as a particular implementation of the lookahead procedure where BaBSR is used as the BASESELECT heuristic, $\mathbf{d}$ is 1, and the strategy is applied uniformly at every branching decision. This design prioritizes efficiency by minimizing per-decision overhead. But by simulating only one branching decision, FSB cannot fully capture the cascading effects that propagate through multiple levels of the search tree. Our work explores whether investing additional computation in critical branching decisions can improve verification performance. Specifically, we investigate depth-2 lookahead applied at the top of the search tree, where branching decisions have the largest downstream impact. This design is motivated by the strong branching literature in MILP [Glankwamdee and Linderoth, 2006], which demonstrates that expensive branching heuristics provide the greatest benefit early in search. Additionally, we explore how phase-fixing lemmas discovered during lookahead (Section 4.3) can further prune the search space. Section 5 evaluates whether these design choices yield net performance improvements despite their computational overhead.

4.1 Preselect Strategies

If lookahead were to be performed on every neuron in a large neural network, the computational overhead would outweigh the improvement in branch quality. Thus, it is essential to preselect a subset of neurons on which to run lookahead. Several preselect strategies are possible.

One approach is to use a polarity-based strategy, that uses a heuristic that scores neurons based on the sensitivity of their activation status to changes in their bounds. For example, a score such as $\max\left(\frac{ub}{lb}, \frac{lb}{ub}\right)$, where ub and lb are the upper and lower bounds of a neuron, can be used to identify neurons whose activation status is most undecided. Neurons with the highest scores are then selected for lookahead. We used this polarity-based preselect strategy in Marabou. Another approach is to use existing branching scores, such as BaBSR scores, to rank neurons. In this case, a fixed number of neurons with the highest scores are chosen for lookahead. This preselect strategy was used in α-β-CROWN. Regardless of the strategy, the preselect method should identify promising candidates using lightweight metrics.

4.2 Scoring Functions

The scoring function is a central component of lookahead branching, as it quantifies the effectiveness of each simulated split and guides the branching decision. Two general classes of scoring metrics are commonly used: neuron-fixing-based metrics and bound-reduction-based metrics.

Neuron-fixing-based metric. This approach evaluates a split by counting how many previously unstable neurons become phase-fixed (i.e., their activation status is determined) after bound propagation in each branch. To encourage both high total progress and balanced outcomes, a balance score is computed for each split. For example, if a split results in a neurons fixed in one branch and b in the other, the score can be defined as $\frac{a\times b}{a+b+1}$. This formula favors splits that both fix many neurons and distribute the fixes evenly between branches. Consider a split that fixes 9 neurons in one branch and 1 in the other. Its score would be $\frac{9\times 1}{9+1+1} \approx 0.8$. In contrast, a split that fixes 5 neurons in each branch would yield a score of $\frac{5\times 5}{5+5+1} \approx 2.3$. Our metric would then favor the more balanced outcome in case the same number of neurons are fixed. When lookahead is performed to greater depths, the scores are computed recursively: at the leaves, the score is based on the number of phase fixes, and at each internal node, the balance formula is applied to the scores of its child branches, propagating upward to yield a final score. This polarity-based metric was used in Marabou.

Bound-reduction-based metric. This metric is based on the reduction in variable bounds or other continuous measures of progress, such as the width of neuron bounds or their proximity to a decision threshold. For example, a scoring function may combine the width of a neuron's bounds, its distance from zero, and an estimate of its influence on the objective (e.g., via gradient approximation). In a lookahead setting, after simulating a split, the scoring function can aggregate the scores from subsequent branches using a balance formula similar to the neuron-fixing metric. For instance, if S^+ and S^- are the scores from the two branches after a split, the lookahead score can be defined as $S_0 + \lambda \cdot \frac{S^+\times S^-}{S^+ + S^- + 1}$, where S_0 is the immediate score for the split and λ is a discount factor to control the influence of deeper lookahead. For deeper lookahead, this aggregation is applied recursively as scores are propagated up the lookahead tree. This bound-reduction-based metric was used in α-β-CROWN.

Regardless of the specific metric, an effective scoring function should capture both the potential for maximal progress toward a solution and the balance of outcomes across different branches. This ensures that the branching decision not only accelerates convergence but also avoids highly imbalanced splits that may lead to inefficient search.

4.3 Phase Fixing via Lookahead Splits

One significant outcome of the lookahead branching procedure is its ability to refine variable bounds, which can lead to phase fixing of previously unstable neurons. Specifically, during the lookahead process, bound propagation is applied after simulating splits, and the resulting bounds can sometimes determine that a neuron is stable (i.e., its phase is fixed). The following theorem formalizes this effect:

Theorem 1 (Phase Fixing via Lookahead)**.** *Let $z = \mathrm{ReLU}(y)$ be an unstable ReLU, i.e., $\ell_y < 0 < u_y$. Suppose we simulate a split on another unstable ReLU $z_r = \mathrm{ReLU}(y_r)$, generating two subproblems corresponding to the inactive ($y_r \le 0$) and active ($y_r \ge 0$) cases.*

Let ℓ_y^{inact} and ℓ_y^{act} be the lower bounds on y obtained in each subproblem. Then the refined lower bound $\ell_y^{\mathrm{new}} := \min(\ell_y^{\mathrm{inact}}, \ell_y^{\mathrm{act}})$ satisfies $\ell_y^{\mathrm{new}} \ge \ell_y$. Similarly, the refined upper bound $u_y^{\mathrm{new}} := \max(u_y^{\mathrm{inact}}, u_y^{\mathrm{act}})$ satisfies $u_y^{\mathrm{new}} \le u_y$.

As a consequence: if $\ell_y^{\mathrm{new}} \ge 0$, the phase of z is fixed to active*; if $u_y^{\mathrm{new}} \le 0$, it is fixed to* inactive.

Proof Sketch. Simulating a split on z_r refines the input domain into two disjoint subdomains. Bound propagation on each subproblem yields tighter constraints on all dependent variables, including y. Because the union of the subdomains

recovers the full feasible region, the maximum lower bound and minimum upper bound across the subproblems provide valid global bounds. □

The phase-fixing capability cannot be used when either CROWN or α-CROWN bound propagation is used, as the adaptive ReLU rule means bounds can become looser when pre-activation bounds are tightened. Therefore we used phase-fixing in Marabou but not α-β-CROWN.

5 Evaluation

To evaluate the effectiveness of lookahead branching, we implemented Algorithm 1 in two state-of-the-art BaB-based verification tools Marabou and α-β-CROWN. The two verifiers employ different paradigms: Marabou is a CPU-based verifier that employs an SMT-based incomplete decision procedure, while α-β-CROWN runs GPU-accelerated bound-propagation. We evaluate lookahead branching against the state-of-the-art heuristics in each verifier on various benchmarks, investigating whether lookahead branching can improve the branch-and-bound across distinct solver paradigms.

5.1 Case Study on Marabou

Experimental Setup

For experimentation on Marabou, we compared lookahead branching to BaBSR and pseudo-impact, two of Marabou's existing heuristics. BaBSR is a widely-used deterministic heuristic [Bunel *et al.*, 2020], while pseudo-impact is a dynamic heuristic specific to Marabou [Wu *et al.*, 2022]. We conducted experiments on three different benchmark sets, NAP, NN4Sys, and MNIST. NAP is a benchmark designed to evaluate neural network verifiers on specifications defined by neural activation patterns, which characterize broad, numerically challenging regions of the input space. NN4Sys is a benchmark suite constructed from neural networks used in computer systems, testing real-world verification challenges that have been proven difficult in recent iterations of the VNN Competition. The MNIST dataset includes various feed forward neural networks for handwritten digit recognition. The MNIST network architectures we tested include, in layers by neurons, 20x20, 2x256, 4x256, and 6x256. We implemented lookahead on top of the most recent version of Marabou.

We use the polarity-based pre-selecting strategy as described in Section 4.1 to select 10 candidate neurons. We use a $\mathbf{d}$ of 2, and instantiate BASESELECT with two different heuristics, BaBSR and pseudo-impact. In the experiments, we perform lookahead splits on the top five branching levels, and fall back to the base heuristics for the rest of it. We used the neuron-fixed-based metric for evaluating a search state and combining the scores across subproblems 4.2. The experiments were run on a server with 2.6-GHz AMD CPUs, with 4 CPU cores allocated per benchmark. Each benchmark was given a 30 minute time limit and a 32 GB memory limit.

Results

Table 1 presents a comparison of the performance of Marabou with and without lookahead branching across the various benchmarks. The tables report the number of instances solved within the time limit (1800 seconds), along with average run time on solved instances. Overall, lookahead branching results in a substantial increase in the number of solved instances for both BaBSR and pseudo-impact heuristics.

Benchmark	#	babsr		babsr+lh		p-i		p-i+lh	
	Bench.	Sol.	Time	Sol.	Time	Sol.	Time	Sol.	Time
NAP	235	30	29.0	**47**	3.7	19	0.5	**24**	3.0
NN4SYS	120	**80**	127.1	**80**	114.9	82	149.9	**94**	238.1
MNIST 20x20	500	183	100.5	**192**	130.6	130	14.9	**141**	9.3
MNIST 2x256	500	**370**	28.3	369	30.3	405	77.9	**407**	83.1
MNIST 4x256	500	281	30.1	**286**	30.5	**298**	72.5	295	52.2
MNIST 6x256	500	248	86.1	**254**	94.0	240	42.5	**240**	40.9

Table 1: Marabou results on various benchmarks. P-I denotes pseudo-impact, LH denotes lookahead. Time is in seconds.

Figure 3 provides cactus plots comparing lookahead against BaBSR and pseudo-impact on the three benchmark sets. Overall, lookahead leads to more solved instances compared to the baseline heuristics, especially at higher time limits. On NAP, overhead makes lookahead slower at low time limits, but branching quality gains dominate overall. On NN4Sys, lookahead yields significantly more solved instances with pseudo-impact and solve time improvements with BaBSR. For MNIST, lookahead solves more instances at higher time limits. Table 2 compares lookahead's overhead an easy and hard instance. On the easy instance, the overhead dominates. On the hard instance, the overhead constitutes a small part of overall runtime and the improved branching yields a substantial speedup.

Instance	Baseline	LH	LH Phase (s)	Overhead
NAP CLS0_ID41	0.22	2.48	2.17	87.5%
NN4SYS LINDEX_1_5	537.62	66.14	7.01	10.6%

Table 2: Lookahead overhead on easy vs. hard instances, time in seconds.

Ablation Studies on Marabou

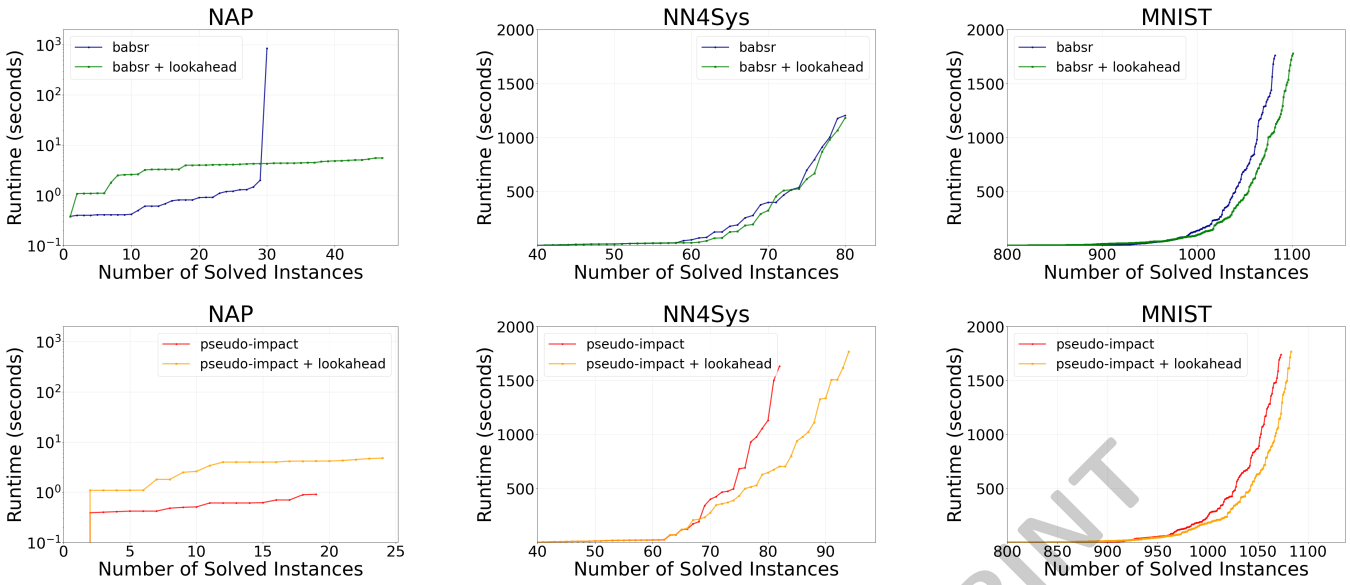
We performed additional ablation studies on the NAP and NN4Sys benchmarks by varying the lookahead depth (D), number of preselected candidates (C), and removing phase fixing (PF). All configurations use the polarity-based preselect strategy and BaBSR as the base select heuristic.

On NAP, all lookahead configurations solve between 36 and 47 instances, compared to 30 for BaBSR. On NN4Sys, varying depth and candidate count has minimal effect on performance, with the exception of the D10C10 configuration, where increased branching overhead reduces performance. Comparing lookahead with and without phase fixing shows that phase fixing has no effect on NAP but yields a 33-second improvement on NN4Sys.

5.2 Case Study on α-β-CROWN

Experimental Setup

For experimentation on α-β-CROWN, we compared lookahead branching to its current state-of-the-art heuristic,



(a) NAP (b) NN4Sys (c) MNIST

Figure 3: Cactus plots comparing existing heuristics and lookahead on Marabou.

	NAP		NN4Sys	
Configuration	Solved	Time (s)	Solved	Time (s)
LH D5C10	**47**	3.7	80	114.9
LH D1C10	36	1.5	80	118.8
LH D4C10	36	1.6	80	121.97
LH D6C10	36	1.59	80	124.73
LH D8C10	36	1.48	80	121.93
LH D10C10	45	5.1	79	120.1
LH D5C5	**47**	2.1	80	118.5
LH D5C20	39	6.6	80	121.9
LH D5C10 no PF	**47**	3.3	80	147.6

Table 3: Ablation studies on Marabou with BaBSR. D denotes lookahead depth, C denotes preselected candidates, PF denotes phase fixing.

		T/O	Solved		Avg Time (s)	
Dataset	**Model**	**#**	**(s)**	FSB	LH	FSB	LH
CIFAR	CIFAR100	200	360	112	112	19.49	**16.47**
	CIFAR10-ResNet	72	360	59	**60**	22.83	**21.87**
	CNN-A-Mix	200	200	83	83	7.58	**6.03**
	CNN-B-Adv	200	450	93	93	10.23	**8.60**
MNIST	CNN-A-Adv	200	200	141	141	11.34	**8.90**
TinyImageNet	tinyimagenet	200	360	129	**130**	23.84	**20.83**

Table 4: α-β-CROWN results on various benchmarks. T/O denotes timeout, LH denotes lookahead.

FSB [De Palma *et al.*, 2021], on seven different convolutional neural networks ranging with various robustness properties for each. The networks came from the CIFAR, MNIST, and TinyImageNet datasets, three computer vision datasets for image and text recognition. The networks from the CIFAR and TinyImageNet datasets range between 14.4 million and 31.6 million parameters, making them challenging verification queries. The MNIST benchmarks were adversarially trained, introducing complexity that makes formal verification particularly challenging. We implemented lookahead in the most recent version of α-β-CROWN [Zhou *et al.*, 2025].

With α-β-CROWN, the solver first attempted to solve the problem with an adversarial attack algorithm [Zhang *et al.*, 2022b], then with incomplete verification using auto-LiRPA [Xu *et al.*, 2021] before beginning complete verification with BaB. When BaB is run, the first five BaB rounds are done using lookahead branching with a $\mathbf{d}$ of 2, and then FSB branching is used. We use BaBSR as the preselect strategy and select 15 candidate neurons. We used the bound-reduction-based metric for the scoring function (Section 4.2) with a discount factor λ of 0.5. We did not implement the phase fixing techniques in α-β-CROWN, as α-β-CROWN leverages α-CROWN for bound propagation and thus phase fixing is not sound. The experiments were run on a server with 2.6-GHz AMD CPUs and A100 GPUs. One A100 GPU and 96 CPU cores were allocated for each experiment, and a 128 GB memory limit was given for each experiment.

Results

Table 4 presents a comprehensive overview of the verification performance on the seven neural network models using both FSB branching and lookahead branching within α-β-CROWN. The table compares the total number of solved instances (including both SAT and UNSAT) and the average runtime on solved instances. To isolate the impact of lookahead branching on the branch-and-bound core of α-β-

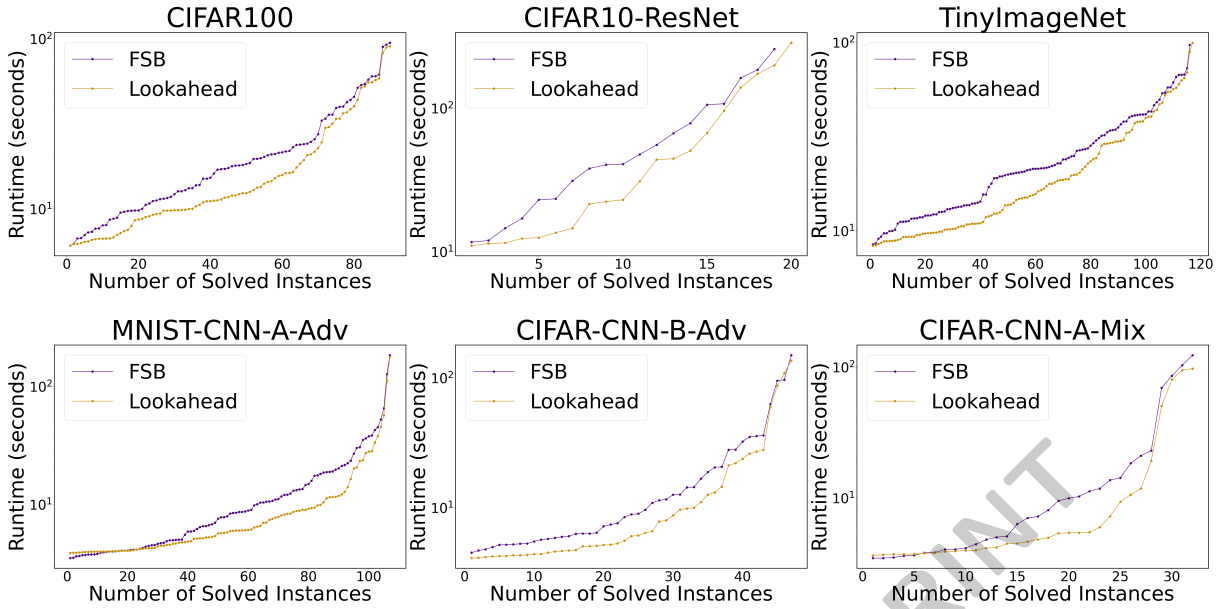


Figure 4: Cactus plots comparing FSB and lookahead on α-β-CROWN.

Dataset	Model	Solved FSB	Solved LH	Avg Time (s) FSB	Avg Time (s) LH	Avg Speedup (%)
CIFAR	CIFAR100	90	90	23.34	**19.58**	16.1
	CIFAR10-ResNet	19	**20**	69.51	**64.27**	24.3
	CNN-A-Mix	32	32	19.10	**15.08**	15.4
	CNN-B-Adv	47	47	19.77	**16.54**	21.9
MNIST	CNN-A-Adv	107	107	14.77	**11.53**	16.3
TinyImageNet	tinyimagenet	116	**117**	26.11	**22.74**	21.0

Table 5: α-β-CROWN results on UNSAT instances solved with BaB. LH denotes lookahead, timeouts remain the same as table 4.

CROWN, Table 5 presents results filtered to include only instances that required the full BaB procedure. SAT instances solved through adversarial attacks and easier UNSAT instances solved with incomplete verification are excluded. We see that lookahead leads to a consistent speedup in solve time, and contributes two unique solutions.

Ablation Studies on α-β-CROWN

Table 6 presents the performance of α-β-CROWN while varying the number of lookahead branches performed before switching to FSB. Across all hyperparameters tested, lookahead either solves more instances or has faster solve times than FSB. Changing the number of lookahead branches does not change the number of solved instances and changes the solve time minimally.

Overall, we found that lookahead can boost the performance of BaB in two fundamentally different neural network verifiers. This suggests that lookahead branching can be a generally applicable strategy for enhancing the performance of complete neural network verification.

Model	1 LH Branch Solved	1 LH Branch Time	5 LH Branches Solved	5 LH Branches Time	10 LH Branches Solved	10 LH Branches Time
CIFAR100	90	64.21	90	64.27	90	**57.25**
CIFAR10-ResNet	20	20.40	20	**19.58**	20	21.49
CNN-A-Mix	32	16.15	32	15.08	32	**14.99**
CNN-B-Adv	47	17.24	47	**16.54**	47	16.86
CNN-A-Adv	107	11.89	107	**11.53**	107	13.65
tinyimagenet	117	24.66	117	**22.74**	117	24.93

Table 6: α-β-CROWN results with varying number of lookahead branches. LH denotes lookahead. Time is in seconds.

6 Conclusion

In this paper, we introduced a general lookahead branching strategy for neural network verification. The key idea is to simulate candidate splits to make more informed branching decisions. We discussed design choices of lookahead and instantiated lookahead branching for two distinct complete verifiers, Marabou and α-β-CROWN. Our results show that using lookahead branching results in substantial performance gains in both solvers across a wide range of benchmarks.

Limitations. As we presented lookahead as a template algorithm, its design space is massive. While we considered two instantiations in Marabou and one in α-β-CROWN, it would be interesting to evaluate more variants of lookahead. In particular, we plan to investigate deeper lookahead simulations or periodically invoking lookahead later in the search. In addition, while we explored leveraging lookahead to fix unstable neurons, it might be interesting to leverage other information, such as dependencies between neurons, which might result in additional pruning of the search space.

Acknowledgments

This work made use of high-performance computing equipment funded by the National Science Foundation under Grant #2117377. Wu's work is supported by a gift from VMware University Research Fund. Additional support was provided by the Stanford Center for Automated Reasoning (Centaur). Huan Zhang acknowledges the support from the Schmidt Sciences AI2050 program and the National Science Foundation under Grant #2331967.

References

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

[De Palma *et al.*, 2021] Alessandro De Palma, Harkirat S Behl, Rudy Bunel, Philip Torr, and M Pawan Kumar. Scaling the convex barrier with active sets. In *Proceedings of the ICLR 2021 Conference*. Open Review, 2021.

[Glankwamdee and Linderoth, 2006] Wasu Glankwamdee and Jeff Linderoth. Lookahead branching for mixed integer programming. Technical Report 06T-004, Lehigh University, Department of Industrial and Systems Engineering, October 2006.

[He *et al.*, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[Heule and van Maaren, 2009] Marijn J.H. Heule and Hans van Maaren. Look-ahead based sat solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 5, pages 155–184. IOS Press, 2009.

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

[Katz *et al.*, 2019] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 443–452, Cham, 2019. Springer International Publishing.

[Lu and Kumar, 2019] Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. *arXiv preprint arXiv:1912.01329*, 2019.

[Mnih *et al.*, 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Sallab *et al.*, 2017] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *arXiv preprint arXiv:1704.02532*, 2017.

[Shi *et al.*, 2025] Zhouxing Shi, Qirui Jin, Zico Kolter, Suman Jana, Cho-Jui Hsieh, and Huan Zhang. Neural network verification with branch-and-bound for general nonlinearities. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2025.

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

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

[Wu *et al.*, 2020] Haoze Wu, Alex Ozdemir, Aleksandar Zeljic, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. Parallelization techniques for verifying neural networks. In *Formal Methods in Computer Aided Design*, volume 1, pages 128–137. TU Wien Academic Press, 2020.

[Wu *et al.*, 2022] Haoze Wu, Aleksandar Zeljić, Guy Katz, and Clark Barrett. Efficient neural network analysis with sum-of-infeasibilities. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 143–163. Springer, 2022.

[Wu *et al.*, 2024] Haoze Wu, Omri Isac, Aleksandar Zeljić, Teruhiro Tagomori, Matthew Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark Barrett. Marabou 2.0: A versatile formal analyzer of neural networks. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 249–264, Cham, 2024. Springer Nature Switzerland.

[Xu *et al.*, 2020] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems*, 33, 2020.

[Xu *et al.*, 2021] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and Complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *International Conference on Learning Representations*, 2021.

[Zhang *et al.*, 2018] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neu-

ral network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31:4939–4948, 2018.

[Zhang *et al.*, 2022a] Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. General cutting planes for bound-propagation-based neural network verification. *Advances in Neural Information Processing Systems*, 2022.

[Zhang *et al.*, 2022b] Huan Zhang, Shiqi Wang, Kaidi Xu, Yihan Wang, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. A branch and bound framework for stronger adversarial attacks of ReLU networks. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pages 26591–26604, 2022.

[Zhou *et al.*, 2024] Duo Zhou, Christopher Brix, Grani A Hanasusanto, and Huan Zhang. Scalable neural network verification with branch-and-bound inferred cutting planes. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[Zhou *et al.*, 2025] Duo Zhou, Jorge Chavez, Hesun Chen, Grani A. Hanasusanto, and Huan Zhang. Clip-and-verify: Linear constraint-driven domain clipping for accelerating neural network verification. In *Advances in Neural Information Processing Systems*, volume 38, 2025.