

A Provable Defense for Deep Residual Networks

Working Paper**Author(s):**

Mirman, Matthew; Singh, Gagandeep; Vechev, Martin

Publication date:

2019-03

Permanent link:

<https://doi.org/10.3929/ethz-b-000395401>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)

Originally published in:

arXiv

A Provable Defense for Deep Residual Networks

Matthew Mirman¹ Gagandeep Singh¹ Martin Vechev¹

Abstract

We present a training system, which can provably defend significantly larger neural networks than previously possible, including ResNet-34 and DenseNet-100. Our approach is based on differentiable abstract interpretation and introduces two novel concepts: (i) abstract layers for fine-tuning the precision and scalability of the abstraction, (ii) a flexible domain specific language (DSL) for describing training objectives that combine abstract and concrete losses with arbitrary specifications. Our training method is implemented in the DiffAI system.

1. Introduction

Recent work has shown that neural networks are susceptible to adversarial attacks [Szegedy et al. \(2013\)](#): small, imperceptible perturbations which cause the network to misclassify the input. This has led to growing interest in training procedures to produce robust networks ([Gu & Rigazio, 2014](#); [Zheng et al., 2016](#)), new adversarial attacks ([Papernot et al., 2016](#); [Moosavi-Dezfooli et al., 2017](#); [Xiao et al., 2018](#); [Athalye & Sutskever, 2017](#); [Evtimov et al., 2017](#)), as well as defenses which use these attacks during training ([Goodfellow et al., 2014](#); [Tramèr et al., 2017](#); [Yuan et al., 2017](#); [Huang et al., 2015](#); [Madry et al., 2018](#); [Dong et al., 2018](#)). While networks defended using attacks may be experimentally robust, it has been shown that in general more data is needed ([Schmidt et al., 2018](#)) and that this style of training is sample inefficient ([Khoury & Hadfield-Menell, 2018](#)). Further, while detecting adversarial attacks ([Rozsa et al., 2016](#); [Bhagoji et al., 2017](#); [Feinman et al., 2017](#); [Grosse et al., 2017](#)) appears a promising contingency, [Carlini & Wagner \(2017\)](#) found that many of these techniques were insufficient.

The list of possible attacks is extensive (e.g., ([Akhtar & Mian, 2018](#))) and constantly expanding, motivating the need for methods which can ensure that neural networks are

provably robust against these attacks. [Katz et al. \(2017\)](#) developed a neural network verification system based on SMT solvers, however it only scaled to small networks. [Gehr et al. \(2018\)](#) introduced abstract interpretation ([Cousot & Cousot, 1977](#)) as a method for verifying much larger networks. However, as the size of networks that verification systems could handle increased, it became clear that verifiable robustness could be significantly improved by employing provably robust training. The first attempts for training provably robust networks ([Raghunathan et al., 2018](#); [Kolter & Wong, 2017](#); [Dvijotham et al., 2018](#)) scaled to small sizes with at most two convolutional layers. Later work saw the development of two methods: (i) the dual-method in the case of [Wong et al. \(2018\)](#), and (ii) differentiable abstract interpretation introduced by [Mirman et al. \(2018\)](#) (DiffAI) and used in [Gowal et al. \(2018\)](#) (IBP) and [Wang et al. \(2018\)](#) (MixTrain). While these pushed the boundary in terms of provable verified robustness and network size (with networks of up to 230k neurons), scaling a provable defense to a full ImageNet sized network remains a key challenge. In particular, ResNet-34 represents an important milestone to achieving this goal as it is the smallest residual network proposed by [He et al. \(2016\)](#).

To address this challenge, we introduce a novel approach to robustness, one where the network itself is designed to be provably robust similar to attempts which aim to design networks to be experimentally robust by construction ([Cisse et al., 2017](#); [Sabour et al., 2017](#)). In particular, we introduce the paradigm of “programming to prove”, long known to the programming languages community ([Delahaye, 2000](#); [Wagner et al., 2013](#)), as a technique for creating provably robust architectures. We show how to integrate this idea with DiffAI, resulting in a system that can train a provably robust ResNet-34 (a smaller resnet is shown in Fig. 1).

Main Contributions Our main contributions are:

- The concept of an abstract layer which has no effect on standard network execution but improves provably robust learning.
- A domain specific language (DSL) for specifying sophisticated training objectives.
- A complete implementation and evaluation of our method. Our experimental results indicate the

¹Department of Computer Science, ETH Zurich, Zurich, Switzerland. Correspondence to: Matthew Mirman <matthew.mirman@inf.ethz.ch>.

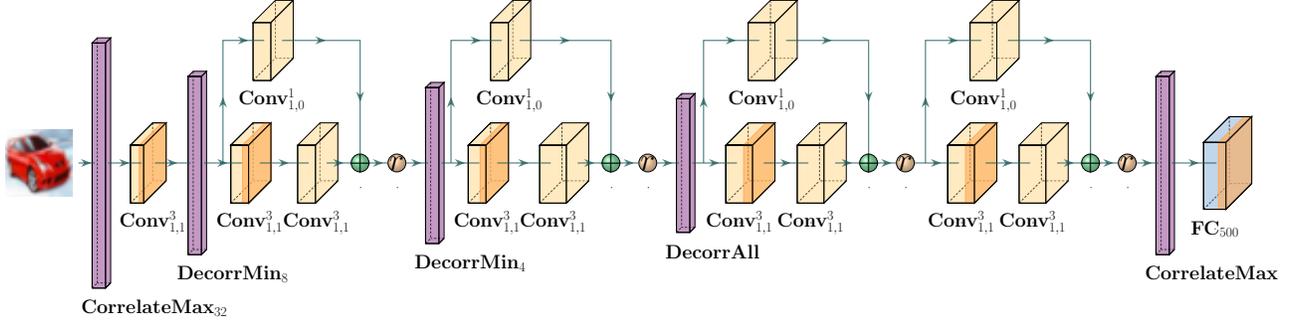


Figure 1. ResNet-Tiny with Abstract Layers. Layers with dark orange on their right include a ReLU, and the sphere with an r in it is also a ReLU. $\text{Conv}_{s,p}^k$ is a convolution with a kernel size of k , a stride of s , and a padding p . This net has 311k neurons and 18m parameters.

approach can achieve provable robustness for networks an order of magnitude larger than prior work.

2. Background on Robust Training

We now provide necessary background on training neural networks to be provably robust against adversarial examples. A neural network $N_\theta: \mathbb{R}^d \rightarrow \mathbb{R}^k$ maps a d -dimensional input to a k -dimensional output based on learned weights θ . Let $B_\epsilon(x)$ be the ℓ_∞ -ball of radius ϵ around an input $x \in \mathbb{R}^d$. A network N_θ is called ϵ -robust around a point $x \in \mathbb{R}^d$ if $\forall \tilde{x} \in B_\epsilon(x), N_\theta(\tilde{x})_i > N_\theta(\tilde{x})_j$ where $i, j \in \{1, \dots, k\}$ and $j \neq i$. The goal of a robust training procedure is to learn a θ such that: (i) N_θ assigns the correct class y_i to each training example x_i , and (ii) N_θ is ϵ -robust around each example x_i .

Differentiable Abstract Interpretation In this work we leverage the differentiable abstract interpretation framework introduced by Mirman et al. (2018). Here, one verifies neural network robustness and formulates provability losses by constructing sound overapproximations using *abstract interpretation* (Cousot & Cousot, 1977). We now introduce the necessary terms used later in the paper.

Definition 2.1. An *abstract domain* \mathcal{D} consists of: (a) abstract elements representing a set of concrete points in $\mathcal{P}(\mathbb{R}^p)$ for $p \in \mathbb{N}$, (b) a *concretization function* $\gamma: \mathcal{D} \rightarrow \mathcal{P}(\mathbb{R}^p)$ mapping an abstract element $d \in \mathcal{D}$ to the set of concrete points it represents, and (c) a set of abstract transformers $T^\#$ approximating the concrete transformer T in $\mathcal{P}(\mathbb{R}^p)$, i.e., $T(\gamma(d)) \subseteq \gamma(T^\#(d))$.

Our approach additionally requires the existence of an abstraction function $\alpha: \mathcal{P}(\mathbb{R}^p) \rightarrow \mathcal{D}$ mapping the set of concrete points in \mathbb{R}^p to an abstract element $d \in \mathcal{D}$. Abstract transformers compose and hence by defining abstract transformers for each basic operation in a neural network N , we can derive an overall abstract transformer $T_N^\#$ for the entire N . We apply abstract interpretation to compute $T_N^\#(\alpha(B_\epsilon(x)))$, describing a superset of the

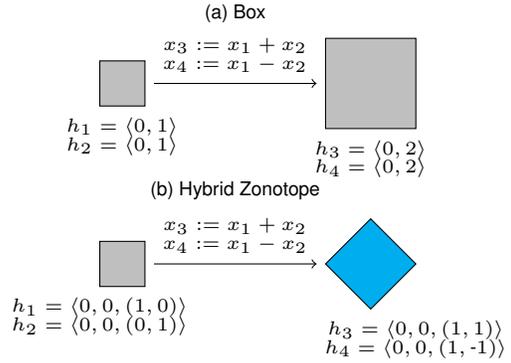


Figure 2. Comparing the precision of the affine transformers in the (a) Box and (b) Hybrid Zonotope domains.

possible outputs of N for all inputs in $B_\epsilon(x)$ which can be used to compute an abstract loss as in Mirman et al. (2018).

Hybrid Zonotope Domain In this work we use the *Hybrid Zonotope Domain* as described by Mirman et al. (2018). This domain, introduced originally by Goubault & Putot (2008), is a generalization of two domains: (i) the simple Box domain (the Box domain is also referred to as interval bound propagation in Gowal et al. (2018)) and, (ii) the base zonotope domain Ghorbal et al. (2009). The main benefit of hybrid zonotopes is that they allow for more fine-grained control of analysis precision and performance.

The Hybrid Zonotope domain associates with every computed result v (e.g., a neuron) in the network, a triplet $h_v = \langle (h_C)_v, (h_B)_v, (h_E)_v \rangle$ where $h = \langle h_C, h_V, h_E \rangle$ and is referred to as the hybrid zonotope over all p variables. Here, $(h_C)_v \in \mathbb{R}$ is a center point, $(h_B)_v \in \mathbb{R}_{\geq 0}$ is a non-negative *uncorrelated error coefficient* (similar to the Box domain), and $(h_E)_v \in \mathbb{R}^m$ are the *correlated error coefficients* the number m of which determine the accuracy of the domain. These coefficients define an affine function \hat{h} which is parameterized by the *correlated error terms* $e \in [-1, 1]^m$ and an *uncorrelated error term*

$\beta \in [-1, 1]^p$:

$$\widehat{h}(\beta, e) = (h_1(\beta, e), \dots, h_p(\beta, e))$$

where:

$$\widehat{h}_v(\beta, e) = (h_C)_v + (h_B)_v \cdot \beta_v + (h_E)_v \cdot e$$

Different variables share the correlated error terms which introduces dependencies between variables making over-approximations more precise than those produced with the Box domain (which does not track dependencies). Formally, the *concretization* function γ_H of a hybrid zonotope h is

$$\gamma_H(h) = \{\widehat{h}(\beta, e) \mid \beta \in [-1, 1]^p, e \in [-1, 1]^m\}.$$

A box b can be expressed as a hybrid zonotope h with $h_C = b_C$ (the box's center), $h_B = b_B$ (the box's radius) and $m = 0$. Descriptions of our hybrid zonotope transformers (e.g., ReLU), can be found in [Mirman et al. \(2018\)](#).

Interval concretization For operations such as constructing an abstract loss or building heuristics in abstract layers, it is necessary to determine the bounds of a hybrid zonotope h for the i -th variable using *interval concretization*:

$$\iota_H(h)_i = [(h_C)_i - \epsilon_H(h)_i, (h_C)_i + \epsilon_H(h)_i]$$

where $\epsilon_H(h)_i = (h_B)_i + \sum_{j=1}^m |(h_E)_{i,j}|$ is the *total error*.

Example: Box vs. Hybrid Zonotope Fig. 2 shows an affine transformation on inputs abstracted in both the Box and the Hybrid Zonotope domains. The box representation in Fig. 2 (a) only contains the center and the uncorrelated error coefficients whereas the hybrid zonotope representation in Fig. 2 (b) also contains non-zero correlated error coefficients. The affine transformation creates dependency between x_3 and x_4 as they are assigned values using affine expressions defined over the same variables x_1 and x_2 . The Box domain cannot capture this and as a result its output is less precise (contains more concrete points) than the one produced with Hybrid Zonotope domain.

3. Abstract Layers for Verifiable Networks

Program verification often involves both the addition of erasable annotations ([Böhme et al., 2008](#)) and program transformations to make the resulting (semantically equivalent program) more suitable for verification ([Wagner et al., 2013](#)). That is, unlike standard transformations which aim to produce a program that runs faster, here, the goal is to produce a more verifiable program. Our key insight is to leverage this “programming to prove” paradigm in a similar fashion when designing robust neural networks.

Based on this insight, we describe the novel concept of *Abstract Layers*. These layers are specifically provided by

the network designer but differ from traditional concrete layers in that they have no effect on the concrete execution. Instead, they only affect the analysis of the network, i.e., they only modify abstract elements that propagate through the layers (e.g., boxes or hybrid zonotopes).

We describe two types of abstract layers designed to tune the precision and scalability of the analysis with the Hybrid Zonotope domain. For all abstract layers, we describe their effect on a given hybrid zonotope h with m correlated error coefficients, producing a new hybrid zonotope h' . For our abstract layers, it holds that $h'_C = h_C$.

3.1. Correlation layers

A correlation layer increases the precision of the analysis in successive layers by producing a new hybrid zonotope h' which contains more correlated error coefficients than the original input h . We note that here we have $\gamma(h') = \gamma(h)$, that is, both hybrid zonotopes actually represent the same set of points. However, the advantage of h' over h is that h' contains more shared dependencies between different dimensions (variables) than h , meaning that successive steps of the analysis using h' will produce more precise results than those same steps using h .

Informally, a correlation layer selects a set of dimension indices (variables) P and creates $|P|$ new correlated error coefficients. For each selected variable $i \in P$, we introduce one correlated error coefficient whose value is that of the variable's uncorrelated error coefficient. All other remaining correlated coefficients (a total of $|P| - 1$) for i are set to 0. For all variables not in P , their new correlated error coefficients are all set to 0. More formally, given h , we define h' as follows:

$$\begin{aligned} h'_{B,i} &= h_{B,i} & i \notin P \\ h'_{B,i} &= 0 & i \in P \\ h'_{E,i,j} &= h_{E,i,j} & \forall 0 \leq i < p, 0 \leq j < m \\ h'_{E,i,m+t} &= h_{B,i} & i \in P \wedge t = |P_{<i}| \\ h'_{E,i,m+t} &= 0 & \forall t < |P|. i \notin P \vee t \neq |P_{<i}| \end{aligned}$$

Here we use $P_{<i}$ to denote the subset of P where each element is smaller than i . Next, we define four variants of a correlation layer based on the choice of the set P .

CorrelateAll correlates all uncorrelated coefficients in all p dimensions thereby adding p correlated error coefficients. Formally, it uses $P = \{i \mid 0 \leq i < p\}$.

CorrelateFixed_k correlates k fixed dimensions, chosen by taking every $\frac{p}{k}$ of the flattened list of dimension indices. Formally, we have $P = \{\lfloor \frac{i \cdot p}{k} \rfloor \mid 0 \leq i < k\}$.

CorrelateMax_k correlates the first k dimensions whose interval concretization (see Section 2) has the largest upper bound value. This heuristic aims to improve precision while still keeping the analysis scalable. Formally, we have

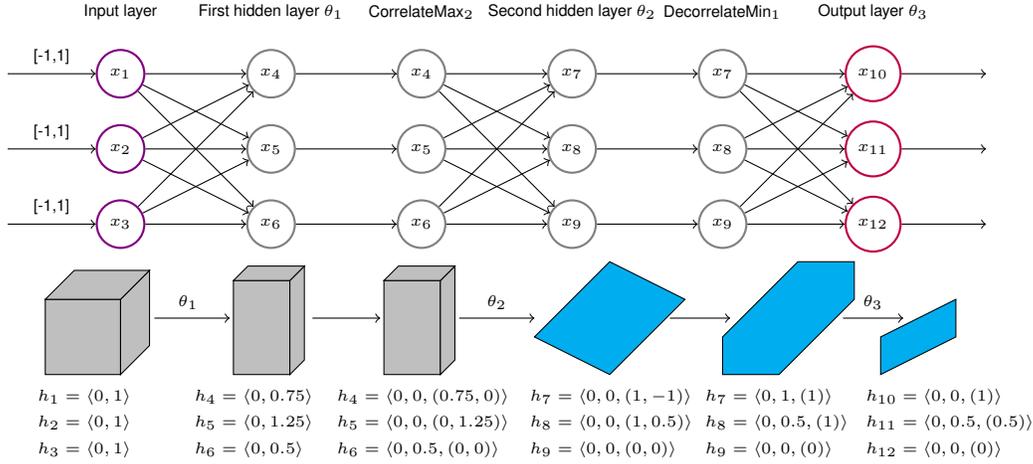


Figure 3. Our analysis on an example toy network augmented with abstract layers.

$P = \{i \mid \text{UB}(\iota_H(h)_i) \in \text{TOP}_k(\text{UB}(\iota_H(h)))\}$ where TOP_k returns the k largest elements and UB returns the upper bound of an interval.

CorrelateMaxPool $_{c,w,h,s}$ correlates dimensions chosen using MaxPooling (Krizhevsky et al., 2012). We apply MaxPooling with kernel size (c, w, h) and stride s on a function f defined over h . If the correlation is applied before the first layer then $f = h_C$ otherwise $f = h_B$. Formally, $P = \{i \mid f(i) \in \text{MAXPOOL}_{c,w,h,s}(f)\}$.

3.2. Decorrelation layers

The purpose of decorrelation layers is opposite that of correlation layers: to reduce the number of correlated coefficients so to make analysis for successive layers more efficient but less precise. Concretely, a decorrelation layer removes correlated error coefficients and adds their absolute sum to the value of uncorrelated error coefficients in each dimension. We now introduce two choices for the set P , each defining different dimensions to be decorrelated:

DecorrelateAll produces a hybrid zonotope with no correlated coefficients in any dimension and in each dimension, the uncorrelated coefficient is defined as:

$$h'_{B,i} = h_{B,i} + \sum_{j=0}^{m-1} |h_{E,i,j}|$$

DecorrelateMin $_k$ is based on a heuristic to minimize the loss of precision due to decorrelation by removing $m - k$ correlated error coefficients whose absolute sum in all dimensions is the smallest. As a result, k correlated coefficients remain in the output. Formally, we define P to be the indices of the $m - k$ smallest elements of the sequence $\{\sum_{i=0}^{p-1} |h_{E,i,j}|\}_{j=0 \dots m-1}$. Then, the new zonotope h' , where $i \in [0, p]$, is defined as:

$$h'_{B,i} = h_{B,i} + \sum_{j \in P} |h_{E,i,j}|$$

$$h'_{E,i,t} = h_{E,i,t} \quad t \notin P \wedge j = t - |P|$$

Informally, in the first equation, we accumulate all correlated coefficients chosen for removal (that is, the set P) into the uncorrelated coefficient, while the second equation ensures the remaining correlated coefficients are shifted to be next to each other (order is preserved).

3.3. DeepLoss

For deeper neural networks such as ResNet-18, it is possible for naive abstraction imprecision to grow exponentially to the point where overflow occurs before the final loss is calculated, making optimizing that loss futile. In such cases, we would like the network to produce more precise results in intermediate layers, before an overflow occurs. As these layers do not have the same number of neurons as target classifications, we cannot optimize using a standard provability loss. Instead, a loss on a generic heuristic for provability must be defined on the output of a specific layer. As this loss does not effect concrete execution and operates using the abstract element from a specific layer, we also consider it a form of an abstract layer. We define the following losses on an interval concretization c in n dimensions:

$$\begin{aligned} L_{\text{lb},f,i}(c) &= \max\{f(c_{j,2} - c_{i,1}) \mid c_{j,1} \leq c_{i,1}\} \\ L_{\text{ub},f,i}(c) &= \max\{f(c_{i,2} - c_{j,1}) \mid c_{i,2} \leq c_{j,2}\} \\ L_{\text{deep},f}(d) &= \frac{1}{2n} \sum_{i=0}^{n-1} (L_{\text{lb},f,i}(\iota(d)) + L_{\text{ub},f,i}(\iota(d))) \end{aligned}$$

where f is a positive activation and $L_{\text{deep},f}$ combines the first two losses for each dimension of an arbitrary abstract element d . Intuitively, this loss measures and sums for each dimension the worst offending overlap between the concretization lower bound in that dimension and the upper

bound of any other dimension, and visa versa.

In our experiments, we used ReLU for f . A naive implementation of the above loss would require n^2 computations (and potentially n^2 space on a GPU), which could be problematic given that the loss is intended to be used on the output of an intermediate, and presumably quite wide, layer. While a matrix multiplication would also typically involve using up to an n^2 sized matrix, this loss is intended to be used between convolutions which typically permits significantly wider outputs through utilizing significantly smaller kernels. We implement it leveraging one dimensional MaxPool and Sort so that marshaling between the CPU and GPU is not required, and such that algorithmic optimizations to MaxPool can be leveraged to potentially¹ provide an $O(n \log n)$ implementation.

3.4. Example network with abstract layers

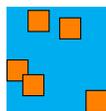
Fig. 3 shows our analysis with abstract layers on an example toy feedforward network. The neural network contains three neurons per layer. We add a CorrelateMax₂ and a DecorrelateMin₁ abstract layers after the first and second hidden layer respectively. We show the 3-dimensional shapes propagated through each layer along with the corresponding hybrid zonotope based encoding. The top, middle, and bottom neurons in each layer represent the x,y, and z-directions in the shapes. Our analysis abstracts the input region with a Box and propagates it through the first hidden layer. After correlations are added, the abstraction is shown in blue (before correlations, the shape is gray).

CorrelateMax₂ changes the encoding of the abstract element obtained after the first hidden layer by creating correlated error coefficients for neurons x_4 and x_5 whose upper bound is larger than that for x_6 . The introduction of correlated error coefficients increases the precision of the result obtained *after* applying the transformers for the second hidden layer as the resulting shape is no longer a box. We note that neuron x_9 is set to 0 in the result.

DecorrelateMin₁ removes the second correlated error coefficient as its absolute sum over all neurons is smaller. The absolute value of this coefficient is added to the uncorrelated coefficient in each dimension. This changes the concretization of the abstract element by removing dependencies making it less imprecise while increasing scalability. The output layer transformations next produce a result more precise than the box obtained without the abstract layers.

¹Provided an optimal implementation of MaxPool

(a) $\delta = 0.2$



(a) $\delta = 0.35$

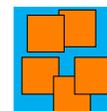


Figure 4. Five two-dimensional abstracts (orange boxes) produced by α^g where g is Sample(δ , Normal, Box)

4. Specifying Training Objectives

We introduce a domain specific language (DSL) for specifying training objectives and parameter scheduling. For example, it can capture the training loss and scheduling proposed by Goyal et al. (2018) (IBP).

4.1. Specifying Schedules

We describe two constructors for describing a schedule used to adjust the values of training parameters (e.g., size of the balls around images used in training) dynamically, leading to improved results. A schedule is a function which uses the current training time step corresponding to the fractional number of epochs completed (e.g., completing 25000 of the 50000 examples from the first epoch on CIFAR10 would provide a time-step value of 0.5). The constructors below describe how to (recursively) build this function.

Lin(a, b, m, n) specifies the parameter value should be the start value a for the first m epochs. Then, linear parameter annealing between start value a and end value b over n epochs should be used to determine the parameter value.

Until(m, s_1, s_2) specifies that the first-schedule constructor s_1 will be used to determine the parameter value until m epochs are reached, and then the second-schedule s_2 will be used but will be given the time with m epochs subtracted.

4.2. Specifying Training Goals

We next describe the goal-constructors for describing how to build the abstraction function and training loss. At timestep s of training a network N on an example o with a target label (t), for each goal constructor (g) in the abstract syntax tree (AST), we build: (i) an abstraction function (α^g) which takes the input box for training specified by the lower (l) and upper bound (u) vectors as input and returns an abstract element $d = \alpha^g(l, u)$, and (ii) a loss function $\text{loss}^g(d, t)$. Before training, the user provides the goal which is parsed into an AST (g_U) and a training width (ϵ). The loss used to train is, for a dataset with values in the range of a to b :

$$\text{loss}^{g_U}(T_N^\#(\alpha^{g_U}(\max(o - \epsilon, a), \min(o + \epsilon, b))), t).$$

Table 1 formalizes our goal constructors, described below:

Point returns the center of the input box specified by l and u for training and uses the cross entropy loss.

Normal returns a point sampled from the normal

Table 1. Our different goal constructors for training. Each assumes the existence of a target label t .

Goal constructor	abstract element $d = \alpha(l, u)$	loss(d, t)
Point	$\frac{l+u}{2}$	cross-entropy(d, t)
Normal	$\text{MAX}(\text{MIN}(0.5 \cdot (u - l) \cdot \text{NORMAL_RAND}(0) + l, l), u)$	cross-entropy(d, t)
Uniform	$\text{MAX}(\text{MIN}(0.5 \cdot (u - l) \cdot (\text{UNIFORM_RAND}(2) - 1) + l, l), u)$	cross-entropy(d, t)
IFGSM $_k$	FGSM(k, l, u, t)	cross-entropy(d, t)
Box	$\langle \frac{l+u}{2}, \frac{l-u}{2}, 0 \rangle$	cross-entropy(d, t)
Mix(g_1, g_2, λ)	$(d_1, d_2) = \alpha^{g_1}(l, u) \times \alpha^{g_2}(l, u)$	$(1 - \lambda) \cdot \text{loss}^{g_1}(d_1, t) + \lambda \cdot \text{loss}^{g_2}(d_2, t)$
Sub(δ, g)	$\alpha^g(0.5 \cdot (u + l - \delta \cdot (u - l)), 0.5 \cdot (u + l + \delta \cdot (u - l)), t)$	loss $^g(d, t)$
Sample(δ, r, g_s, g_t)	$\alpha^{g_t}(b - 0.5 \cdot \delta(u - l), b + 0.5 \cdot \delta(u - l), t)$ where $b = \text{CENTER}(\text{Sub}(1 - r \cdot \delta, g_s))$	loss $^{g_t}(d, t)$
BiSample(g_1, g_2)	$\alpha^{g_2}(l', u', t)$ where $l' = 0.5 \cdot (l + u) - \text{UB}(\alpha^{g_1}(l, u)) - 0.5 \cdot (l + u) $ and $u' = 0.5 \cdot (l + u) + \text{UB}(\alpha^{g_1}(l, u)) - 0.5 \cdot (l + u) $	loss $^{g_2}(d, t)$

Table 2. Training schemes used for the experiments. Those with subscript 18 or L are used by ResNet-18 or ResNetLarge respectively.

Name	Training Scheme
Baseline	Mix(Point, Sub(Lin(0, 1, 150, 10), Box), Lin(0, 0.5, 150, 10))
InSamp	Mix(Point, Sample(Lin(0, 1, 150, 10), 0.5, Normal, Box), Lin(0, 0.5, 150, 10))
InSampLPA	Mix(Point, Sub(Lin(0,1,150,10), Sample(Lin(0,1,150,10), 0.5, Normal, Box)), Lin(0, 0.5, 150, 10))
Adv $_k$ IS	Mix(Sub(Lin(0, 1, 20, 20), IFGSM $_k$), Sample(Lin(0, 1, 150, 10), 0.5, Normal, Box), Lin(0, 0.5, 150, 10))
Adv $_k$ ISLPA	Mix(Sub(Lin(0, 1, 20, 20), IFGSM $_k$), Sub(Lin(0,1,150,10),Sample(Lin(0, 1, 150, 10), 0.5, Normal, Box)), Lin(0, 0.5, 150, 10))
Adv $_k$ ISLPAUS	Mix(Sub(Lin(0, 1, 20, 20), IFGSM $_k$), Sub(Lin(0,1,150,10),Sample(Lin(0, 1, 150, 10), Uniform $_1$, Box)), Lin(0, 0.35, 150, 10))
Baseline $_{S18}$	Mix(Point, Sub(Lin(0, 1, 200, 40), Box), Lin(0, 0.5, 200, 40))
InSamp $_{S18}$	Mix(Point, Sample(Lin(0, 1, 200, 40), 0.5, Normal, Box), Lin(0, 0.5, 200, 40))
Adv $_k$ IS $_{S18}$	Mix(Sub(Lin(0, 1, 20, 20), IFGSM $_k$), Sample(Lin(0, 1, 200, 40), 0.5, Normal, Box), Lin(0, 0.5, 200, 40))
Baseline $_{S18}$	Mix(Point, Sub(Lin(0, 1, 200, 40), Box), Lin(0, 0.5, 200, 40))
InSamp $_{S18}$	Mix(Point, Sample(Lin(0, 1, 200, 40), 0.5, Normal, Box), Lin(0, 0.5, 200, 40))
Adv $_k$ IS $_{S18}$	Mix(Sub(Lin(0, 1, 20, 20), IFGSM $_k$), Sample(Lin(0, 1, 200, 40), 0.5, Normal, Box), Lin(0, 0.5, 200, 40))
Adv $_k$ ISLPA $_{R18}$	Mix(Sub(Lin(0, 1, 20, 20), IFGSM $_k$), Sub(Lin(0,1,200,40),Sample(Lin(0, 1, 200, 40), 1, Uniform, Box)), Lin(0, 0.5, 200, 40))
InSampLPA $_{R34}$	Mix(Point, Sub(Lin(0,1,200,40), Sample(Lin(0, 1, 200, 40), 1, Uniform, Box)), Lin(0, 0.5, 200, 40))
Adv $_k$ ISLPA $_{D100}$	Mix(IFGSM $_k$, Sub(Lin(0,1,150,50),Sample(Lin(0, 1, 150, 50), 1, Uniform, Box)), Lin(0, 0.5, 150, 50))
BiAdv $_L$	Mix(IFGSM $_2$, BiSample(Sub(Lin(0, 1, 150, 30), IFGSM $_3$), Lin(0, 0.6, 200, 30))

distribution around the input box (via the function `NORMAL_RAND`) and clipped to that box. It uses the cross entropy loss.

IFGSM $_k$ uses k iterations of FGSM to find an adversarial example in the input box and uses the corresponding point for training. The cross entropy loss is used.

Box returns a hybrid zonotope abstract element with no correlated error coefficients abstracting the box between the lower (l) and upper-bound (u). The loss concretizes the abstract element d and returns the maximum cross entropy loss on a point in the concretization (Gowal et al., 2018).

Mix(g_1, g_2, λ) takes two goal constructors g_1 and g_2 and a float λ as inputs. The abstract element used for training is the cartesian product of the abstractions d_1 and d_2 of the input box in g_1 and g_2 . The loss linearly combines the loss functions from g_1 and g_2 using λ .

Sub(δ, g) takes a float δ and a goal constructor g as inputs. It computes the abstract element for training by calling the abstraction function α^g of the constructor g using the new bounds $l' = 0.5 \cdot (u + l - \delta \cdot (u - l))$ and $u' = 0.5 \cdot (u + l + \delta \cdot (u - l))$. The insight behind this constructor is to use training elements constructed from boxes that overlap with the input box. The output loss is the loss from g .

Sample(δ, g_s, g_t) uses `Sub(1 - δ, g_s)` to find a point b , by taking the center of the returned training element, and passes $l' = b - 0.5 \cdot \delta(u - l)$ and $u' = b + 0.5 \cdot \delta(u - l)$ to the abstraction function α^{g_t} of g_t . The output loss is from g_t . This is visualized in Fig. 4.

BiSample(g_1, g_2) uses the abstract element $\alpha^{g_1}(l, u, t)$ for the input in g_1 and computes $l' = 0.5 \cdot (l + u) - |(\text{UB}(\alpha^{g_1}(l, u))) - 0.5 \cdot (l + u)|$ and $u' = 0.5 \cdot (l + u) + |(\text{UB}(\alpha^{g_1}(l, u))) - 0.5 \cdot (l + u)|$.

The output element is $\alpha^{g_2}(l', u', t)$. It uses g_2 's loss.

We note that floating point parameters such as δ and λ used in the constructors above can use scheduling constructors.

4.3. Example Training Schemes

Earlier, we observed that our training DSL could be used to specify complex training schemes such as IBP. In particular, IBP uses linear parameter annealing on both the epsilon used in training and the weight of the provability loss, together with a cross entropy based loss function instead of the hinge loss designed by Mirman et al. (2018) (DiffAI). Using this customization, IBP improves on the results of DiffAI while still using the interval domain for training as done by DiffAI. In our DSL, this training scheme could be written as: `Mix(Point, Sub(Lin(0,1,150,10), Box), Lin(0,0.5, 150,10))`.

In Table 2, we show a number of example training schemes captured as expressions in our DSL. We found the following schemes to be particularly useful (these are evaluated next):

InSamp interpolates between training on random points in the L_∞ ϵ -Ball and an abstract box surrounding the example. The idea is that it might be easier to train a network on a point to be ϵ robust if instead of it being only $\epsilon - \mu$ robust already, every point around in the ϵ box around it is also $\epsilon - \mu$ robust for small $\mu > 0$.

InSampLPA is the same as InSamp, but also uses scheduling for the size of the sampling domain, by surrounding it with Sub. The idea is that using the sampling domain is a kind of adversarial training, and it might be easier for the network to learn the standard dataset first.

5. Experimental Setup

Our system, and the code for reproducing experiments, is publicly available at <https://github.com/eth-sri/diffai>. We implemented this system using PyTorch-0.4.1. We ran all experiments using GeForce RTX 2080 Ti GPUs. We do not use weight normalization reparameterization and clipping.

To demonstrate the effectiveness of our technique, we evaluate using the most challenging dataset commonly used for provable verification tasks, CIFAR-10 (Krizhevsky, 2009). We also use the largest commonly used epsilon, $\epsilon = 0.031373 \sim 8/255$. All accuracies and verifiable robustness percentages use the full 10,000 image test set. To augment the dataset and make it easier to learn, random cropping with a padding of 4 was used (this maintains image size) as well as random horizontal flipping.

While IBP and MixTrain presented improvements to robust training, we did not compare against these systems. For IBP, the public code did not contain residual networks though we were able to integrate the proposed training improvements

into DiffAI. For MixTrain, the codebase was unavailable, and we chose not to perform normalization on the dataset prior to usage. Instead, we add a fixed layer to each network in order to make attack and verification epsilons easier to compare across different systems.

For testing the attacked accuracy, we used MI-FGSM (Dong et al., 2018) with $\mu = 0.8$, 20 iterations, and a step size of 0.0031373. To test verifiable robustness, we used DiffAI's built-in Hybrid-Zonotope domain (described earlier).

5.1. Evaluated Networks

A brief overview of the network sizes we evaluate on and their training speed under well performing training schemes, is shown in Table 3. To the best of our knowledge, no other system can train as deep and as large provable networks as our system. In the Appendix, we provide the complete table for all training schemes. Next, we give a brief description of these networks and our training parameters.

ResNet-Tiny is a wide residual network, 12 layers deep, similar to the ResNet described by Wong et al. (2018) but with more and wider layers shown in Fig. 1. It has 50% more neurons than the largest CIFAR10 network trained via IBP or Wang et al. (2018). For this network we always use an initial learning rate of 0.001 with a schedule as used by IBP, and Adam optimization (Kingma & Ba, 2014). We also use an L_2 regularization constant of 0.01.

SkipNet-18 is an 18 layer deep network with 4 residual connections adapted from PyTorch's vision library. For this network we always use an initial learning rate of 0.1 and a schedule where the rate is multiplied by 0.1 at steps 10, 20, 250 and 300. Instead of Adam, standard SGD is used. The L_2 regularization constant is set to 0.0005.

ResNet-18 and ResNet-34 are 18 and 34 layer (respectively) deep residual networks adapted from PyTorch's vision library. For these network we always use an initial learning rate of 0.1 and a similar schedule where the rate is multiplied by 0.1 at steps 10, 20, 250, 300, and 350, and a batch size of 200. We also use SGD here, but do not use any regularization.

DenseNet-100 is a network with 99 layers, and many residual connections, adapted from the models proposed by Huang et al. (2017). To our knowledge, this is the largest network in terms of depth and the number of neurons to have been provably trained so far. For this network, we always use an initial learning rate of 0.1 and a schedule where the rate is multiplied by 0.1 at steps 20, 50, 200, 250, and 300. We also use SGD here, and no regularization, and a batch size of 50. Due to its size, it is only verified using the Box domain and not with the hybrid Zonotope domain.

Table 3. List of networks, and the training schemes that achieved the highest provable robustness.

Net	Neurons	Params	Abstract Layers	Training Scheme	s/epoch	Accuracy %	Verified Robustness %
ResNet-Tiny	312k	18m	ManyFixed	Adv ₁ ISLPAUS	303	40.2	23.2
SkipNet-18	558k	15m	None	Adv ₅ IS _{S18}	260	28.4	21.2
ResNet-Large	639k	66m	LargeCombo	BiAdv _L	527	38.1	3.0
ResNet-18	558k	15m	None	Adv ₅ ISLPA _{R18}	233	32.3	22.3
ResNet-34	967k	25m	None	InSampLPA _{R34}	176	35.1	19.5
DenseNet-100	4.5m	748k	None	Adv ₅ ISLPA _{D100}	727	36.4	21.9

5.2. Abstract Layers

To evaluate the effect of abstract layers, we investigated a variety of configurations for the above networks. For all of our networks, we use CorrelateAll before the last linear layer during both training and testing. This has the effect of not causing any loss of accuracy by that linear layer before the concretization of the loss function.

None means that no additional abstract layers are used.

FewCombo for ResNet-Tiny, has a CorrelateMax₃₂ layer before the first layer, a DecorrelateMin₈ after the first layer, a DecorrelateMin₄ after the first wide residual block, a DecorrelateAll after the second wide residual block, and a CorrelateMax₁₀ before the fully connected layers.

ManyFixed for ResNet-Tiny, has a CorrelateMax₃₂ layer before the first layer, a CorrelateFixed₁₆ then DecorrelateMin₁₆ after the first layer, a CorrelateFixed₈ then DecorrelateMin₈ after the first and second wide blocks, and a CorrelateFixed₄ then DecorrelateMin₄ after the third wide block, and a DecorrelateAll after the fourth.

Combo for SkipNet-18, has pairs of CorrelateFixed_k and DecorrelateMin_[0.5k] with $k = 20, 10, 5$ after layers 3, 4 and 5 respectively and uses DeepLoss after the fourth layer with a weight schedule of Until(90, Lin(0, 0.2, 50, 40), 0).

LargeCombo for ResNet-Large, has a CorrelateFixed₄ then DecorrelateMin₄ before wide residual blocks 1, 2, 3, and 4. Before the wide residual block 5, we place DecorrelateMin₂. It uses DeepLoss after block 2 and 5, with weight schedules of Until(1, 0, Lin(0.5, 0, 50, 3)) and Until(24, Lin(0, 0.1, 20, 4), Lin(0.1, 0, 50)).

A complete description of each network with each abstract layer combination can be found in the Appendix, along with a table showing its performance for every training scheme.

6. Experimental Results

We now demonstrate how our training schemes shown in Table 2 (and discussed earlier) can be used to train provably robust networks of sizes an order of magnitude larger than prior work. We additionally show how abstract layers can be used to further push the envelope of provable robustness.

Table 4. Comparison of abstract layers on ResNet-Tiny.

Layers	s/epoch	Acc%	Attck%	Ver%
None	130	29.4	21.4	17.7
FewCombo	220	29.0	21.9	19.6
ManyFixed	345	28.9	21.4	19.2

Comparing Training Schemes and Abstract Layers To evaluate which combinations of training schemes and abstract layers provide the best results, we first trained ResNet-Tiny using four training schemes both without abstract layers, and with the abstract layer setup described by FewCombo. We trained each for 400 epochs. The complete results are included in the Appendix, here we show the accuracy and verified robustness in Figure 5. We can observe that using abstract layers improves both provable robustness and accuracy when a more complex training scheme is used, and that benefits exist for provable robustness as well. When the objective is only to maximize provable robustness, the training schemes (there are several) which utilize InSamp *and* abstract layers, are optimal. Without abstract layers, inclusion sampling alone appears to have a benefit on accuracy without significant detriment to provable robustness.

To further investigate the effect of abstract layers, we compare the results of training three configurations of abstract layers on ResNet-Tiny. These can be seen in Table 4, which shows the results on the test set after training with Adv₁ISLPA for 350 epochs.

In this experiment, we can see that ManyFixed actually does not perform as well as FewCombo in any metric, while for verified robustness both networks with abstract layers outperform the network without abstract layers.

While ManyFixed contains many more abstract layers and uses more correlation (thus making it significantly slower to train), the layers in FewCombo have been chosen more selectively. ManyFix contains multiple iterations of CorrelateFix_k immediately before DecorrelateMin_k of decreasing size. We hypothesize that placing a CorrelateFix immediately before DecorrelateMin diminishes the utility of DecorrelateMin’s heuristic. As uncorrelated error coefficients tend to accumulate and grow while correlated

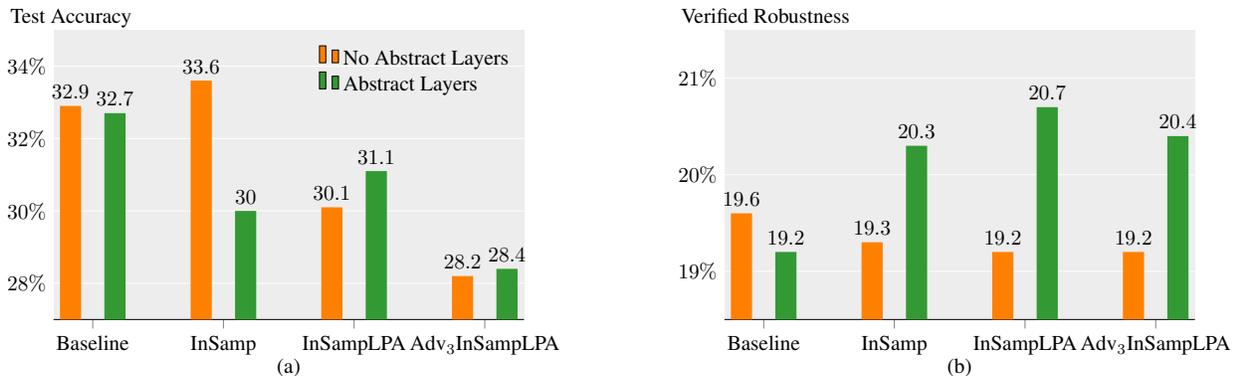


Figure 5. A toy comparison of Accuracy (a) and Verified Robustness (b) of training schemes with similar parameters to Baseline on ResNet-Tiny with and without abstract layers.

Table 5. Comparison of training instances on SkipNet-18.

Scheme	Layers	s/epoch	Acc%	Attck%	Ver%
Baseline ₁₈	None	152	10.2	-	-
InSamp ₁₈	None	102	28.5	23.4	20.5
Adv ₅ IS ₁₈	None	260	28.4	23.8	21.2
InSamp ₁₈	Combo	342	29.5	23	18.5

error terms tend to shrink, a saddle point is generated wherein the network would need to maximize error coefficients (and thus decreasing accuracy) for neurons decided previously to be important (which will become correlated) and minimize error coefficients (and thus increasing accuracy) for neurons that will be decided to be unimportant in order to keep the coefficients from switching.

In summary, FewCombo is more efficient and more accurate than ManyFixed and is a key example for the necessity of the “programming to prove” methodology.

Scaling to SkipNet-18 In order to build a defense scheme capable of training SkipNet-18 we found it necessary to, at a minimum, use InSamp₁₈ training. Table 5 demonstrates the result of training SkipNet-18 with a variety of training schemes for 400 epochs². One can observe that Baseline diverged and while InSamp₁₈ without abstract layers or DeepLoss was able to train a SkipNet-18 model with highest provable robustness, the highest accuracy was obtained using the same training scheme and Combo abstract layers. The training scheme Adv₅IS₁₈ achieved a better compromise between provable robustness and accuracy.

Larger And More Complex Architectures While the majority of our comparisons were performed on ResNet-Tiny and SkipNet-18 we use our schemes to scale to significantly larger networks. To show this, we designed a larger wide residual network, ResNet-Large, with 70k more neurons than SkipNet-18 and 66 million parameters (more

² Baseline was stopped early at 350 epochs as it had clearly failed to train.

than four times as many as SkipNet-18). Here, we found it necessary to use a combination of previously evaluated techniques, in addition to two DeepLoss layers.

For this network we used the BiAdv_L training scheme, which constructs abstract boxes from adversarial attacks. As training this network was significantly more expensive, taking 527 seconds per epoch, we halted training after 100 epochs. The results for this network can be seen in Table 3. While neither the accuracy nor verifiable robustness are particularly competitive with smaller networks, this is the deepest network (by shortest path from input to output) to have proved robust for a competitive epsilon value and that also comes with non-trivial accuracy. While ResNet-34 and DenseNet-100 have longer paths from input to output, they also have very short paths which means that they could potentially learn a small and provably robust network first as an easier sub-problem.

On deeper networks and larger networks that have more residual connections, we found that abstract layers were not as necessary for training. Here, we hypothesize that the network can provably learn the smaller network without the residual layers first, and then use them as possible when they do not too seriously hurt or provability. Table 3 also shows the results of training ResNet-18, ResNet-34, and DenseNet-100. The largest, DenseNet-100 is 4.5 times the number of neurons to appear in any other paper at the time of this publication to have a non-trivial number of points verified to be robust.

7. Conclusion

We introduced a method for training provably robust networks based on the novel concept of abstract layers and a domain specific language for specifying complex training objectives. Our experimental evaluation demonstrates that our approach is effective in training provably robust networks that are an order of magnitude larger than those considered in prior work.

References

- Akhtar, N. and Mian, A. Threat of adversarial attacks on deep learning in computer vision: A survey. *arXiv preprint arXiv:1801.00553*, 2018.
- Athalye, A. and Sutskever, I. Synthesizing robust adversarial examples. *arXiv preprint arXiv:1707.07397*, 2017.
- Bhagoji, A. N., Cullina, D., and Mittal, P. Dimensionality reduction as a defense against evasion attacks on machine learning classifiers. *arXiv preprint*, 2017.
- Böhme, S., Leino, K. R. M., and Wolff, B. Hol-boogie — an interactive prover for the boogie program-verifier. In *Theorem Proving in Higher Order Logics*, pp. 150–166, 2008.
- Carlini, N. and Wagner, D. A. Adversarial examples are not easily detected: Bypassing ten detection methods. *CoRR*, abs/1705.07263, 2017. URL <http://arxiv.org/abs/1705.07263>.
- Cisse, M., Bojanowski, P., Grave, E., Dauphin, Y., and Usunier, N. Parseval networks: Improving robustness to adversarial examples. In *International Conference on Machine Learning*, pp. 854–863, 2017.
- Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, 1977.
- Delahaye, D. A tactic language for the system coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 85–95. Springer, 2000.
- Dong, Y., Liao, F., Pang, T., Su, H., Zhu, J., Hu, X., and Li, J. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9185–9193, 2018.
- Dvijotham, K., Goyal, S., Stanforth, R., Arandjelovic, R., O’Donoghue, B., Uesato, J., and Kohli, P. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018.
- Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., Rahmati, A., and Song, D. Robust physical-world attacks on deep learning models. *arXiv preprint arXiv:1707.08945*, 2017.
- Feinman, R., Curtin, R. R., Shintre, S., and Gardner, A. B. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.
- Gehr, T., Mirman, M., Tsankov, P., Drachler Cohen, D., Vechev, M., and Chaudhuri, S. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *Symposium on Security and Privacy (SP)*, 2018.
- Ghorbal, K., Goubault, E., and Putot, S. The zonotope abstract domain taylor1+. In *International Conference on Computer Aided Verification (CAV)*, 2009.
- Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- Goubault, E. and Putot, S. Perturbed affine arithmetic for invariant computation in numerical program analysis. *arXiv preprint arXiv:0807.2961*, 2008.
- Goyal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Mann, T., and Kohli, P. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.
- Grosse, K., Manoharan, P., Papernot, N., Backes, M., and McDaniel, P. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.
- Gu, S. and Rigazio, L. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014.
- He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
- Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.
- Huang, R., Xu, B., Schuurmans, D., and Szepesvári, C. Learning with a strong adversary. *arXiv preprint arXiv:1511.03034*, 2015.
- Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, 2017.
- Khoury, M. and Hadfield-Menell, D. On the geometry of adversarial examples. *arXiv preprint arXiv:1811.00525*, 2018.
- Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

- Kolter, J. Z. and Wong, E. Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*, 2017.
- Krizhevsky, A. Learning multiple layers of features from tiny images. 2009.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. Towards deep learning models resistant to adversarial attacks. 2018.
- Mirman, M., Gehr, T., and Vechev, M. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning (ICML)*, 2018.
- Moosavi-Dezfooli, S.-M., Fawzi, A., Fawzi, O., and Frossard, P. Universal adversarial perturbations. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 86–94. Ieee, 2017.
- Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pp. 372–387. IEEE, 2016.
- Raghunathan, A., Steinhardt, J., and Liang, P. Certified defenses against adversarial examples. *arXiv preprint arXiv:1801.09344*, 2018.
- Rozsa, A., Rudd, E. M., and Boulton, T. E. Adversarial diversity and hard positive generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 25–32, 2016.
- Sabour, S., Frosst, N., and Hinton, G. E. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pp. 3856–3866, 2017.
- Schmidt, L., Santurkar, S., Tsipras, D., Talwar, K., and Madry, A. Adversarially robust generalization requires more data. *arXiv preprint arXiv:1804.11285*, 2018.
- Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
- Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I., Boneh, D., and McDaniel, P. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.
- Wagner, J., Kuznetsov, V., and Candea, G. -overify: Optimizing programs for fast verification. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, 2013. USENIX.
- Wang, S., Chen, Y., Abdou, A., and Jana, S. Mixtrain: Scalable training of formally robust neural networks. *arXiv preprint arXiv:1811.02625*, 2018.
- Wong, E., Schmidt, F., Metzen, J. H., and Kolter, J. Z. Scaling provable adversarial defenses. *arXiv preprint arXiv:1805.12514*, 2018.
- Xiao, C., Li, B., Zhu, J.-Y., He, W., Liu, M., and Song, D. Generating adversarial examples with adversarial networks. *arXiv preprint arXiv:1801.02610*, 2018.
- Yuan, X., He, P., Zhu, Q., Bhat, R. R., and Li, X. Adversarial examples: Attacks and defenses for deep learning. *arXiv preprint arXiv:1712.07107*, 2017.
- Zheng, S., Song, Y., Leung, T., and Goodfellow, I. Improving the robustness of deep neural networks via stability training. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4480–4488, 2016.

A. Evaluation

Table 6. The networks compared and their sizes and speeds under different training schemes.

Network Name	Neurons	Parameters	Abstract Layers	Training Scheme	Batch Size	Seconds Per Epoch
ResNet-Tiny	311796	18415231	None	Baseline	50	106
				InSamp	50	106
				InSampLPA	50	107
				Adv ₁ ISLPA	50	161
				Adv ₃ ISLPA	50	130
			FewCombo	Baseline	50	209
				InSamp	50	205
				InSampLPA	50	206
				Adv ₁ ISLPA	50	220
				Adv ₃ ISLPA	50	265
ManyFixed	Adv ₁ ISLPA	50	347			
SkipNet-18	558080	15626634	None	Baseline ₁₈	200	152
				InSamp ₁₈	200	102
				Adv ₅ IS ₁₈	200	260
			Combo	InSamp ₁₈	100	342
ResNet-Large	639976	65819474	LargeCombo	BiAdv _L	50	527
ResNet-Large	558k	18m	ResNet-18	Adv ₅ ISLPA _{R18}	200	233
ResNet-Large	967k	25m	ResNet-34	InSampLPA _{R34}	200	176

Table 7. Comparison of different abstract layers and training schemes on ResNet-Tiny.

Train Scheme	Abstract Layers	Seconds Per Epoch	Standard Accuracy %	Attacked Accuracy %	Verified Robust %
Baseline	None	105	32.9	23.7	19.6
	FewCombo	209	32.7	24.1	19.2
InSamp	None	106	33.6	24.7	19.3
	FewCombo	205	30	23.2	20.3
InSampLPA	None	107	30.1	22.5	19.2
	FewCombo	206	31.1	23	20.7
Adv ₃ ISLPA	None	161	28.2	22.2	19.2
	FewCombo	267	28.4	22.5	20.4

B. Further Accuracy Results for MNIST

Table 8. Networks for MNIST.

Network Name	Neurons	Parameters	Depth (ReLUs)
FFNN	500	119910	5
ConvSmall	3604	89606	3
ConvMed	4804	166406	3
ConvBig	48064	1974762	6
ConvLargeIBP	175816	5426402	6
TruncatedVGG	151040	13109706	5

Table 9. MNIST with 0.1

Network	Standard Accuracy	PGD Accuracy	HBox Provability
FFNN	93.3%	90.8%	88.9%
ConvSmall	97.8%	96.2%	95.5%
ConvMed	97.8%	96.3%	95.5%
ConvBig	98.5%	97.2%	95.6%
ConvLargeIBP	98.7%	97.5%	95.8%
TruncatedVGG	98.9%	97.7%	95.6%

Table 10. MNIST with $\epsilon = 0.3$.

Network	Standard Accuracy	PGD Accuracy	HBox Provability
FFNN	80.2%	73.4%	62.6%
ConvSmall	96.9%	93.6%	89.1%
ConvMed	96.6%	93.1%	89.3%
ConvBig	97.0%	95.2%	87.8%
ConvLargeIBP	97.2%	95.4%	88.8%
TruncatedVGG	96.5%	94.4%	87.6%

C. Further Accuracy Results for CIFAR10

Table 11. Networks for CIFAR10.

Network Name	Neurons	Parameters	Depth (ReLUs)
FFNN	500	348710	5
ConvSmall	4852	125318	3
ConvMed	6244	214918	3
ConvBig	62464	2466858	6
ConvLargeIBP	229576	6963554	6
TruncatedVGG	197120	17043018	5

Table 12. CIFAR10 with $\epsilon = 3/255$.

Network	Standard Accuracy	PGD Accuracy	HBox Provability
FFNN	45.1%	37.0%	33.1%
ConvSmall	56.1%	46.2%	42.4%
ConvMed	56.9%	46.6%	43.2%
ConvBig	61.9%	51.4%	45.0%
ConvLargeIBP	61.1%	51.4%	44.5%
TruncatedVGG	62.3%	51.4%	45.5%

Table 13. CIFAR10 with $\epsilon = 8/255$.

Network	Standard Accuracy	PGD Accuracy	HBox Provability
FFNN	33.5%	23.8%	19.0%
ConvSmall	42.6%	30.5%	24.9%
ConvMed	43.6%	30.3%	24.7%
ConvBig	46.0%	34.2%	25.2%
ConvLargeIBP	46.2%	34.7%	27.2%
TruncatedVGG	45.9%	34.4%	27.0%

D. Networks and Abstract Layers

Figure 6. ResNet-Tiny, None

```

Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=16, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
ParSum
ReLU
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
Linear out=500
ReLU
CorrelateAll only_train=False
Linear out=10

```

Figure 7. ResNet-Tiny, FewCombo

```
Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
CorrMaxK only_train=True num_correlate=32
Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
DecorrMin only_train=True k=8 num_to_keep=True
ParNet1
  Conv2D, filters=16, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
ParSum
ReLU
DecorrMin only_train=True k=4 num_to_keep=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
Concretize only_train=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
CorrMaxK only_train=True num_correlate=10
Linear out=500
ReLU
CorrelateAll only_train=False
Linear out=10
```

Figure 8. ResNet-Tiny, ManyFixed

```
Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
CorrMaxK only_train=True num_correlate=32
Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
CorrFix only_train=True k=16
DecorrMin only_train=True k=16 num_to_keep=True
ParNet1
  Conv2D, filters=16, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
ParSum
ReLU
CorrFix only_train=True k=8
DecorrMin only_train=True k=8 num_to_keep=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
CorrFix only_train=True k=8
DecorrMin only_train=True k=8 num_to_keep=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
CorrFix only_train=True k=4
DecorrMin only_train=True k=4 num_to_keep=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
Concretize only_train=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
Linear out=500
ReLU
CorrelateAll only_train=False
Linear out=10
```

Figure 9. SkipNet-18, None

```
Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=128, kernel_size=[1, 1], input_shape=[32, 32, 64], stride=[2, 2], padding=0
ParSum
ReLU
Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ReLU
Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=256, kernel_size=[1, 1], input_shape=[16, 16, 128], stride=[2, 2], padding=0
ParSum
ReLU
Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ReLU
Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=512, kernel_size=[1, 1], input_shape=[8, 8, 256], stride=[2, 2], padding=0
ParSum
ReLU
Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ReLU
Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ReLU
Linear out=512
ReLU
Linear out=512
ReLU
CorrelateAll only_train=False
Linear out=10
```

Figure 10. SkipNet-18, Combo

```

Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=128, kernel_size=[1, 1], input_shape=[32, 32, 64], stride=[2, 2], padding=0
ParSum
ReLU
CorrFix only_train=True k=20
DecorMin only_train=True k=10 num_to_keep=True
Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ReLU
Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ReLU
CorrFix only_train=True k=10
DecorMin only_train=True k=5 num_to_keep=True
DeepLoss only_train=True bw=Until(90, Lin(%s,%s,%s,%s), 0.0) act=<function relu at 0x10b94d620>
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=256, kernel_size=[1, 1], input_shape=[16, 16, 128], stride=[2, 2], padding=0
ParSum
ReLU
CorrFix only_train=True k=5
DecorMin only_train=True k=2 num_to_keep=True
Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ReLU
Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=512, kernel_size=[1, 1], input_shape=[8, 8, 256], stride=[2, 2], padding=0
ParSum
ReLU
Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ReLU
Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ReLU
Linear out=512
ReLU
Linear out=512
ReLU
CorrelateAll only_train=False
Linear out=10

```

Figure 11. ResNet-Large, LargeCombo

```

Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
CorrMaxK only_train=True num_correlate=4
ParNet1
  Conv2D, filters=16, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=16, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
ParSum
ReLU
CorrMaxK only_train=True num_correlate=4
DecorrMin only_train=True k=4 num_to_keep=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 16], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 16], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
CorrMaxK only_train=True num_correlate=4
DecorrMin only_train=True k=4 num_to_keep=True
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
DeepLoss only_train=True bw=Until(1, 0.0, Lin(0.5,0,50,3))
ParNet1
  Conv2D, filters=32, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=32, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
ParSum
ReLU
CorrMaxK only_train=True num_correlate=4
DecorrMin only_train=True k=4 num_to_keep=True
ParNet1
  Conv2D, filters=64, kernel_size=[1, 1], input_shape=[32, 32, 32], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 32], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParSum
ReLU
CorrMaxK only_train=True num_correlate=4
DecorrMin only_train=True k=2 num_to_keep=True
ParNet1
  Conv2D, filters=64, kernel_size=[1, 1], input_shape=[32, 32, 64], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParSum
ReLU
DeepLoss only_train=True bw=Until(24, Lin(0,0.1,20,4), Lin(0.1,0,50,0))
ParNet1
  Conv2D, filters=64, kernel_size=[1, 1], input_shape=[32, 32, 64], stride=[1, 1], padding=0
ParNet2
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParSum
ReLU
Linear out=1000
ReLU
CorrelateAll only_train=False
Linear out=10

```


Figure 12. ResNet-18, None

```
Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=128, kernel_size=[1, 1], input_shape=[32, 32, 64], stride=[2, 2], padding=0
ParSum
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=256, kernel_size=[1, 1], input_shape=[16, 16, 128], stride=[2, 2], padding=0
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=512, kernel_size=[1, 1], input_shape=[8, 8, 256], stride=[2, 2], padding=0
ParSum
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
Linear out=512
ReLU
Linear out=512
ReLU
CorrelateAll only_train=False
Linear out=10
```

Figure 13. ResNet-34, None

```

Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
ReLU
ParNet1
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=64, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[32, 32, 64], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=128, kernel_size=[1, 1], input_shape=[32, 32, 64], stride=[2, 2], padding=0
ParSum
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=128, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[16, 16, 128], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=256, kernel_size=[1, 1], input_shape=[16, 16, 128], stride=[2, 2], padding=0
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU

```

A Provable Defense for Deep Residual Networks

```
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=256, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[8, 8, 256], stride=[2, 2], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
  Conv2D, filters=512, kernel_size=[1, 1], input_shape=[8, 8, 256], stride=[2, 2], padding=0
ParSum
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
ParNet1
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
  ReLU
  Conv2D, filters=512, kernel_size=[3, 3], input_shape=[4, 4, 512], stride=[1, 1], padding=1
ParNet2
ParSum
ReLU
Linear out=512
ReLU
Linear out=512
ReLU
CorrelateAll only_train=False
Linear out=10
```

Figure 14. DenseNet-100, None

```
Normalize mean=[0.4914, 0.4822, 0.4465] std=[0.2023, 0.1994, 0.201]
Conv2D, filters=24, kernel_size=[3, 3], input_shape=[32, 32, 3], stride=[1, 1], padding=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 24], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 36], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 48], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 60], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 72], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 84], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 96], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 108], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 120], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
  Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 132], stride=[1, 1], padding=0
  ReLU
  Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
  ReLU
```

A Provable Defense for Deep Residual Networks

```
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 144], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 156], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 168], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 180], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 192], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[32, 32, 204], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[32, 32, 48], stride=[1, 1], padding=1
SkipCat dim=1
ReLU
Conv2D, filters=108, kernel_size=[1, 1], input_shape=[32, 32, 216], stride=[1, 1], padding=0
AvgPool2D
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 108], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 120], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 132], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 144], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 156], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 168], stride=[1, 1], padding=0
```

A Provable Defense for Deep Residual Networks

```
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 180], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 192], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 204], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 216], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 228], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 240], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 252], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 264], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 276], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[17, 17, 288], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[17, 17, 48], stride=[1, 1], padding=1
SkipCat dim=1
ReLU
Conv2D, filters=150, kernel_size=[1, 1], input_shape=[17, 17, 300], stride=[1, 1], padding=0
AvgPool2D
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 150], stride=[1, 1], padding=0
ReLU
```

A Provable Defense for Deep Residual Networks

```
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 162], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 174], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 186], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 198], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 210], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 222], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 234], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 246], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 258], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 270], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
    ReLU
    Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 282], stride=[1, 1], padding=0
    ReLU
    Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
```

A Provable Defense for Deep Residual Networks

```
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 294], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 306], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 318], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
SkipNet1
SkipNet2
ReLU
Conv2D, filters=48, kernel_size=[1, 1], input_shape=[9, 9, 330], stride=[1, 1], padding=0
ReLU
Conv2D, filters=12, kernel_size=[3, 3], input_shape=[9, 9, 48], stride=[1, 1], padding=1
SkipCat dim=1
ReLU
AvgPool2D
CorrelateAll only_train=False
Linear out=10
```

