Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond

Kaidi Xu1,*, Zhouxing Shi2,*, Huan Zhang3,*, Yihan Wang2
Kai-Wei Chang3, Minlie Huang4, Bhavya Kailkhura5, Xue Lin1, Cho-Jui Hsieh3

1Northeastern University  2Tsinghua University  3UCLA
4DCST, THUAI, SKLits, BNRist, Tsinghua University  5Lawrence Livermore National Laboratory
{xu.kaid, xue.lin}@northeastern.edu, zhouxingshichn@gmail.com, huan@huan-zhang.com,
wangyihan617@gmail.com, kw@kwchang.net, aihuang@tsinghua.edu.cn,
kailkhura1@llnl.gov, chohsieh@cs.ucla.edu

*Kaidi Xu, Zhouxing Shi and Huan Zhang contributed equally

Abstract

Linear relaxation based perturbation analysis (LiRPA) for neural networks, which computes provable linear bounds of output neurons given a certain amount of input perturbation, has become a core component in robustness verification and certified defense. The majority of LiRPA-based methods focus on simple feedforward networks and need particular manual derivations and implementations when extended to other architectures. In this paper, we develop an automatic framework to enable perturbation analysis on any neural network structures, by generalizing existing LiRPA algorithms such as CROWN to operate on general computational graphs. The flexibility, differentiability and ease of use of our framework allow us to obtain state-of-the-art results on LiRPA based certified defense for fairly complicated networks like DenseNet, ResNeXt and Transformer that are not supported by prior works. Our framework also enables loss fusion, a technique that significantly reduces the computational complexity of LiRPA for certified defense. For the first time, we demonstrate LiRPA based certified defense on Tiny ImageNet and Downscaled ImageNet where previous approaches cannot scale to due to the relatively large number of classes. Our work also yields an open-source library for the community to apply LiRPA to areas beyond adversarial robustness without much LiRPA expertise, e.g., we create a neural network with a provably flat optimization landscape by applying LiRPA to network parameters and considering perturbations on model weights. Our open source library is available at https://github.com/KaidiXu/auto_LiRPA.

1 Introduction

Bounding the range of a neural network outputs given a certain amount of input perturbation has become an important theme for neural network verification and certified adversarial defense [48,31,45,57]. However, computing the exact bounds for output neurons is usually intractable [21]. Recent research studies have developed perturbation analysis bounds that are sound, computationally feasible, and relatively tight [48,54,40,47,38,46]. For a neural network function \( f(x) \in \mathbb{R} \), to study its behaviour at \( x_0 \) with bounded perturbation \( \delta \) such that \( x = x_0 + \delta \in S \) (e.g., \( S \) is a \( \ell_p \) norm ball around \( x_0 \)), these works provide two linear functions \( \ell(x) := a^\top x + b \) and \( \bar{f}(x) := \bar{a}^\top x + \bar{b} \) that are guaranteed lower and upper bounds respectively for output neurons w.r.t. the input under perturbation: \( \ell(x) \leq f(x) \leq \bar{f}(x) \) (\( \forall x \in S \)). We refer to the technique used in these works as Linear Relaxation.
based Perturbation Analysis (LiRPA). CROWN [54] and DeepPoly [40] are two representative LiRPA algorithms. Beyond its usage in neural network verification and certified defense, LiRPA is capable to serve as a general toolbox to understand the behavior of deep neural networks (DNNs) within a predefined input region, and has been used for interpretation and explanation [24, 37].

To compute LiRPA bounds, the first step is to obtain linear relaxations of any non-linear units [54, 36] (e.g., activation functions) in a network. Then, these relaxations need to be “glued” together according to the network structure to obtain the final bounds. Early developments of LiRPA focused on feed-forward networks, and it has been extended to a few more complicated network structures for real-world applications. For example, Wong et al. [50] implemented LiRPA for convolutional ResNet on computer vision tasks; Zügner & Günnemann [59] extended [49] to graph convolutional networks; Ko et al. [24] and Shi et al. [37] extended CROWN [54] to recurrent neural networks and Transformers respectively. Unfortunately, each of these works extends LiRPA with an ad-hoc implementation that only works for specific network architecture. This is similar to the “pre-automatic differentiation” era where researchers have to implement gradient computation by themselves for their designed network structure. Since LiRPA is significantly more complicated than backpropagation, non-experts in neural network verification can find it challenging to understand and use LiRPA for their purpose.

Our paper takes a big leap towards making LiRPA a useful tool for general machine learning audience, by generalizing existing LiRPA algorithms to general computational graphs. Our framework is a superset of many existing works [49, 54, 47, 24, 37], and our automatic perturbation analysis algorithm is analogous to automatic differentiation. Our algorithm can compute LiRPA automatically for a given PyTorch model without manual derivation or implementation for the specific network architecture. Importantly, our LiRPA bounds are differentiable which allows efficient training of these bounds. In addition, our proposed framework enables the following contributions:

- The flexibility and ease-of-use of our framework allow us to easily obtain state-of-the-art certified defense and robustness verification results for fairly complicated networks, such as DenseNet, ResNeXt and Transformer that are hardly supported in existing works due to tremendous efforts required for manual LiRPA implementation.
- We propose loss fusion, a technique that significantly reduces the computational complexity of LiRPA for certified defense. We demonstrate the first LiRPA-based certified defense training on Tiny ImageNet and Downscaled ImageNet [5], with a two-magnitude improvement on training efficiency.
- Our framework allows flexible perturbation specifications beyond $\ell_p$-balls. For example, we demonstrate a dynamic programming approach to concretize linear bounds under discrete perturbation of synonym-based word substitution in a sentiment analysis task.
- We showcase that LiRPA can be a valuable tool beyond adversarial robustness, by demonstrating how to create a neural network with a provably flat optimization landscape and revisit a popular hypothesis on generalization and the flatness of optimization landscape. This is enabled by our unified treatment and automatic derivation of LiRPA bounds for parameter space variables (model weights).

2 Background and Related Work

Giving certified lower and upper bounds for neural networks under input perturbations is the core problem in robustness verification of neural networks. Early works formulated robustness verification for ReLU networks as satisfiability modulo theory (SMT) and integer linear programming (ILP) problems [10, 21, 43], which are hardly feasible even for a MNIST-scale small network. Wong & Kolter [49] proposed to relax the verification problem with linear programming and investigated its dual solution. Many other works have independently discovered similar algorithms [8, 31, 38, 47, 54, 40, 46] in either primal or dual space which we refer to as linear relaxation based perturbation analysis (LiRPA). Recently, Salman et al. [36] unified these algorithms under the framework of convex relaxation. Among them, CROWN [54] and DeepPoly [40] achieve the tightest bound for efficient single neuron linear relaxation and are representative algorithms of LiRPA. Several further refinements for the LiRPA bounding process were also proposed recently, including using an optimizer to choose better linear bounds [7, 29], relaxing multiple neurons [59] or further tighten convex relaxations [42], but these methods typically involve much higher computational costs. The contribution of our work is to extend LiRPA to its most general form, and allow automatic derivation and computation for general network architectures. Additionally, our framework allows a general purpose perturbation analysis for any nodes in the graph and flexible perturbation specifications, not
Our final goal is to compute provable lower and upper bounds for the value of output node $h_o(X)$, i.e., lower bound $h_o^-$, and upper bound $h_o^+$ of $h_o$. For independent nodes, $h_i$ is the vector produced by node $i$. Although $h_i$ can be a tensor in practice, we assume it has been flattened into a vector for simplicity in this paper. Each node $i$ is either an independent node with $m(i) = 0$ representing the input nodes of the graph (e.g., network parameters, model inputs), or a dependent node representing some computations (e.g., ReLU, MatMul). For independent nodes, $H_i$ is an identity function and we denote $h_i = x_i$. We let $X$ be the concatenation of all $x_i$, such that the output of each node $i$ can be written as a function of $X$, $h_i = h_i(X)$, without explicitly referring to $u_j(i)$. Without losing generality, we assume that the computational graph has a single output node $o$. To conduct perturbation analysis, we consider $x_o$ to be arbitrarily taken from an input space $S_o$. In particular, if $x_o$ is not perturbed, $S_o = \{c_o\}$ and $c_o$ is a constant vector. We denote $S$ to be the space of $X$ when each part of $X$, i.e., $x_i$, is perturbed within $S_i$ respectively.

Linear Relaxation based Perturbation Analysis (LiRPA) Our final goal is to compute provable lower and upper bounds for the value of output node $h_o(X)$, i.e., lower bound $h_o^-$, and upper bound $h_o^+$ of $h_o$. For independent nodes, $h_i$ is the vector produced by node $i$. Although $h_i$ can be a tensor in practice, we assume it has been flattened into a vector for simplicity in this paper. Each node $i$ is either an independent node with $m(i) = 0$ representing the input nodes of the graph (e.g., network parameters, model inputs), or a dependent node representing some computations (e.g., ReLU, MatMul). For independent nodes, $H_i$ is an identity function and we denote $h_i = x_i$. We let $X$ be the concatenation of all $x_i$, such that the output of each node $i$ can be written as a function of $X$, $h_i = h_i(X)$, without explicitly referring to $u_j(i)$. Without losing generality, we assume that the computational graph has a single output node $o$. To conduct perturbation analysis, we consider $x_o$ to be arbitrarily taken from an input space $S_o$. In particular, if $x_o$ is not perturbed, $S_o = \{c_o\}$ and $c_o$ is a constant vector. We denote $S$ to be the space of $X$ when each part of $X$, i.e., $x_i$, is perturbed within $S_i$ respectively.

Linear Relaxation based Perturbation Analysis (LiRPA) Our final goal is to compute provable lower and upper bounds for the value of output node $h_o(X)$, i.e., lower bound $h_o^-$, and upper bound $h_o^+$ of $h_o$. For independent nodes, $h_i$ is the vector produced by node $i$. Although $h_i$ can be a tensor in practice, we assume it has been flattened into a vector for simplicity in this paper. Each node $i$ is either an independent node with $m(i) = 0$ representing the input nodes of the graph (e.g., network parameters, model inputs), or a dependent node representing some computations (e.g., ReLU, MatMul). For independent nodes, $H_i$ is an identity function and we denote $h_i = x_i$. We let $X$ be the concatenation of all $x_i$, such that the output of each node $i$ can be written as a function of $X$, $h_i = h_i(X)$, without explicitly referring to $u_j(i)$. Without losing generality, we assume that the computational graph has a single output node $o$. To conduct perturbation analysis, we consider $x_o$ to be arbitrarily taken from an input space $S_o$. In particular, if $x_o$ is not perturbed, $S_o = \{c_o\}$ and $c_o$ is a constant vector. We denote $S$ to be the space of $X$ when each part of $X$, i.e., $x_i$, is perturbed within $S_i$ respectively.
We substitute the \( G_i(\{B_j | j \in u(i)\}) \) into (3) which makes (3) hold true. When node \( i \) is a dependent node, we have a \textit{backward LiRPA oracle function} \( G_i \) which takes \( W_j, b_j, \bar{W}_j, \bar{b}_j \) for every \( j \in u(i) \) as input and produce new linear bounds for node \( i \), assuming all node \( j \in u(i) \) have been bounded:

\[
(W_j, b_j, \bar{W}_j, \bar{b}_j) = G_i(\{B_j | j \in u(i)\}), \text{ where } B_j := (W_j, b_j, \bar{W}_j, \bar{b}_j).
\]

We defer the discussions on oracle function \( G_i \) to a later section. Now, we focus on extending this method on a general graph with known oracle functions in Algorithm 1. The forward mode perturbation analysis is straightforward to extend to a general computational graph: for each dependent node \( i \), we obtain its bounds by recursively applying (2). We check every input node \( j \) and compute the bounds of node \( j \) if they are unavailable. We then use \( G_i \) to obtain the linear bounds of node \( i \). The correctness of this procedure is guaranteed by the property of \( G_i \); given \( B_j \) as inputs, it always produces valid bounds for node \( i \). We analyze its complexity in Appendix A.2.

\begin{algorithm}[h]
\begin{algorithmic}
\Function{BoundForward}{$i$}
\For{$j \in u(i)$}
\If{attributes \( W_j, b_j, \bar{W}_j, \bar{b}_j \) of node \( j \) are unavailable}
\EndFunction
\EndIf
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

\textbf{Forward Mode LiRPA on General Computation Graphs} For each node \( i \) on the graph, we compute the linear bounds of \( h_i(X) \) w.r.t. all the independent nodes:

\[
W_i X + b_i \leq h_i(X) \leq \bar{W}_i X + \bar{b}_i \quad \forall X \in S.
\]

We start from independent nodes. For an independent node \( i \), we have \( h_i(X) = x_i \) so we trivially have the bounds \( Ix_i \leq h_i(X) \leq Ix_i \). For a dependent node \( i \), we have a \textit{forward LiRPA oracle function} \( G_i \) which takes \( W_j, b_j, \bar{W}_j, \bar{b}_j \) for every \( j \in u(i) \) as input and produce new linear bounds for node \( i \), assuming all node \( j \in u(i) \) have been bounded:

\[
(W_j, b_j, \bar{W}_j, \bar{b}_j) = G_i(\{B_j | j \in u(i)\}), \text{ where } B_j := (W_j, b_j, \bar{W}_j, \bar{b}_j).
\]

\textbf{Backward Mode LiRPA on General Computation Graphs} For each node \( i \), we maintain two attributes: \( A_i \) and \( \bar{A}_i \), representing the coefficients in the linear bounds of \( h_i(X) \) w.r.t \( h_i(X) \):

\[
\sum_{i \in \mathcal{V}} A_i h_i(X) + d \leq h_o(X) \leq \sum_{i \in \mathcal{V}} \bar{A}_i h_i(X) + \bar{d} \quad \forall X \in S, \tag{3}
\]

where \( d, \bar{d} \) are bias terms that are maintained in our algorithm. Suppose that the output dimension of node \( i \) is \( s_i \), then the shape of matrices \( A_i \) and \( \bar{A}_i \) is \( s_o \times s_i \). Initially, we trivially have

\[
A_o = \bar{A}_o = I, \quad A_i = \bar{A}_i = 0 (i \neq o), \quad d = \bar{d} = 0. \tag{4}
\]

which makes (3) hold true. When node \( i \) is a dependent node, we have a \textit{backward LiRPA oracle function} \( F_i \), aiming to compute the lower bound of \( A_i h_i(X) \) and the upper bound of \( \bar{A}_i h_i(X) \), and represent the bounds with linear functions of its predecessor nodes \( u_1(i), u_2(i), \ldots, u_{m(i)}(i) \):

\[
(A_{u_1(i)}(i), \bar{A}_{u_1(i)}(i), A_{u_2(i)}(i), \bar{A}_{u_2(i)}(i), \ldots, A_{u_{m(i)}(i)}(i), \bar{A}_{u_{m(i)}(i)}(i), \Delta, \bar{\Delta}) = F_i(A_i, \bar{A}_i),
\]

s.t.

\[
\sum_{j \in u(i)} A_j h_j(X) + \Delta \leq A_i h_i(X), \quad \bar{A}_i h_i(X) \leq \sum_{j \in u(i)} \bar{A}_j h_j(X) + \bar{\Delta}. \tag{5}
\]

We substitute the \( h_j(X) \) terms in (3) with the new bounds (4), and thereby these terms are backward propagated to the predecessor nodes and replaced by the \( h_j(X) (j \in u(i)) \) related terms in (5). In the end, all such terms are propagated to the independent nodes and \( h_o(X) \) will be bounded by linear functions of independent nodes only, where (3) becomes equivalent to (1).
Algorithm 2 Backward Mode Bound Propagation on a General Computational Graph

function BoundBackward(o)
Create BFS queue Q and Q.push(o)
Δ_o ← I, Λ_o ← I, Δ_i ← 0, Λ_i ← 0 (∀i ≠ o), d ← 0, ̄d ← 0 (Eq. 4)
GetOutDegree(o) { ∀i obtain d_i, the number of unprocessed output nodes of node i that o depends on.}
while Q is not empty do
    i ← Q.pop()
    (∆_{i1}, Λ_{i1}), ∆_{i2}, Λ_{i2}, ..., ∆_{i_{d_m(i)}}, Λ_{i_{d_m(i)}}) = F_i(Δ_i, Λ_i) (Eq. 5)
    for j ∈ u(i) do
        Δ_j ← Δ_j + Δ_i, Λ_j ← Λ_j, d_j ← d_j + 1
        if d_j = 0 and node j is a dependent node then
            Q.push(j)
    d ← d + Δ_i, ̄d ← ̄d + Λ_i, Δ_i ← 0, ̄Δ_i ← 0 { Clear Δ_i and Λ_i once we propagated through i.}
return d, ̄d {The algorithm has modified Δ_i, Λ_i on the graph.}

Figure 1: Illustration of the backward mode perturbation analysis. Node 1 ~ 5 are independent nodes and the others are dependent nodes. Red arrows represent the flow of A matrices including both A and ̄A that are propagated from the final output node (node 14) to previous nodes. Finally, only independent nodes retain non-zero A matrices (highlighted in red), and these matrices represent linear bounds w.r.t. independent nodes.

We present the full algorithm in Algorithm 2. We let d_i denote the number of unprocessed output nodes of node i that node o depends on, which is initially obtained by a “GetOutDegree” function detailed in Appendix A.3. We use a BFS for propagating the linear bounds, starting from node o as (4). For each node i picked from the head of the queue, we backward propagate h_i(X) using (5). We update the bound parameters and decrease all d_j(j ∈ u(i)) by one. If d_j = 0 becomes true for a dependent node j, all its related successor nodes have been processed and we push node j to the queue. We repeat this process until the queue is empty. Figure 1 illustrates the flow of backward propagating the bound parameters on an example computational graph, and Figure 2 illustrates the BFS algorithm. We show its soundness in Theorem 1 and its proof is given in Appendix B.1.

Theorem 1 (Soundness of backward mode LiRPA). When Algorithm 2 terminates, we have
\[ \sum_{i \in V} A_i h_i(X) + d \leq h_o(X) \leq \sum_{i \in V} \Lambda_i h_i(X) + \hat{d} \quad \forall X \in S, \]
where \( A_i, \Lambda_i \) are guaranteed to be 0 for any dependent node i, and thus we obtain provable linear upper and lower bounds of node o w.r.t. all independent nodes.

Oracle Functions. Oracle functions \( F_i \) and \( G_i \) are defined for each type of operations. Previous works [48, 53, 56, 57] have covered many common operations such as affine transformations, activation functions, matrix multiplication, etc. Since the major focus of this paper is on handling general computational graph structures, rather than deriving bounds for these elementary operations, we leave the detailed formalization of these oracle functions in Appendix A.1.

1Note that the oracle functions of some operations also require \( h_j, \Lambda_j(j \in u(i)) \) for linear relaxation, although we do not explicitly mention them in the algorithm description for simplicity.
We show two examples: classic $\ell_\epsilon$-ball perturbations where
we allow at most $\delta$ whenever $\|\cdot\|$ is the $p$-norm for each row in the matrix and the result is a vector. The case for $p = 0$ (sparse $\ell_0$ perturbations) is slightly different and will be discussed in Appendix C.4.

### Synonym-based Word Substitution

Beyond $\ell_p$-ball perturbations, we show an example of a perturbation specification defined by synonym-based word substitution in language tasks. Let the clean input to the model be a sequence of words $w_1, w_2, \ldots, w_1$ mapped to embeddings $e(w_1), e(w_2), \ldots, e(w_t)$. Following a common adversarial perturbation setting in NLP [18, 20], we allow at most $\delta$ words to be replaced and each word $w_1$ can be replaced by words within its pre-defined substitution set $S(w_1)$. $S(w_1)$ is constructed from the synonyms of $w_1$ and validated with a language model. We denote each actual input word as $\tilde{w}_1 \in \{w_1\} \cup S(w_1)$, and we show that the linear bounds of node $o$ can be concretized with dynamic programming (DP) in Theorem 2 as proved in Appendix B.2.

**Theorem 2.** Let $\mathbf{W}_d$ be columns in $\mathbf{W}_o$ that correspond to the coefficients of $e(\tilde{w}_t)$ in the linear bounds. The lower bound of $\mathbf{b}_o + \sum_{i=1}^{t} \mathbf{W}_i e(\tilde{w}_i)$, when $j$ words among $\tilde{w}_1, \ldots, \tilde{w}_t$ have been replaced, denoted as $\mathbf{g}_{i,j}$, can be computed by:

$$g_{i,j} = \min( g_{i-1,j} + \mathbf{W}_i e(\tilde{w}_i), g_{i-1,j-1} + \min_{w' \in S(w_1)} \{ \mathbf{W}_i e(w') \}) \quad (i, j > 0) \quad \text{s.t. } w' \in S(w_i),$$

and $g_{0,j} = \mathbf{b}_o + \sum_{i=1}^{t} \mathbf{W}_i e(\tilde{w}_i)$. The concrete lower bound is $\min_{j=0}^{t} g_{0,j}$. The upper bound can also be computed similarly by taking the maximum instead of the minimum in the above DP computation.
3.3 Loss Fusion for Scalable Training of Certifiably Robust Neural Networks

The optimization problem of robust training can be formulated as minimizing the robust loss:

\[
\min_{\theta} \sum_{x \in S} \max_{y \in \mathbb{Y}} L(f_{\theta}(X), y),
\]

where \( f_{\theta}(X) \) is the network output at the logit layer, and \( y \) is the ground truth. Let \( g_{\theta}(X, y) = (e_y 1^T - 1)f_{\theta}(X) \) be the margins between the ground truth label and all the classes (similarly defined in Wong & Kolter [49], Zhang et al. [57]). In previous works, the cross-entropy loss is upper bounded by lower bounds on margins, as a consequence of Theorem 2 in Wong & Kolter [49]:

\[
\max_{x \in S} L(f_{\theta}(X), y) \leq L(-g_{\theta}(X, y), y) \leq \min_{x \in S} g_{\theta}(X, y).
\]

This requires us to first lower bound \( g_{\theta}(X, y) \) using LiRPA. The most efficient LiRPA approach [57] used IBP+Backward to obtain this bound, requiring \( O(Kr) \) time where \( K \) is the output (logit) layer size (equal to the number of classes in this case), and \( O(r) \) is the time complexity of a regular computation pass without computing bounds (see Appendix [A.2]). This cannot scale to large datasets when \( K \) is large (e.g. in Tiny ImageNet \( K = 200 \); in ImageNet \( K = 1000 \)).

We propose a new technique, loss fusion, which computes an upper bound of \( L(f_{\theta}(X), y) \) directly without \( g_{\theta}(X, y) \) as a surrogate. This is possible by treating \( L \) as the output node of the computational graph. When \( L \) is the cross-entropy loss, we have \( L(f_{\theta}(X), y) = \log S(X, y) \), where \( S(X, y) = \sum_{i \in K} \exp(-g_{\theta}(X, y)_i) \). We can thus upper bound \( L(f_{\theta}(X), y) \) by computing an LiRPA upper bound for \( S(X, y) \) directly, yielding \( \overline{S}(X, y) \), and thereby \( \max_{x \in S} L(f_{\theta}(X), y) \leq \log \overline{S}(X, y) \).

This is a novel method that has not appeared in previous works and it yields two benefits. First, this reduces the time complexity of upper bounding \( L(f_{\theta}(X), y) \) to \( O(r) \), as now the output layer size has been reduced from \( K \) to 1. This is the first time in the literature that a tight LiRPA based bound can be computed in the same asymptotic complexity as regular forward propagation and IBP. Second, we show that this is not only faster, but also produces tighter bounds in certain cases:

**Theorem 3.** Given same concrete lower and upper bounds of \( g_{\theta}(X, y) \) as \( g_{\theta}(X, y) \) and \( \overline{g}_{\theta}(X, y) \) which may be used in linear relaxation, for \( S(X, y) = \sum_{i \in K} \exp(-g_{\theta}(X, y)_i) \), we have

\[
\max_{x \in S} L(f_{\theta}(X), y) \leq \log \overline{S}(X, y) \leq L(-g_{\theta}(X, y), y),
\]

where \( L \) is the cross-entropy loss, \( \overline{S}(X, y) \) is the upper bound of \( S(X, y) \) by backward mode LiRPA.

This theorem is proved in Appendix [B.3]. Intuitively, the original approach of propagating \( g_{\theta}(X, y) \) through the cross-entropy loss is similar to using IBP for bounding the loss function, but in loss fusion we treat the loss function as part of the computational graph and apply LiRPA bounds to it directly; it produces tighter bounds as we can use a tighter relaxation for the nonlinear function \( S(X, y) \).

4 Experiments

Table 2: Error rates of different certifiably trained models on CIFAR-10 and Tiny-ImageNet datasets (results on downscaled ImageNet are in Table 3). “Standard”, “PGD” and “verified” rows report the standard test error, test error under PGD attack, and verified test error, respectively.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>67.10</td>
<td>64.31%</td>
<td>67.75%</td>
<td>65.09%</td>
<td>67.23%</td>
<td>64.42%</td>
<td>67.55%</td>
<td>64.16%</td>
<td>66.94%</td>
<td>68.73%</td>
<td>68.73%</td>
<td>72.5%</td>
</tr>
<tr>
<td>Tiny-ImageNet</td>
<td>81.05</td>
<td>80.96%</td>
<td>80.32%</td>
<td>80.52%</td>
<td>79.49%</td>
<td>79.48%</td>
<td>80.17%</td>
<td>79.80%</td>
<td>80.44%</td>
<td>78.94%</td>
<td>78.94%</td>
<td>75.86%</td>
</tr>
</tbody>
</table>

\( \epsilon = \frac{1}{84} \)

Robust Training of Large-scale Vision Models: Our loss fusion technique allows us to scale to Tiny-ImageNet [25] and downscaled ImageNet [51]: to the best of our knowledge, this is the first LiRPA based certified defense on Tiny-ImageNet and downscaled ImageNet with a large number of class labels (200 and 1000, respectively). Besides, the automatic LiRPA bounds allow us to train certifiably robust models on complicated network architectures (WideResNet [59], DenseNet [17] and ResNetX [52]) and

Table 4: Certified defense on Downscaled ImageNet dataset. We use WideResNet in this experiment.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Standard</th>
<th>PGD</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageNet (64 × 64)</td>
<td>IBP Ours</td>
<td>83.77%</td>
<td>89.74%</td>
<td>91.27%</td>
</tr>
</tbody>
</table>
Table 3: Per-epoch training time and memory usage of 4 large models on CIFAR-10 with batch size 256, and 3 large models on Tiny-ImageNet with batch size 100. “LF”=loss fusion; “OOM”=out of memory. Numbers in parentheses are multiples of natural training time or memory usage. With loss fusion, LiRPA based bounds are only 3 to 5 times slower than natural training even on datasets with many labels. Without loss fusion (e.g., in [57]), LiRPA cannot scale to the Tiny-ImageNet dataset.

Table 5: Verification and certified defense for LSTM and Transformer based NLP models. \(\delta_{\text{train}}\) and \(\delta\) represent the number of perturbed words during training and evaluation. For the most important setting \(\delta_{\text{train}}=6\), we run training with different settings and report the mean and standard deviation. \(\delta_{\text{train}}=0\) stands for natural training; \(\delta=0\) stands for evaluating standard test accuracy. “IBP+Backward (alt.)” on \(\delta_{\text{train}}=1\) has an alternative training schedule focusing on the small \(\delta\) (see Appendix [C.2]).

achieve state-of-the-art results, where previous works use simpler models [50, 31] due to implementation difficulty. We extend CROWN-IBP [57] to the general IBP+Backward approach: we use IBP to compute bounds of intermediate nodes of the graph and use tight backward mode LiRPA for the bounds of the last layer. Unlike in CROWN-IBP, we apply loss fusion to avoid the time complexity dependency on the number of class labels, and we train a few state-of-the-art classification models (CNNs) used in the CIFAR-10 and Tiny-ImageNet. With loss fusion, we can enable LiRPA at a cost similar to IBP, allowing us to use much tighter bounds and obtain better-verified errors than IBP (Table 2). The computational cost is significantly better than [57], which is up to 10 (number of labels) times slower than natural training on CIFAR-10, and impossible to scale to Tiny-ImageNet with 200 labels or downsampled ImageNet with 1000 labels. We also report an additional comparison where we use the largest possible batch size rather than a fixed batch size in each setting in Appendix [C.2].

Verifying and Training Robust NLP Models Previous works were only able to implement simple algorithms such as IBP on simple (e.g., CNN and LSTM) NLP models [20] for certified defense. None of them can handle complicated models like Transformer [44] or train with tighter LiRPA bounds. We show that our algorithms can train certifiably robust models for LSTM and Transformer sentiment classifiers on SST-2 [41]. We consider synonym-based word substitution with \(\delta \leq 6\) (up
We then conduct certified defense with \( \theta_{\text{train}} = \{1, 6\} \) using IBP as in \[18\] and our efficient IBP+Backward perturbation analysis. Models trained using IBP+Backward outperforms pure IBP (similar to our observations in computer vision tasks), and the verified test accuracy is significantly better than naturally trained models. The results demonstrate that our framework allows us to better verify and train complex NLP models using tight LiRPA bounds.

**Training Neural Networks with Guaranteed Flatness**

Recently, some researchers \[14\], \[19\], \[13\], \[16\] have hypothesized that DNNs optimized with stochastic gradient descent (SGD) can find wide and flat local minima which may be associated with good generalization performance. Most previous works on LiRPA based certified defense only implemented input perturbations analysis. Our framework naturally extends to perturbation analysis on network parameters \( \theta \) as they are also independent nodes in a computational graph (e.g., node \( x_2 \) in Figure 1). This requires to relax the multiplication operation (e.g., the MatMul nodes in Figure 1) which was first discussed in Shi et al. \[37\], and our Algorithm 2 can then be directly applied. With this advantage, LiRPA can compute provable upper and lower bounds on the local “flatness” around a certain point \( \theta_0 \) for some loss \( \mathcal{L} \):

\[
\mathcal{L}(\theta_0) - C_L(\theta_0) \leq \mathcal{L}(\theta_0 + \Delta \theta) \leq \mathcal{L}(\theta_0) + C_U(\theta_0), \quad \text{for all } ||\Delta \theta||_2 \leq \epsilon, \quad (9)
\]

where \( C_L \) and \( C_U \) are linear functions of \( \theta_0 \) that can be found by LiRPA. This is a “zeroth-order” flatness criterion, where we guarantee that the loss value does not change too much in a small region around \( \theta_0 \), and we do not have further assumptions on gradients or Hessian of the loss. When \( \theta_0 \) is a good solution, \( \mathcal{L}(\theta_0) \) is close to 0, so we can simply set the left hand side of (9) to 0 and upper bound \( \mathcal{L}(\theta_0 + \Delta \theta) \) to ensure flatness. Using our framework, we can train a classifier that guarantees flatness of local optimization landscape, by minimizing the “flat” objective \( \mathcal{L}(\theta_0) + C_U(\theta_0) \) for the perturbation set \( \mathcal{S}(\theta_0) = \{ \theta : ||\theta - \theta_0||_2 \leq \epsilon \} \) where \( \theta_0 \) is the current network parameter. When this “flat” objective is close to 0, we guarantee that \( \mathcal{L} \) is close to 0 for all \( \theta \in \mathcal{S}(\theta_0) \). We build a 3-layer MLP model with \[64, 64, 10\] neurons in each layer and conduct experiments using only 10% and 1% of the training data in MNIST and FashionMNIST, and we then test on the full test set to aggressively evaluate the generalization performance. We also aggressively set the batch size to \{0.01N, 0.1N, N\} as in \[19\] where \( N \) is the size of training dataset. Additional details are in Appendix C.3.

The test accuracies of the models trained with regular cross entropy and our “flat” objective are shown in Figure 3a. We visualize their loss surfaces in Figure 3b. When batch size is increased or fewer data are used, test accuracy generally decreases due to overfitting, which is consistent with \[23\]. For models trained with the flat objective, the accuracy tends to be better, especially when a very large batch size is used. These observations provide some evidence for the hypothesis that a flat local minimum generalizes better, however, we cannot exclude the possibility that the improvements come from side effects of our objective. Our focus is to demonstrate potential applications beyond neural network verification of our framework rather than proving this hypothesis.
Broader Impact

In this paper, we develop an automatic framework to enable perturbation analysis on any neural network structure. Our framework can be used in a wide variety of tasks ranging from robustness verification to certified defense, and potentially many more applications requiring a provable perturbation analysis. It can also play an important building block for several safety-critical ML applications, such as transportation, engineering, and healthcare, etc. We expect that our framework will significantly improve the robustness and reliability of real-world ML systems with theoretical guarantees.

An important product of this paper is an open-source LiRPA library with over 10,000 lines of code, which provides automatic and differentiable perturbation analysis. This library can tremendously facilitate the use of LiRPA for the research community as well as industrial applications, such as verifiable plant control [51]. Our library of LiRPA on general computational graphs can also inspire further improved implementations on automatic outer bounds calculations with provable guarantees.

Although our focus on this paper has been on exploring known perturbations and providing guarantees in such clairvoyant scenarios, in real-world an adversary (or nature) may not adhere to our assumptions. Thus, we may additionally want to understand implication of these unknown scenarios on the system performance. This is a relatively unexplored area in robust machine learning, and we encourage researchers to understand and mitigate the risks arising from unknown perturbations in these contexts.

Acknowledgments and Disclosure of Funding

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was partly supported by the National Science Foundation CNS-1932351, NSFC key project No. 61936010, NSFC regular project No. 61876096, NSF IIS-1901527, NSF IIS-2008173, and ARL-0011469453.

References


