FAST AND COMPLETE: ENABLING COMPLETE NEURAL NETWORK VERIFICATION WITH RAPID AND MASSIVELY PARALLEL INCOMPLETE VERIFIERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification of neural networks (NNs) is a challenging and important problem. Existing efficient complete solvers typically require the branch-and-bound (BaB) process, which splits the problem domain into sub-domains and solves each sub-domain using faster but weaker incomplete verifiers, such as Linear Programming (LP) on linearly relaxed sub-domains. In this paper, we propose to use the backward mode linear relaxation based perturbation analysis (LiRPA) to replace LP during the BaB process, which can be efficiently implemented on the typical machine learning accelerators such as GPUs and TPUs. However, unlike LP, LiRPA when applied naively can produce much weaker bounds and even cannot check certain conflicts of sub-domains during splitting, making the entire procedure incomplete after BaB. To address these challenges, we apply a fast gradient based bound tightening procedure combined with batch splits and the design of minimal usage of LP bound procedure, enabling us to effectively use LiRPA on the accelerator hardware for the challenging complete NN verification problem and significantly outperform LP-based approaches. On a single GPU, we demonstrate an order of magnitude speedup compared to existing LP-based approaches.

1 INTRODUCTION

Although neural networks (NNs) have achieved great success on various complicated tasks, they remain susceptible to adversarial examples (Szegedy et al., 2013): imperceptible perturbations of test samples might unexpectedly change the NN predictions. Therefore, it is crucial to conduct formal verification for NNs such that they can be adopted in safety or security-critical settings.

Formally, the neural network verification problem can be cast into the following decision problem:

Given a neural network $f(\cdot)$, an input domain $\mathcal{C}$, and a property $\mathcal{P}$. $\forall x \in \mathcal{C}$, does $f(x)$ satisfy $\mathcal{P}$?

The property $\mathcal{P}$ is typically a set of desirable outputs of the NN conditioned on the inputs. Typically, consider a binary classifier $f(x)$ and a positive example x_0 ($f(x_0) \geq 0$), we can set $\mathcal{P}$ to be non-negative numbers $\mathbb{R}^+$ and x is bounded within an l_∞ norm ball $\mathcal{C} = \{x | \|x - x_0\|_\infty \leq \epsilon\}$. The success of verification guarantees that the label of x_0 cannot flip for any perturbed inputs within $\mathcal{C}$.

In this paper we study the *complete verification* setting, where given sufficient time, the verifier should give a definite "yes/no" answer for a property under verification. In the above setting, it must solve the non-convex optimization problem $\min_{x \in \mathcal{C}} f(x)$ to a global minimum. Complete NN verification is generally a challenging NP-Hard problem (Katz et al., 2017) which usually requires expensive formal verification methods such as SMT (Katz et al., 2017) or MILP solvers (Tjeng et al., 2019b). On the other hand, incomplete solvers such as convex relaxations of NNs (Salman et al., 2019) can only provide a sound analysis, i.e., they can only approximate the lower bound of $\min_{x \in \mathcal{C}} f(x)$ as $\underline{f}$ and verify the property when $\underline{f} \geq 0$. No conclusion can be drawn when $\underline{f} < 0$.

Recently, a Branch and Bound (BaB) style framework (Bunel et al., 2018; 2020b) has been adopted for efficient complete verification. BaB solves the optimization problem $\min_{x \in \mathcal{C}} f(x)$ to a global minimum by branching into multiple sub-domains recursively and bounding the solution for each sub-domain using incomplete verifiers. BaB typically uses a Linear Programming (LP) bounding procedure as an incomplete verifier to provide feasibility checking and relatively tight bounds for

each sub-domain. However, the relatively high solving cost of LPs and incapability of parallelization (especially on massively parallel hardware accelerators like GPUs or TPUs) greatly limit the performance and scalability of the existing complete BaB based verifiers.

In this paper, we aim to use fast and typically weak incomplete verifiers for complete verification. Specifically, we focus on a class of incomplete verifiers using efficient bound propagation operations, referred to as linear relaxation based perturbation analysis (LiRPA) algorithms (Xu et al., 2020). Representative algorithms in this class include convex outer adversarial polytope (Wong & Kolter, 2018), CROWN (Zhang et al., 2018) and DeepPoly (Singh et al., 2019b). LiRPA algorithms exhibit high parallelism as the bound propagation process is similar to forward or backward propagation of NNs, which can fully exploit machine learning accelerators (e.g., GPUs and TPUs).

Although LiRPA bounds are very efficient for incomplete verification, especially in training certified adversarial defenses (Wong et al., 2018; Mirman et al., 2018; Wang et al., 2018a; Zhang et al., 2020), they are generally considered too loose to be useful compared to LPs in the complete verification settings with BaB. As we will demonstrate later, using LiRPA bounds naively in BaB cannot even guarantee the completeness when splitting ReLU nodes, and thus we need additional measures to make them useful for complete verification. In fact, LiRPA methods have been used to get upper and lower bounds for each ReLU neuron in constructing tighter LPs (Bunel et al., 2018; Lu & Kumar, 2020). It was also used in (Wang et al., 2018c) for verifying small-scale problems with relatively low dimensional input domains using input splits, but splitting the input space can be quite ineffective (Bunel et al., 2018) and is unable to scale to high dimensional input case like CIFAR-10. Except one concurrent work (Bunel et al., 2020a), most complete verifiers are based on relatively expensive solvers like LP and cannot fully take benefit from massively parallel hardware (e.g., GPUs) to obtain tight bounds for accelerating large-scale complete verification problems. Our main contributions are:

- We show that LiRPA bounds, when improved with fast gradient optimizers, can potentially outperform bounds obtained by LP verifiers. This is because LiRPA allows joint optimization of both intermediate layer bounds of ReLU neurons (which determine the tightness of relaxation) and output bounds, while LP can only optimize output bounds with fixed relaxations on ReLU neurons.
- We show that BaB purely using LiRPA bounds is insufficient for complete verification due to the lack of feasibility checking for ReLU node splits. To address this issue, we design our algorithm to only invoke LP when absolutely necessary and exploits hardware parallelism when possible.
- To fully exploit the hardware parallelism on the machine learning accelerators, we use a batch splitting approach that splits on multiple neurons simultaneously, further improving our efficiency.
- On a few standard and representative benchmarks, our proposed NN verification framework can outperform previous baselines significantly, with a speedup of around 30X compared to basic BaB+LP baselines, and up to 5X compared to recent state-of-the-art complete verifiers.

2 BACKGROUND

2.1 FORMAL DEFINITION OF NEURAL NETWORK (NN) VERIFICATION

Notations of NN. For illustration, we define an L-layer feedforward NN $f : \mathbb{R}^{|x|} \to \mathbb{R}$ with L weights $\mathbf{W}^{(i)}$ ($i \in \{1, \cdots, L\}$) recursively as $f^{(i)}(x) = \mathbf{W}^{(i)} h^{(i-1)}(x)$, hidden layer $h^{(i)}(x) = \text{ReLU}(f^{(i)}(x))$, input layer $h^{(0)}(x) = x$, and final output $f(x) = f^{(L)}(x)$. For simplicity we ignore biases. We use $h_j^{(i)}$ to represent the j-th ReLU neuron in i-th layer. We focus on verifying ReLU based NNs, but our method is generalizable to other activation functions supported by LiRPA.

NN Verification Problem. Given an input x, its bounded input domain $\mathcal{C}$, and a feedforward NN $f(\cdot)$, the aim of formal verification is to prove or disprove certain properties $\mathcal{P}$ of NN outputs. Since most properties studied in previous works can be expressed as a Boolean expression over a linear equation of network output, where the linear property can be merged into the last layer weights of a NN, the ultimate goal of complete verification reduces to prove or disprove:

$$\forall x \in \mathcal{C}, f(x) \geq 0 \tag{1}$$

One way to prove Eq. 1 is to solve $\min_{x \in \mathcal{C}} f(x)$. Due to the non-convexity of NNs, finding the exact minimum of $f(x)$ over the input domain $\mathcal{C}$ is challenging as the optimization process is generally NP-complete (Katz et al., 2017). However, in practice, a *sound* approximation of the lower bound for $f(x)$, denoted as $\underline{f}$, can be more easily obtained and is sufficient to verify the property. Thus,

a good verification strategy to obtain a tight approximation $\underline{f}$ can save several magnitudes of time cost. Note that the approximation must be sound, i.e., $\forall x \in \mathcal{C}, \underline{f} \leq f(x)$ such that proving $\underline{f} \geq 0$ is sufficient to prove the desired property $f(x) \geq 0$ given $x \in \mathcal{C}$.

2.2 THE BRANCH AND BOUND (BAB) FRAMEWORK FOR NEURAL NETWORK VERIFICATION

Branch and Bound (BaB), an effective strategy in solving traditional combinatorial optimization problems, has been customized and widely adopted for NN verification (Bunel et al., 2018; 2020b). Specifically, BaB based verification framework is a recursive process, consisting of two main steps: branching and bounding. For *branching*, BaB based methods will divide the bounded input domain $\mathcal{C}$ into sub-domains $\{\mathcal{C}_i | \mathcal{C} = \bigcup_i \mathcal{C}_i\}$, each defined as a new independent verification problem. For instance, it can split a ReLU unit h to be negative and positive cases as $\mathcal{C}_0 = \mathcal{C} \cap (h \geq 0)$ and $\mathcal{C}_1 = \mathcal{C} \cap (h < 0)$ for a ReLU-based network; for each sub-domain $\mathcal{C}_i$, BaB based methods perform *bounding* to obtain a relaxed but sound lower bound $\underline{f}_{\mathcal{C}_i}$. A tightened global lower bound over $\mathcal{C}$ can then be obtained by taking the minimum values of the sub-domain lower bounds from all the sub-domains: $\underline{f} = \min_i \underline{f}_{\mathcal{C}_i}$. Branching and bounding will be performed recursively to tighten the approximated global lower bound over $\mathcal{C}$ until either (1) the global lower bound $\underline{f}$ becomes larger than 0 and prove the property or (2) a violation (e.g., adversarial example) is located in a sub-domain to disprove the property. Essentially, we build a search tree where each leaf is a sub-domain, and the property $\mathcal{P}$ can be proven only when it is valid on all leaves.

Soundness of BaB We say the verification process is sound if we can always trust the "yes" ($\mathcal{P}$ is verified) answer given by the verifier. It is straightforward to see that the whole BaB based verification process is sound as long as the bounding method used for each sub-domain $\mathcal{C}_i$ is sound.

Completeness of BaB The completeness of the BaB-based NN verification process, which was usually assumed true in some previous works (Bunel et al., 2020b; 2018), in fact, is not always true even if all possible sub-domains are considered with a sound bounding method. Additional requirements for the bounding method are required - we point out that a key factor for completeness involves feasibility checking in the bounding method which we will discuss in Section 3.2.

Branching in BaB Since branching step determines the shape of the search tree, the main challenge is to efficiently choose a good leaf to split, which can significantly reduce the total number of branches and running time. In this work we focus on branching on activation (ReLU) nodes. BaBSR (Bunel et al., 2018) includes a simple branching heuristic which assigns each ReLU node a score to estimate the improvement for tightening $\underline{f}$ by splitting it, and splits the node with the highest score.

Bounding with Linear Programming (LP) A typical bounding method used in BaB based verification is the *Linear Programming bounding procedure* (sometimes simply referred to as "LP" or "LP verifier" in our paper). Specifically, we transform the original verification problem into a linear programming problem by relaxing every activation unit as a convex (linear) domain (Ehlers, 2017) and then get the lower bound $\underline{f}_{\mathcal{C}_i}$ with a linear solver given domain $\mathcal{C}_i$. For instance, as shown in Figure 1a, $h_j^{(i)} = \text{ReLU}(x_j^{(i)})$ can be linearly relaxed with the following 3 constraints: (1) $h_j^{(i)} \geq x_j^{(i)}$; (2)$h_j^{(i)} \geq 0$; (3) $h_j^{(i)} \leq \frac{\boldsymbol{u}_j^{(i)}}{\boldsymbol{u}_j^{(i)} - \boldsymbol{l}_j^{(i)}}(x_j^{(i)} - \boldsymbol{l}_j^{(i)})$. Note that the lower bound $\boldsymbol{l}_j^{(i)}$ and upper bound $\boldsymbol{u}_j^{(i)}$ for each activation node $h_j^{(i)}$ are required in the LP construction given $\mathcal{C}_i$. They are typically computed by the existing cheap bounding methods like LiRPA variants (Wong & Kolter, 2018) with low cost. The tighter the intermediate bounds ($\boldsymbol{l}_j^{(i)}$, $\boldsymbol{u}_j^{(i)}$) are, the tighter $\underline{f}$ approximated by LP is.

2.3 LINEAR RELAXATION BASED PERTURBATION ANALYSIS (LIRPA)

Bound propagation in LiRPA Linear relaxation based perturbation analysis (LiRPA)[1], which will be used as our bound procedure in BaB, aims to find linear upper and lower bounds of NN output w.r.t input $x \in \mathcal{C}$:

$$\underline{\mathbf{A}}x + \underline{\mathbf{b}} \leq f(x) \leq \overline{\mathbf{A}}x + \overline{\mathbf{b}}, \quad x \in \mathcal{C} \tag{2}$$

A lower bound $\underline{f}$ can then be simply obtained by taking the lower bound of the linear equation $\underline{\mathbf{A}}x + \underline{\mathbf{b}}$ w.r.t input $x \in \mathcal{C}$, which can be obtained via Hölder's inequation when $\mathcal{C}$ is a ℓ_p norm ball.

[1]We only use the backward mode LiRPA bounds (e.g., CROWN and DeepPoly) in this paper.

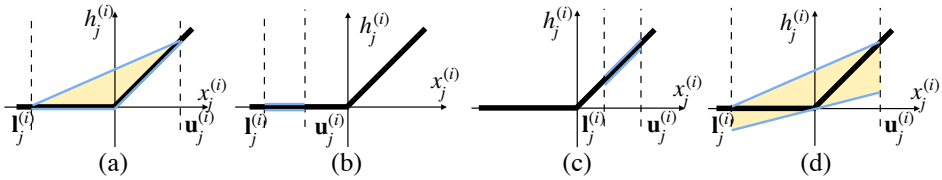


Figure 1: Relaxations of a ReLU: (a) "triangle" relaxation in LP; (b)(c) No relaxation when $\mathbf{u}_j^{(i)} \leq 0$ (always inactive) or $\mathbf{l}_j^{(i)} \geq 0$ (always active); (d) linear relaxation in LiRPA when $\mathbf{l}_j^{(i)} < 0, \mathbf{u}_j^{(i)} > 0$ (unstable).

To get the coefficients $\underline{\mathbf{A}}$, $\overline{\mathbf{A}}$, $\underline{\mathbf{b}}$, $\overline{\mathbf{b}}$, LiRPA propagates bounds of $f(x)$ as a linear function to the output of each layer, in a backward manner. At the output layer we simply have:

$$\mathbf{I}f^{(L)}(x) \leq f(x) \leq \mathbf{I}f^{(L)}(x), \quad x \in \mathcal{C} \tag{3}$$

Then, the next step is to backward propagate the identity linear relationship through a linear layer $f^{(L)}(x) = \mathbf{W}^{(L)}h^{(L-1)}(x)$ to get the linear bounds of $f(x)$ w.r.t $h^{(L-1)}$:

$$\mathbf{W}^{(L)}h^{(L-1)}(x) \leq f(x) \leq \mathbf{W}^{(L)}h^{(L-1)}(x), \quad x \in \mathcal{C} \tag{4}$$

To get the linear relationship of $f^{(L-1)}$ w.r.t $f(x)$, we need to backward propagate through ReLU layer $h^{(L-1)}(x) = \text{ReLU}(f^{(L-1)}(x))$. Since it is nonlinear, we perform linear relaxations. For illustration, considering the j-th ReLU neuron at i-th layer, $h_j^{(i)}(x) = \text{ReLU}(f_j^{(i)}(x))$, we can linearly upper and lower bound it by $\underline{a}_j^{(i)}x + \underline{b}_j^{(i)} \leq h_j^{(i)}(x) \leq \overline{a}_j^{(i)}x + \overline{b}_j^{(i)}$, where $\underline{a}_j^{(i)}, \overline{a}_j^{(i)}, \underline{b}_j^{(i)}, \overline{b}_j^{(i)}$ are:

$$\begin{cases} \underline{a}_j^{(i)} = \overline{a}_j^{(i)} = 0, \underline{b}_j^{(i)} = \overline{b}_j^{(i)} = 0 & \mathbf{u}_j^{(i)} \leq 0 \quad \text{(always inactive for } x \in \mathcal{C}) \\ \underline{a}_j^{(i)} = \overline{a}_j^{(i)} = 1, \underline{b}_j^{(i)} = \overline{b}_j^{(i)} = 0 & \mathbf{l}_j^{(i)} \geq 0 \quad \text{(always active for } x \in \mathcal{C}) \\ \underline{a}_j^{(i)} = \alpha_j^{(i)}, \overline{a}_j^{(i)} = \frac{\mathbf{u}_j^{(i)}}{\mathbf{u}_j^{(i)} - \mathbf{l}_j^{(i)}}, \underline{b}_j^{(i)} = 0, \overline{b}_j^{(i)} = -\frac{\mathbf{u}_j^{(i)}\mathbf{l}_j^{(i)}}{\mathbf{u}_j^{(i)} - \mathbf{l}_j^{(i)}} & \mathbf{l}_j^{(i)} < 0, \mathbf{u}_j^{(i)} > 0 \quad \text{(unstable for } x \in \mathcal{C}) \end{cases} \tag{5}$$

Here $\mathbf{l}_j^{(i)} \leq f_j^{(i)}(x) \leq \mathbf{u}_j^{(i)}$ are intermediate pre-activation bounds for $x \in \mathcal{C}$, and $\alpha_j^{(i)}$ is an arbitrary value between 0 and 1. The pre-activation bounds $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ can be computed by treating $f_j^{(i)}(x)$ as the output neuron with LiRPA. Figure 1(b,c,d) illustrate the relaxation for each state of ReLU neuron. With these linear relaxations, we can get the linear equation of $f^{(L-1)}$ w.r.t output $f(x)$:

$$\begin{gathered} \mathbf{W}^{(L)}\underline{\mathbf{D}}_{\alpha}^{(L-1)}f^{(L-1)}(x) + \underline{\mathbf{b}}^{(L)} \leq f(x) \leq \mathbf{W}^{(L)}\overline{\mathbf{D}}_{\alpha}^{(L-1)}f^{(L-1)}(x) + \overline{\mathbf{b}}^{(L)}, \quad x \in \mathcal{C} \\ \underline{\mathbf{D}}_{\alpha,(j,j)}^{(L)} = \begin{cases} \underline{a}_j^{(L)}, & \mathbf{W}_j^{(L)} \geq 0 \\ \overline{a}_j^{(L)}, & \mathbf{W}_j^{(L)} < 0 \end{cases}, \quad \underline{\mathbf{b}}^{(L)} = \underline{\mathbf{b}}'^{(L)\top}\mathbf{W}^{(L)}, \quad \text{where } \underline{\mathbf{b}}_j'^{(L)} = \begin{cases} \underline{b}_j^{(L)}, & \mathbf{W}_j^{(L)} \geq 0 \\ \overline{b}_j^{(L)}, & \mathbf{W}_j^{(L)} < 0 \end{cases} \end{gathered} \tag{6}$$

The diagonal matrices $\underline{\mathbf{D}}_{\alpha}^{(L-1)}$, $\overline{\mathbf{D}}_{\alpha}^{(L-1)}$ and biases reflects the linear relaxations and also considers the signs in $\mathbf{W}^{(L)}$ to maintain the lower and upper bound. The definitions for j-th diagonal element $\overline{\mathbf{D}}_{\alpha,(j,j)}^{(L)}$ and bias $\overline{\mathbf{b}}^{(L)}$ are similar, with the conditions for checking the signs of $\mathbf{W}_j^{(L)}$ swapped. Importantly, $\underline{\mathbf{D}}_{\alpha}^{(L-1)}$ has free variables $\alpha_j^{(i)} \in [0, 1]$ which do not affect correctness of the bounds. We can continue backward propagating these bounds layer by layer (e.g., $h^{(L-2)}(x)$, $f^{(L-2)}(x)$, etc) until reaching $h^{(0)}(x) = x$, getting the eventual linear equations of $f(x)$ in terms of input x:

$$\begin{gathered} \mathbf{L}(x, \boldsymbol{\alpha}) \leq f(x) \leq \mathbf{U}(x, \boldsymbol{\alpha}), \quad \forall x \in \mathcal{C}, \quad \text{where} \\ \mathbf{L}(x, \boldsymbol{\alpha}) = \mathbf{W}^{(L)}\underline{\mathbf{D}}_{\alpha}^{(L-1)} \cdots \underline{\mathbf{D}}_{\alpha}^{(1)}\mathbf{W}^{(1)}x + \underline{\mathbf{b}}, \quad \mathbf{U}(x, \boldsymbol{\alpha}) = \mathbf{W}^{(L)}\overline{\mathbf{D}}_{\alpha}^{(L-1)} \cdots \overline{\mathbf{D}}_{\alpha}^{(1)}\mathbf{W}^{(1)}x + \overline{\mathbf{b}} \end{gathered} \tag{7}$$

Here $\boldsymbol{\alpha}$ denotes $\alpha_j^{(i)}$ for all unstable ReLU neurons in NN. The obtained bounds $(\mathbf{L}(x, \boldsymbol{\alpha}), \mathbf{U}(x, \boldsymbol{\alpha}))$ of $f(x)$ are linear functions in terms of x. Beyond the simple feedforward NN presented here, LiRPA can support more complicated NN architectures like DenseNet and Transformers by computing $\mathbf{L}$ and $\mathbf{U}$ automatically and efficiently on general computational graphs (Xu et al., 2020).

Soundness of LiRPA The above backward bound propagation process guarantees that $\mathbf{L}(x, \alpha)$ and $\mathbf{U}(x, \alpha)$ soundly bound $f(x)$ for all $x \in \mathcal{C}$. Detailed proofs can be found in (Zhang et al., 2018; Singh et al., 2019b) for feedforward NNs and (Xu et al., 2020) for general networks.

3 Proposed Algorithm

Overview In this section, we will first introduce our proposed efficient optimization of LiRPA bounds on GPUs that can allow us to achieve tight approximation on par with LP or even tighter for

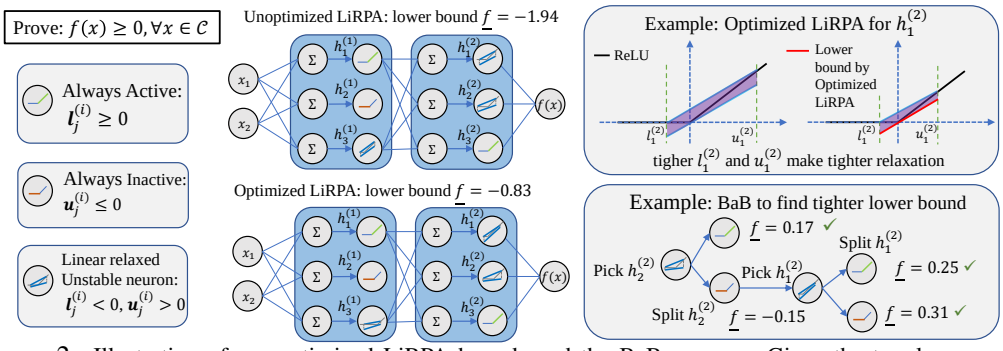


Figure 2: Illustration of our optimized LiRPA bounds and the BaB process. Given the two-layer neural network, we aim to verify output $f(x) \geq 0$. Optimized LiRPA chooses optimized slopes for ReLU lower bounds, allowing tightening the intermediate layer bounds $\boldsymbol{l}_j^{(i)}$ and $\boldsymbol{u}_j^{(i)}$ and also the output layer lower bound $\underline{f}$. BaB splits two unstable neurons $h_2^{(2)}$ and $h_1^{(2)}$ to improve $\underline{f}$ and verify all sub-domains ($\underline{f} \geq 0$ for all cases).

some cases but in a much faster manner. In Fig. 2, we provide a two-layer NN example to illustrate how our optimized LiRPA can improve the performance of BaB verification. In Section 3.2, we then show that feasibility checking is important to guarantee the completeness of BaB, and BaB using LiRPA without feasibility checking will end up to be incomplete. To ensure completeness, we design our algorithm with minimal usage of LP for checking feasibility of splits. Finally, we propose a batch split design by solving a batch of sub-domains in a massively parallel manner on GPUs to fully leverage the benefits of cheap and parallelizable LiRPA. We further improve BaBSR in a parallel fashion for branching and we summarize the detailed proposed algorithm in Section 3.4.

3.1 Optimized LiRPA for Complete Verification

Concrete outer bounds with optimizable parameters We propose to use LiRPA as the bounding step in BaB. A pair of sound and concrete lower bound and upper bound $(\underline{f}, \overline{f})$ to $f(x)$ can be obtained according to Eq. 7 given fixed $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0$:

$$\underline{f}(\boldsymbol{\alpha}_0) = \min_{x \in \mathcal{C}} \mathbf{L}(x, \boldsymbol{\alpha}_0), \quad \overline{f}(\boldsymbol{\alpha}_0) = \max_{x \in \mathcal{C}} \mathbf{U}(x, \boldsymbol{\alpha}_0) \tag{8}$$

Because $\mathbf{L}$, $\mathbf{U}$ are linear functions w.r.t x when $\boldsymbol{\alpha}_0$ is fixed, it is easy to solve Eq. 8 using Hölder's inequality when $\mathcal{C}$ is a ℓ_p norm ball (Xu et al., 2020). In incomplete verification settings, $\boldsymbol{\alpha}$ can be set via certain heuristics (Zhang et al., 2018). Salman et al. (2019) showed that, the variable $\boldsymbol{\alpha}$ is equivalent to dual variables in the LP relaxed verification problem (Wong & Kolter, 2018). Thus, an optimal selection of $\boldsymbol{\alpha}$ given the *same* pre-activation bounds $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ can in fact, lead to the the same optimal solution for $\underline{f}$ and $\overline{f}$ as in LP.

Previous complete verifiers typically use LiRPA variants to obtain intermediate layer bounds $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ to construct a LP problem (e.g., Bunel et al. (2018); Lu & Kumar (2020) used convex outer adversarial polytope (Wong & Kolter, 2018)), and solve the LP to obtain bounds at output layer. The main reason for using LP is that it typically produces much tighter bounds than LiRPA when $\boldsymbol{\alpha}$ is not optimized. To replace LP with fast and accelerator-friendly LiRPA algorithms in the complete verification setting, we use optimized LiRPA to produce tighter bounds:

$$\underline{f} = \min_{\boldsymbol{\alpha}} \min_{x \in \mathcal{C}} \mathbf{L}(x, \boldsymbol{\alpha}), \quad \overline{f} = \max_{\boldsymbol{\alpha}} \max_{x \in \mathcal{C}} \mathbf{U}(x, \boldsymbol{\alpha}), \quad \alpha_j^{(i)} \in [0, 1] \tag{9}$$

The inner minimization or maximization has closed form solutions (Xu et al., 2020) based on Hölder's inequality, so we only need to optimize on $\boldsymbol{\alpha}$. Since we use a differentiable framework (Xu et al., 2020) to compute the LiRPA bound functions $\mathbf{L}$ and $\mathbf{U}$, the gradients $\frac{\partial \mathbf{L}}{\partial \boldsymbol{\alpha}}$ and $\frac{\partial \mathbf{U}}{\partial \boldsymbol{\alpha}}$ can be obtained easily. Optimization over $\boldsymbol{\alpha}$ can be done via projected gradient descent (each coordinate of $\boldsymbol{\alpha}$ is constrained in $[0, 1]$). Since the gradient computation and optimization are done on GPUs, the bounding process is still very fast and can be one or two magnitudes faster than solving an LP.

Optimized LiRPA bounds can be tighter than LP Solving Eq. 9 using gradient descent with a small number of iterations cannot guarantee to converge to the global optima, so it seems the bounds must be looser than LP. Counter-intuitively, by optimizing $\boldsymbol{\alpha}$, we can potentially *obtain tighter bounds* than LP. When a LP relaxation is constructed, preactivation bounds $\mathbf{l}_j^{(i)}$, $\mathbf{u}_j^{(i)}$ must be fixed to construct the "triangle" relaxation for the j-th ReLU neuron in layer i. During the LP

optimization process, only the output bounds are optimized; $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ stay unchanged. However, in the LiRPA formulation, $\mathbf{L}(x, \boldsymbol{\alpha})$ and $\mathbf{U}(x, \boldsymbol{\alpha})$ are complex functions of $\boldsymbol{\alpha}$: since pre-activation bounds $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ are also computed by LiRPA, they depend on all $\alpha_{j'}^{(i')}(0 < i' < i)$ in previous layers. Thus, the gradients $\frac{\partial \mathbf{L}}{\partial \boldsymbol{\alpha}}$ and $\frac{\partial \mathbf{U}}{\partial \boldsymbol{\alpha}}$ can tighten output layer bounds $\underline{f}$ and $\overline{f}$ indirectly by tightening intermediate layer bounds $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$, forming a tighter convex relaxation for the next iteration. An LP solver cannot achieve this because adding $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ as optimization variables makes the problem non-linear. This is the key to our success of applying LiRPA based bounds for the complete verification setting, where tighter bounds are essential.

In Figure 3, we illustrate our optimized LiRPA bounds and the LP solution. Initially, we use LiRPA with $\boldsymbol{\alpha}$ set via a fast heuristic to compute intermediate layer bounds $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$, and then use them to build a relaxed LP problem. The solution to this initial LP problem is the red line. This LP solution is much tighter than the LiRPA solution with a heuristically set $\boldsymbol{\alpha}$, shown as the left-most point of the blue line. Then, we optimize $\boldsymbol{\alpha}$ with gradient decent, and LiRPA quickly outperforms this initial LP solution. The optimized $\boldsymbol{\alpha}$ also produces tighter intermediate layer bounds, $\mathbf{l}'^{(i)}_j$ and $\mathbf{u}'^{(i)}_j$. We can create a new LP with optimized intermediate layer bounds $\mathbf{l}'^{(i)}_j$ and $\mathbf{u}'^{(i)}_j$, whose solution is the light blue line. This new LP only produces a slightly tighter bound than LiRPA with optimized $\boldsymbol{\alpha}$. In most previous works on complete verification, $\mathbf{l}_j^{(i)}$ and $\mathbf{u}_j^{(i)}$ are obtained from unoptimized LiRPA bounds or even weaker methods such as interval arithmetic, so the corresponding LP is similar to the red line in Figure 3; instead, our optimized LiRPA bounds can produce quite competitive bounds, and also exploit parallel acceleration from machine learning accelerators. The tightness and efficiency of our optimized LiRPA procedure lead to huge improvements in verification time compared to existing baselines.

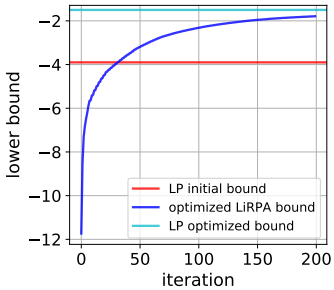


Figure 3: Compare LP optimized bound with LP initial bound and optimized LiRPA bound.

ReLU Split Constraints In the BaB process, when a ReLU neuron $h_j^{(i)}$ is split into two sub-domains ($h_j^{(i)} \geq 0$ and $h_j^{(i)} < 0$), we simply set $\boldsymbol{l}_j^{(i)} \geq 0$ and $\boldsymbol{u}_j^{(i)} < 0$ in the bound propagation process. This allows LiRPA to tighten bounds because a split ReLU neuron essentially becomes linear, reducing relaxation errors. However, when splits are added, LiRPA and LP are not equivalent even under fixed $\mathbf{l}_j^{(i)}$, $\mathbf{u}_j^{(i)}$ and optimal $\boldsymbol{\alpha}$. After splits, LiRPA cannot check certain constraints where LP is capable to, as we will discuss in the next section.

3.2 Completeness with Minimal Usage of LP Bounding Procedure

Even though our optimized LiRPA can bring us huge speed improvement over LP for BaB based verification, we observe that it may end up to be incomplete due to the lack of feasibility checking: it cannot detect some conflicting settings of ReLU splits. We state such an observation in Theorem 3.1:

Theorem 3.1 (Incompleteness without feasibility checking) *When using LiRPA variants described in Section 2.3 as the bounding procedure, BaB based verification is incomplete.*

We prove the theorem by giving a counter-example in Appendix A.1 where *all* ReLU neurons are split and thus LiRPA runs on a linear network for each sub-domain. As a result, LiRPA can still be indecisive for the verification problem. The main reason is that LiRPA variants will lose the feasibility information encoded by the sub-domain constraints. For illustration, consider a sub-domain $\mathcal{C}_i = C \cap (h_{j_1}^{(i_1)} < 0) \cap (h_{j_2}^{(i_2)} \geq 0)$, LiRPA will force $h_{j_1}^{(i_1)}(x) = 0$ (inactive ReLU, a zero function) and $h_{j_2}^{(i_2)}(x) = f_{j_2}^{(i_2)}(x)$ (active ReLU, an identity function) respectively and propagate these bounds to get the approximated lower bound $\underline{f}_{\mathcal{C}_i}$. However, the split feasibility constraint $(h_{j_1}^{(i_1)} < 0) \cap (h_{j_2}^{(i_2)} \geq 0)$ is ignored, so two conflict splits may be conducted (e.g., when $h_{j_1}^{(i_1)} < 0$, $h_{j_2}^{(i_2)}$ cannot be ≥ 0). On the contrary, LP can fully preserve such feasibility information due to the linear solver involved and detect the infeasible sub-domains. Then, in Theorem 3.2 we show that the minimal usage of feasibility checking with LP can guarantee the completeness of BaB with LiRPA.

Algorithm 1 Parallel BaB with optimized LiRPA bounding (we highlight the differences between our algorithm and regular BaB (Bunel et al., 2018) in blue. Comments are in brown.)

1: **Inputs**: $f, \mathcal{C}, n$ (batch size), η (threshold to switch to LP)
2: $(\underline{f}, \overline{f}) \leftarrow$ `optimized_LiRPA`$(f, [\mathcal{C}])$
3: $\mathbb{P} \leftarrow [(\underline{f}, \overline{f}, \mathcal{C})]$ ▷ $\mathbb{P}$ is the set of all unverified sub-domains
4: **while** $\underline{f} < 0$ **and** $\overline{f} \geq 0$ **do**
5: $\quad(\mathcal{C}_1, \ldots, \mathcal{C}_n) \leftarrow$ `batch_pick_out`$(\mathbb{P}, n)$ ▷ Pick sub-domains to split and removed them from $\mathbb{P}$
6: $\quad[\mathcal{C}_1^l, \mathcal{C}_1^u, \ldots, \mathcal{C}_n^l, \mathcal{C}_n^u] \leftarrow$ `batch_split`$(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ ▷ Each $\mathcal{C}_i$ splits into two sub-domains $\mathcal{C}_i^l$ and $\mathcal{C}_i^u$
7: $\quad\left[\underline{f}_{\mathcal{C}_1^l}, \overline{f}_{\mathcal{C}_1^l}, \underline{f}_{\mathcal{C}_1^u}, \overline{f}_{\mathcal{C}_1^u}, \ldots, \underline{f}_{\mathcal{C}_n^l}, \overline{f}_{\mathcal{C}_n^l}, \underline{f}_{\mathcal{C}_n^u}, \overline{f}_{\mathcal{C}_n^u}\right] \leftarrow$ `optimized_LiRPA`$(f, [\mathcal{C}_1^l, \mathcal{C}_1^u, \ldots, \mathcal{C}_n^l, \mathcal{C}_n^u])$ ▷ Compute lower and upper bounds using LiRPA for each sub-domain on GPUs in a batch
8: $\quad\mathbb{P} \leftarrow \mathbb{P} \bigcup$ `Domain_Filter` $\left([\underline{f}_{\mathcal{C}_1^l}, \overline{f}_{\mathcal{C}_1^l}, \mathcal{C}_1^l], [\underline{f}_{\mathcal{C}_1^u}, \overline{f}_{\mathcal{C}_1^u}, \mathcal{C}_1^u], \ldots, [\underline{f}_{\mathcal{C}_n^l}, \overline{f}_{\mathcal{C}_n^1}, \mathcal{C}_n^l], [\underline{f}_{\mathcal{C}_n^u}, \overline{f}_{\mathcal{C}_n^u}, \mathcal{C}_n^u]\right)$ ▷ Filter out verified sub-domains, insert the left domains back to $\mathbb{P}$
9: $\quad\underline{f} \leftarrow \min\{\underline{f}_{\mathcal{C}_i} \mid (\underline{f}_{\mathcal{C}_i}, \mathcal{C}_i) \in \mathbb{P}\}, i = 1, \ldots, n$ ▷ To ease notation, $\mathcal{C}_i$ here indicates both $\mathcal{C}_i^u$ and $\mathcal{C}_i^l$
10: $\quad\overline{f} \leftarrow \min\{\overline{f}_{\mathcal{C}_i} \mid (\overline{f}_{\mathcal{C}_i}, \mathcal{C}_i) \in \mathbb{P}\}, i = 1, \ldots, n$
11: $\quad$**if** `length`$(\mathbb{P}) > \eta$ **then** ▷ Fall back to LP for completeness
12: $\quad\quad\left[\underline{f}_{\mathcal{C}_1^l}, \overline{f}_{\mathcal{C}_1^l}, \underline{f}_{\mathcal{C}_1^u}, \overline{f}_{\mathcal{C}_1^u}, \ldots, \underline{f}_{\mathcal{C}_n^l}, \overline{f}_{\mathcal{C}_n^l}, \underline{f}_{\mathcal{C}_n^u}, \overline{f}_{\mathcal{C}_n^u},\right] \leftarrow$ `compute_bound_LP`$(f, [\mathcal{C}_1^l, \mathcal{C}_1^u, \ldots, \mathcal{C}_n^l, \mathcal{C}_n^u])$
13: $\quad\quad\mathbb{P} \leftarrow \mathbb{P} \bigcup$ `Domain_Filter` $\left([\underline{f}_{\mathcal{C}_1^l}, \overline{f}_{\mathcal{C}_1^l}, \mathcal{C}_1^l], [\underline{f}_{\mathcal{C}_1^u}, \overline{f}_{\mathcal{C}_1^u}, \mathcal{C}_1^u], \ldots, [\underline{f}_{\mathcal{C}_n^l}, \overline{f}_{\mathcal{C}_n^1}, \mathcal{C}_n^l], [\underline{f}_{\mathcal{C}_n^u}, \overline{f}_{\mathcal{C}_n^u}, \mathcal{C}_n^u]\right)$
14: **Outputs:** $\underline{f}, \overline{f}$

Theorem 3.2 (Minimal feasibility checking for completeness) *When using LiRPA variants described in Section 2.3 as the bounding procedure, BaB based verification is complete if all infeasible leaf sub-domains (i.e., sub-domains cannot be further split) are detected by linear programming.*

We prove the theorem in Appendix A.2, where we show that by checking the feasibility of splits with LP, we can eliminate the cases where incompatible splits are chosen in the LiRPA BaB process. Since LP is slow while LiRPA is highly efficient, we propose to only use LP when the LiRPA based bounding process is stuck, either (1) when partitioning and bounding new sub-domains with LiRPA cannot further improve the bounds, or (2) when all unstable neurons have been split. In this way, the infeasible sub-domains can be eventually detected by occasional usage of LP while the advantage of massive parallel LiRPA on GPUs is fully enjoyed. We will describe our full algorithm in Sec. 3.4.

3.3 Batch Splits

In state-of-the-art BaB methods (Bunel et al., 2020b; Lu & Kumar, 2020), only one sub-domain can be selected to split during each branching step. Benefiting from our design of using cheap and GPU-friendly LiRPA bounds, we can select a batch of sub-domains to split and propagate LiRPA bounds for them in a batch. Such a batching splitting design can greatly improve hardware efficiency on GPUs. Given a batch size n that allows us to fully use the memory of one GPU device, we can obtain n bounds simultaneously. This can help us grow the search tree on a single leaf by a depth of $\log_2 n$, or split $n/2$ leaf nodes at the same time, accelerating by up to n times.

3.4 Our Complete Verification Framework

Our LiRPA based complete verification framework is presented in Alg. 1. The algorithm takes a target NN function f and a domain $\mathcal{C}$ as inputs. We run optimized LiRPA to get initial bounds $(\underline{f}, \overline{f})$ for $x \in \mathcal{C}$ (Line 2). Then we utilize the power of GPUs to split in parallel and maintain a global set $\mathbb{P}$ storing all the sub-domains which cannot be verified with optimized LiRPA (Line 5-10). Specifically, `batch_pick_out` improves BaBSR (Bunel et al., 2018) in a parallel manner to select n sub-domains in $\mathbb{P}$ and determine the corresponding ReLU neuron to split for each of them. If the length of $\mathbb{P}$ is less than n, then we reduce n to the length of $\mathbb{P}$. `batch_split` splits each selected $\mathcal{C}_i$ to two sub-domains $\mathcal{C}_i^l$ and $\mathcal{C}_i^u$ by forcing the selected unstable ReLU neuron to be positive and negative, respectively. `optimize_LiRPA` runs optimized LiRPA in parallel as a batch and returns the lower and upper bounds for n selected sub-domains simultaneously. `Domain_Filter` filters out verified sub-domains (proved with $\underline{f}_{\mathcal{C}_i} \geq 0$) and we insert the remaining ones to $\mathbb{P}$. The loop breaks if the property is proved ($\underline{f} \geq 0$) or a counter-example is found in any sub-domain ($\overline{f} < 0$).

To avoid excessive splits, we set the maximum length of the sub-domains to η (Line 12). Once the length of $\mathbb{P}$ reaches this threshold, `compute_bound_LP` will be called. It solves these η sub-domains

by LP (one by one in a loop, or in parallel if using multiple CPUs is allowed) with optimized LiRPA computed intermediate layer bounds. If a sub-domain $\mathcal{C}_i \in \mathbb{P}$ (which previously cannot be verified by LiRPA) is proved or detected to be infeasible by LP, as an effective heuristic, we will backtrack and prioritize to check its parent node with LP. If the parent sub-domain is also proved or infeasible, we can prune all its child nodes to greatly reduce the size of the search tree.

Completeness of our framework Our algorithm is *complete*, because we follow Theorem 3.2 and check feasibility of all split sub-domains that have deep BaB search tree depth (length of $\mathbb{P}$ reaches threshold η), forming a superset of the worst case where all ReLU neurons are split.

4 EXPERIMENTS

In this section, we compare our verifier against the state-of-the-art ones to illustrate the effectiveness of our proposed framework. Overall, our verifier is about 4X, 7X and 20X faster than the best LP-based verifier (Lu & Kumar, 2020) on the Base, Wide and Deep models, respectively.

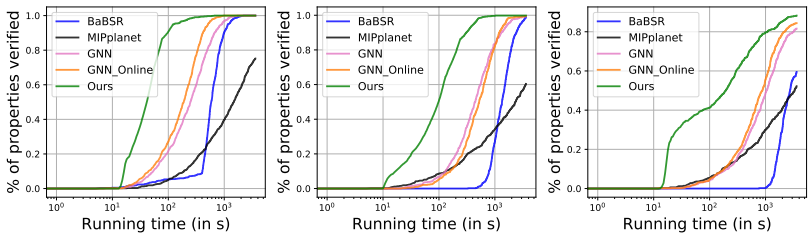
Setup We follow the most challenging experimental setup used in the state-of-the-art verifiers GNN-ONLINE (Lu & Kumar, 2020) and BABSR (Bunel et al., 2020b). Specifically, we evaluate on CIFAR10 dataset on three NNs: Base, Wide and Deep. The dataset is categorized into three difficulty levels: Easy, Medium, and Hard, which is generated according to the performance of BaBSR. The verification task is defined as given a l_∞ norm perturbation less than ϵ, the classifier will not predict a specific (predefined) wrong label for each image x (see Appendix B). We set batch size $n = 64$ and threshold $\eta = 12000$. More details on experimental setup are provided in Appendix B.

Comparisons against state-of-the-art verifiers We include five different methods for comparison: (1) BaBSR (Bunel et al., 2020b), a BaB and LP based verifier using a simple ReLU split heuristic; (2) MIPPLANET (Ehlers, 2017), a customized MIP solver for NN verification where unstable ReLU neurons are randomly selected for split; (3) GNN (Lu & Kumar, 2020) and (4) GNN-ONLINE (Lu & Kumar, 2020) are BaB and LP based verifiers using a learned graph neural network (GNN) to guide the ReLU split. (5) PROXIMAL BABSR (Bunel et al., 2020a) is a very recently proposed verification framework based on Lagrangian decomposition which also supports GPU acceleration without solving LPs.[2] All methods use 1 CPU with 1 GPU. The timeout threshold is 3,600 seconds.

For the Base model, Table 1 shows that we are around 10X faster than baseline BaBSR and around 4X faster than GNN split baselines. The accumulative solved properties with increasing runtime is shown in Figure 4. In all our experiments, we use the basic heuristic in BaBSR for branching and do not use GNNs, so our speedup comes purely from the faster LiRPA based bounding procedure. We are also competitive against Lagrangian decomposition on GPUs, especially on hard properties.

Table 1: Performance of various methods on the Base (smallest) model. We compare each method's avg. solving time, the avg. number of branches required, and the percentage of timed out properties.

	Easy			Medium			Hard		
Method	time(s)	branches	%timeout	time(s)	branches	%timeout	time(s)	branches	%timeout
BABSR	522.48	585.27	0.0	1335.40	1471.55	0.0	3017.11	2350.80	40.2
MIPPLANET	1462.24	-	16.5	1912.25	-	43.5	2172.23	-	46.2
GNN	312.93	301.72	0.0	625.12	547.40	0.9	1017.35	1059.42	18.3
GNN-ONLINE	207.43	269.81	0.0	613.15	542.51	0.4	844.35	959.40	15.8
PROXIMAL BABSR[2]	73.61	1377.67	0.4	129.54	3269.535	**0.3**	1126.86	12036.00	21.9
OURS	**56.49**	6493.87	0.0	**124.38**	7570.41	0.8	**267.35**	18241.39	**11.7**



(a) Easy (b) Medium (c) Hard

Figure 4: Cactus plots for our method and other baselines in Easy, Medium and Hard properties for base model. We plot the percentage of solved properties with growing running time.

[2]The code link provided in (Bunel et al., 2020a) does not contain complete verification, so we directly report the best results in their paper in our table. Since we use a similar GPU model as theirs, the time is comparable.

Table 2: Performance comparison on larger models (Wide and Deep) in similar metrics as in Table 1.

	Wide			Deep		
Method	time	branches	%timeout	time	branches	%timeout
BABSR	2305.65	646.22	50.3	2455.17	299.04	54.0
MIPPLANET	5712.40	-	74.3	5122.54	-	60.8
GNN	1491.52	352.41	20.1	1587.63	169.82	20.8
GNN-ONLINE	1343.03	338.85	18.9	1569.71	166.82	18.4
PROXIMAL BABSR[2]	891.16	3343.60	18.0	384.98	1655.52	6.8
OURS	**181.30**	15795.45	**12.7**	**78.17**	592.20	**6.0**

Performance on Larger Models In Table 2, we show that our verifier is more scalable on larger (wider or deeper) NNs compared to other state-of-the-art verifiers. Our method enjoys efficient GPU acceleration particularly on deep models and can achieve 30X speedup compared to BABSR, and we are also significantly faster than Lagrangian decomposition based GPU verifier (Proximal BaBSR). When compared to the state-of-the-art LP based BaB, GNN-ONLINE, our method can save 20X running time. In Appendix C, we analyze the effectiveness of optimized LiRPA and batch splits separately, and find that optimized LiRPA is crucial for NN verification. Performance comparisons of our proposed framework on CPU cores without GPU acceleration are included in Appendix D.

5 RELATED WORK

Complete verifiers Early complete verifiers rely on satisfiability modulo theory (SMT) (Katz et al., 2017; Huang et al., 2017; Ehlers, 2017) and mixed integer linear programming (MILP) (Tjeng et al., 2019a; Dutta et al., 2018), and they typically do not scale well. Higher order logic provers such as proof assistant (Bentkamp et al., 2018) can also be potentially used for NN verification, but their scalability to the NN setting has not been demonstrated. Recently, Bunel et al. (2018) unified many approaches used in various complete verifiers into a BaB framework. An LP based bounding procedure is used in most of the existing BaB framework (Bunel et al., 2018; Wang et al., 2018c; Royo et al., 2019; Lu & Kumar, 2020). For branching, two categories of branching strategies were proposed: (1) input node branching (Wang et al., 2018c; Bunel et al., 2020b; Royo et al., 2019; Anderson et al., 2019) where input features are divided into sub-domains, and (2) activation node (especially, ReLU) branching (Katz et al., 2017; Bunel et al., 2018; Wang et al., 2018b; Ehlers, 2017; Lu & Kumar, 2020) where hidden layer activations are split into sub-domains. Bunel et al. (2018) found that input node branching cost is exponential to input dimension. Thus, many state-of-the-art verifiers use activation node branching instead, focusing on heuristics to select good nodes to split. BaBSR (Bunel et al., 2018) prioritizes ReLUs for splitting based on their pre-activation bounds; Lu & Kumar (2020) used a graph neural network (GNN) to learn good splitting heuristics. Our work focuses on improving bounding and can use better branching heuristics to achieve further speedup.

Two mostly relevant concurrent works using GPUs for accelerating NN verification are: (1) GPUPoly (Müller et al., 2020), an extension of DeepPoly on CUDA, is *still an incomplete verifier*. Also, it is implemented in CUDA C++, requiring manual effort for customization and gradient computation, so it is not easy to get the gradients for optimizing bounds as we have done in Section 3.1. (2) Lagrangian Decomposition (Bunel et al., 2020a) is a GPU-accelerated BaB based complete verifier that iteratively tightens the bounds based on a Lagrangian decomposition optimization formulation and does not reply on LP. However, it solves a much more complicated optimization problem than LiRPA, and typically requires hundreds of iterations to converge for a single sub-domain.

Incomplete verifiers Many incomplete verification methods rely on convex relaxations of NN, replacing nonlinear activations like ReLUs with linear constraints (Wong & Kolter, 2018; Zhang et al., 2018; Weng et al., 2018; Wang et al., 2018b; Gehr et al., 2018; Singh et al., 2018a;b; 2019b;a) or semidefinite constraints (Raghunathan et al., 2018; Dvijotham et al., 2020; Dathathri et al., 2020). Tighter relaxations require more computation and memory in general. We refer the readers to (Salman et al., 2019) for a comprehensive survey. Recently, Xu et al. (2020) categorized the family of linear relaxation based incomplete verifiers into LiRPA framework, allowing efficient implementation on machine learning accelerators. Our work uses LiRPA as the bounding procedure for complete verification and exploits its computational efficiency to accelerate complete verification.

6 CONCLUSION

We use a LiRPA based incomplete NN verifier to accelerate the bounding procedure in branch and bound (BaB) for complete NN verification on massively parallel accelerators. We use a fast gradient based procedure to tighten LiRPA bounds. We study the completeness of BaB with LiRPA, and show up to 5X speedup compared to state-of-the-art verifiers across multiple models and properties.

REFERENCES

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

Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. In *International Joint Conference on Automated Reasoning*, pp. 28–46. Springer, 2018.

Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Lagrangian decomposition for neural network verification. *Conference on Uncertainty in Artificial Intelligence (UAI) 2020*, 2020a.

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, P Kohli, P Torr, and P Mudigonda. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020b.

Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. In *Advances in Neural Information Processing Systems*, pp. 4790–4799, 2018.

Sumanth Dathathri, Krishnamurthy Dvijotham, Alexey Kurakin, Aditi Raghunathan, Jonathan Uesato, Rudy R Bunel, Shreya Shankar, Jacob Steinhardt, Ian Goodfellow, Percy S Liang, et al. Enabling certification of verification-agnostic networks via memory-efficient semidefinite programming. *Advances in Neural Information Processing Systems*, 33, 2020.

Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pp. 121–138. Springer, 2018.

Krishnamurthy Dj Dvijotham, Robert Stanforth, Sven Gowal, Chongli Qin, Soham De, and Pushmeet Kohli. Efficient neural network verification with exactness characterization. In *Uncertainty in Artificial Intelligence*, pp. 497–507. PMLR, 2020.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pp. 269–286. Springer, 2017.

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

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pp. 3–29. Springer, 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 *International Conference on Computer Aided Verification*, pp. 97–117. Springer, 2017.

Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. *International Conference on Learning Representation (ICLR)*, 2020.

Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pp. 3575–3583, 2018.

Christoph Müller, Gagandeep Singh, Markus Püschel, and Martin Vechev. Neural network robustness verification on gpus. *arXiv preprint arXiv:2007.10868*, 2020.

Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. Semidefinite relaxations for certifying robustness to adversarial examples. In *Advances in Neural Information Processing Systems*, pp. 10877–10887, 2018.

Vicenc Rubies Royo, Roberto Calandra, Dusan M Stipanovic, and Claire Tomlin. Fast neural network verification via shadow prices. *arXiv preprint arXiv:1902.07247*, 2019.

Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. A convex relaxation barrier to tight robustness verification of neural networks. In *Advances in Neural Information Processing Systems 32*, pp. 9832–9842, 2019.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, pp. 10825–10836, 2018a.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations*, 2018b.

Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems*, pp. 15072–15083, 2019a.

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):41, 2019b.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2013.

Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *ICLR*, 2019a.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019b. URL `https://openreview.net/forum?id=HyGIdiRqtm`.

Shiqi Wang, Yizheng Chen, Ahmed Abdou, and Suman Jana. Mixtrain: Scalable training of formally robust neural networks. *arXiv preprint arXiv:1811.02625*, 2018a.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pp. 6367–6377, 2018b.

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

Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pp. 5273–5282, 2018.

Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pp. 5283–5292, 2018.

Eric Wong, Frank Schmidt, Jan Hendrik Metzen, and J Zico Kolter. Scaling provable adversarial defenses. In *NIPS*, 2018.

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.

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *Advances in neural information processing systems*, pp. 4939–4948, 2018.

Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. In *International Conference on Learning Representations*, 2020.

A PROOFS

A.1 PROOF OF THEOREM 3.1

We prove Theorem 3.1 by providing a simple counterexample, and we illustrate the necessity of feasibility checking for the completeness of BaB based verification. Consider an NN with only two hidden nodes $h_1 = h_2 = \text{ReLU}(x)$ and one dimension input x, and its output function is defined as $f(x) = h_1 - h_2 = \text{ReLU}(x) - \text{ReLU}(x)$. As a verification problem, we want to verify the property $f(x) \geq 0$ where $x = [-1, 1]$. Since hidden nodes h_1 and h_2 are exactly the same, the ground-truth output range is $f^*(x) = [0, 0]$. A complete BaB based verifier is expected to obtain that optimal bound and prove the property after split h_1 and h_2 together while BaB with only LiPRA cannot guarantee that completeness. Specifically, BaB with only LiRPA will split the original domain $x = [-1, 1]$ into four sub-domains and approximate the bound with LiRPA respectively:

(1) (feasible) sub-domain $x = [-1, 1], h_1 > 0, h_2 > 0$ with output $f = [x, x] - [x, x] = [0, 0]$

(2) (feasible) sub-domain $x = [-1, 1], h_1 \leq 0, h_2 \leq 0$ with output $f = [0, 0] - [0, 0] = [0, 0]$

(3) (infeasible) sub-domain $x = [-1, 1], h_1 \leq 0, h_2 > 0$ with output $f = [0, 0] - [x, x] = [-1, 1]$

(4) (infeasible) sub-domain $x = [-1, 1], h_1 > 0, h_2 \leq 0$ with output $f = [x, x] - [0, 0] = [-1, 1]$

Only the first two split sub-domains are feasible and therefore the ground-truth lower bound 0 can be obtained by taking the minimum of the estimated bounds from sub-domains (1) and (2). However, pure LiRPA is not able to tell the infeasibility of sub-domains (3) and (4) and thus BaB with pure LiRPA will report the minimum -1 got from all these four sub-domains as the global lower bound for the original input domain, ending up not being able to verify the property, i.e., incomplete.

A.2 PROOF OF THEOREM 3.2

We prove Theorem 3.2 by considering the worst case where all unstable ReLU neurons are split.

Given a neural network function f with input domain $\mathcal{C}$, assume there are N unstable ReLU neurons $\{h_i | i = 1, \cdots, N\}$. In the worst case, we have 2^N leaf sub-domains $\mathcal{S} = \{\mathcal{C}_i | i = 1, \cdots, 2^N\}$, where each $\mathcal{C}_i$ corresponds to one assignment of unstable ReLU neuron splits. For example, we can have

$$
\begin{aligned}
\mathcal{C}_1 &= \mathcal{C} \cap (h_1 \geq 0) \cap (h_2 \geq 0) \cap \cdots \cap (h_N \geq 0) \\
\mathcal{C}_2 &= \mathcal{C} \cap (h_1 < 0) \cap (h_2 \geq 0) \cap \cdots \cap (h_N \geq 0) \\
\mathcal{C}_3 &= \mathcal{C} \cap (h_1 \geq 0) \cap (h_2 < 0) \cap \cdots \cap (h_N \geq 0) \\
\mathcal{C}_4 &= \mathcal{C} \cap (h_1 < 0) \cap (h_2 < 0) \cap \cdots \cap (h_N \geq 0) \\
&\cdots
\end{aligned}
$$

Note that by definition $\mathcal{C} = \cup_{\mathcal{C}' \in \mathcal{S}} \mathcal{C}'$; in other words, all the 2^N split sub-domains combined will be the same as the original input domain.

Not all of the sub-domains are actually feasible, due to the consistency requirements between neurons. For example, in our proof in Section A.1, h_1 and h_2 cannot be both ≥ 0 or < 0. We can divide the sub-domains $\mathcal{S}$ into two mutually exclusive sub-sets, $\mathcal{S}^{\text{feas}}$ for all the feasible sub-domains, and $\mathcal{S}^{\text{infeas}}$ for all the infeasible sub-domains. We have $\mathcal{C} = \cup_{\mathcal{C}' \in \mathcal{S}^{\text{feas}}} \mathcal{C}'$ since these infeasible sub-domains are empty sets.

We first show that linear programming can be used to effective detect these infeasible sub-domains. For some $\mathcal{C}' \in \mathcal{S}^{\text{infeas}}$, because all the ReLU neurons are fixed to be positive or negative, no relaxation is needed and the network is essentially linear; thus, the value of every hidden neural h_i can be written as a linear equation w.r.t. input x. We add all the Boolean predicates on h_i to a LP problem as linear constraints w.r.t x. If this LP is feasible, then we can find some input x_0 that assigns correct values to all h_i; otherwise, the LP is infeasible.

Due to the lack of feasibility checking in LiRPA, the computed global lower (or upper) bounds from LiRPA is $\underline{f}_{\text{LiRPA}} = \min_{\mathcal{C}' \in \mathcal{S}} \underline{f}_{\mathcal{C}'} = \min\left(\min_{\mathcal{C}' \in \mathcal{S}^{\text{feas}}} \underline{f}_{\mathcal{C}'}, \min_{\mathcal{C}' \in \mathcal{S}^{\text{infeas}}} \underline{f}_{\mathcal{C}'}\right)$. With feasibility checking from LP, we can remove all infeasible sub-domains from this $\min$ such that they do not contribute to the global lower bound: $\underline{f} = \min_{\mathcal{C}' \in \mathcal{S}^{\text{feas}}} \underline{f}_{\mathcal{C}'}$.

To prove the whole BaB verification is complete, it is sufficient to prove this lower bound $\underline{f}$ is the exact minimum of f bounded in $\mathcal{C}$. Since any sub-domain $\mathcal{C}' \in \mathcal{S}^{\text{feas}}$ is a leaf sub-domain with no unstable ReLU neurons, the neural network bounded within $\mathcal{C}'$ is a linear function. LiRPA can give an exact minimum of f within sub-domain $\mathcal{C}'$. Since $\mathcal{C} = \cup_{\mathcal{C}' \in \mathcal{S}^{\text{feas}}} \mathcal{C}'$ (in other words, $\mathcal{S}^{\text{feas}}$ covers all the feasible sub-domains within $\mathcal{C}$), the minimal value for all of them $\underline{f} = \min_{\mathcal{C}' \in \mathcal{S}^{\text{feas}}} \underline{f}_{\mathcal{C}'}$ forms the exact minimum of f within the input domain $\mathcal{C}$. Thus, BaB with LiRPA based bounding procedure is complete when feasibility checking is applied.

B Experimental Setup

We follow the most challenging experimental setup used in the state-of-the-art verifier GNN-ONLINE (Lu & Kumar, 2020) and BABSR (Bunel et al., 2020b). Specifically, we evaluate on CIFAR10 dataset with the same standard robustly trained convolutional neural networks, Base, Wide, and Deep structures, used in (Lu & Kumar, 2020). The Base model contains 2 convolution layers with 8 and 16 filters as well as two linear layers with 100 and 10 hidden units. In total, the Base model has 3,172 ReLU activation units. The Wide model contains 2 convolution layers with 16 and 32 filters and two linear layers with 100 and 10 hidden units, which has 6,244 ReLU activation units in total. The Deep model contains 4 convolution layers and all of them have 8 filters and two linear layers with 100 and 10 hidden units, which has 3,756 ReLU activation units in total.

Given an correctly classified image x with label y_c, and another wrong label $y_{c'} \neq y_c$ (pre-defined in this benchmark) and ϵ, the verifier needs to prove:

$$(\boldsymbol{e}^{(c)} - \boldsymbol{e}^{(c')})^T f(x') > 0 \qquad \text{s.t} \, \forall x' \quad \|x - x'\|_\infty \leq \epsilon \tag{10}$$

where $f(\cdot)$ is a multi-class classifier, $\boldsymbol{e}^{(c)}$ and $\boldsymbol{e}^{(c')}$ are one-hot encoding vectors for labels y_c and $y_{c'}$. We want to verify that for a given ϵ, the trained classifier will not predict wrong label $y_{c'}$ for image x. All properties including x, ϵ, and c' are provided by (Lu & Kumar, 2020). Specifically, they categorize verification properties solved by BABSR within 800s as easy, between 800s and 2400s as medium and more than 2400s as hard.

Our experiments are conducted on one Intel E5 CPU and one Nvidia RTX 1080TI GPU. The parallel batch size n is set to 64 and the η is set to 12,000 due to the memory constraint. To make a fair comparison, we use one CPU core for all methods since the LP based method can only utilize one CPU core. Also, we use one GPU for GNN, GNN-ONLINE, and our methods. In Optimized LiRPA, we apply 200 steps gradient decent at the initial bounds solving. After that during splits, we use 10 steps gradient decent and early stop once $\underline{f} > 0$ or no improvement.

C Ablation study

Our efficient framework leverages two powerful components: (1) Optimized LiRPA and (2) Batch Splits. In this section, we conduct breakdown experiments to show how the individual technique can help with complete verification. As we can see from Table 3, only using Batch Splits cannot help with the problems. We observe that, without Optimized LiRPA and LP in the Batch Splits stage, the LiRPA bounds we obtained are very loose, making the global lower bound improvement slow. In contrast, only using Optimized LiRPA even split single node at one time can still save around 70% computational time benefited from the tighter LiRPA bounds. Finally, combining Batch Splits and Optimized LiRPA allows us to rarely use LP and save up to 96.82% computational time.

Table 3: Ablation study for different parts of our algorithm. The "change (%)" is computed as the improvements on speedup: (time of BABSR - x)/ (time of BaBSR).

	Easy		Medium		Hard		Wide		Deep	
Method	time(s)	change(%)	time(s)	change(%)	time(s)	change(%)	time(s)	change(%)	time(s)	change(%)
BABSR	522.48	–	1335.40	–	3017.11	–	2305.65	–	2455.17	–
+Batch Splits	587.10	-12.37	1470.02	-10.08	3242.57	-7.50	2497.30	-8.31	2835.50	-15.50
+Optimized LiRPA	191.08	+63.43	477.53	+64.24	684.22	+77.32	487.56	+78.85	207.33	+91.56
+Optimized LiRPA & Batch Splits	**56.49**	+89.19	**124.38**	+90.68	**267.35**	+91.14	**181.30**	+92.13	**78.17**	**+96.82**

D COMPLETE VERIFICATION WITH LIRPA ON CPU VS GPU

In this section, we investigate the performance of our algorithm for the cases where one/multiple CPU cores are available without GPU acceleration. Note that existing baselines such as BABSR and MIPPLANET can only utilize one CPU core subject to the Gurobi solver. GNN and GNN-ONLINE need one GPU to forward GNN during branching while the rest of the verification processes all perform on CPU cores. In contrast, our method is much more flexible, we are not limited by the number of CPU cores or GPUs and for a fair comparison, we only use one CPU core and one GPU (same with GNN and GNN-ONLINE) in our paper's experimental results. In Figure 5, results of various computational environments are shown. As we can see, the running time decreases by the number of CPU cores increasing. However, the descent rate seems not constant. It may be caused by the CPU dispatcher or memory cache, and we leave the problem for future work. Also, there is a huge gap between 8 CPU cores with one CPU core + one GPU and the performance gap is more obvious on Wide and Deep models. Surprisingly, even by using one CPU core, we still get very competitive performance even compare with GNN-ONLINE which needs one GPU additionally.

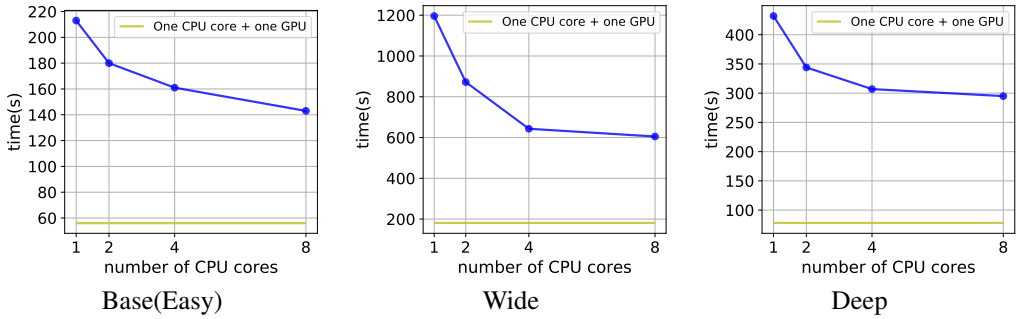


Figure 5: Running time of Base (Easy), Wide, and Deep model when using 1, 2, 4, 8 CPU cores (no GPU) and using 1 CPU core + 1 GPU.