Boosting Robustness Verification of Semantic Feature Neighborhoods

Anan Kabaha[1 [0000−0002−0969−6169]] and Dana Drachsler-Cohen[1 [0000−0001−6644−5377]]

Technion, Haifa, Israel {anan.kabaha@campus,ddana@ee}.technion.ac.il

Abstract. Deep neural networks have been shown to be vulnerable to adversarial attacks that perturb inputs based on semantic features. Existing robustness analyzers can reason about semantic feature neighborhoods to increase the networks' reliability. However, despite the significant progress in these techniques, they still struggle to scale to deep networks and large neighborhoods. In this work, we introduce VeeP, an active learning approach that splits the verification process into a series of smaller verification steps, each is submitted to an existing robustness analyzer. The key idea is to build on prior steps to predict the next optimal step. The optimal step is predicted by estimating the robustness analyzer's *velocity* and *sensitivity* via parametric regression. We evaluate VeeP on MNIST, Fashion-MNIST, CIFAR-10 and ImageNet and show that it can analyze neighborhoods of various features: brightness, contrast, hue, saturation, and lightness. We show that, on average, given a 90 minute timeout, VeeP verifies 96% of the maximally certifiable neighborhoods within 29 minutes, while existing splitting approaches verify, on average, 73% of the maximally certifiable neighborhoods within 58 minutes.

1 Introduction

The reliability of deep neural networks (DNNs) has been undermined by adversarial examples: perturbations to inputs that deceive the network. Many adversarial attacks perturb an input image by perturbing each pixel independently by up to a small constant ϵ [14,45,27,36,46]. To understand the local robustness of a DNN in ϵ-balls around given images, many analysis techniques have been proposed [52,12,24,48,34,38,54,16,42,13,47]. In parallel, semantic adversarial attacks have been introduced, such as HSV transformations [21] and colorization and texture attacks [5]. Figure 1 illustrates some of these transformations. Unlike ϵ-ball adversarial attacks which are not visible, feature adversarial attacks can be visible, because the assumption is that humans and networks should not misclassify an image due to perturbations of semantic features. Reasoning about networks' robustness to semantic feature perturbations introduces new challenges to robustness analyzers. The main challenge is that unlike ϵ-ball attacks, where pixels can be perturbed independently, feature attacks impose dependencies on the pixels. Abstracting a feature neighborhood to its smallest bounding ϵ-ball

will lead to too many false alarms. Thus, existing robustness analyzers designed for ϵ-ball neighborhoods perform very poorly on feature neighborhoods.

This gave rise to several works on analyzing the robustness of feature neighborhoods [32,3,42]. These works rely on existing ϵ-ball robustness analyzers and employ two main techniques to reduce the loss of precision. First, they encode the pixels' dependencies imposed by the features by adding layers to the network [32] or by computing a tight linear abstraction of the feature neighborhood [3]. Second, they split the input range into smaller parts, each is verified independently, e.g., using uniform splitting [32,3,42]. Despite of these techniques, for deep networks and large neighborhoods, existing works either lose too much precision and fail to verify or split the neighborhoods into too many parts. In the latter case, approaches must choose between a very long execution time (several hours for deep networks and a single neighborhood) or forcing the analysis to terminate within a certain timeout, leading to certification of neighborhoods that are significantly smaller than the maximal certifiable neighborhoods. These inherent limitations diminish the ability to understand how vulnerable a network is to feature attacks.

Our work: splitting of feature neighborhoods via active learning We address the following problem: given a set of features, each with a target perturbation diameter, find a maximally robust neighborhood defined by these features. We propose a dynamic close-to-optimal input splitting to boost the robustness certification of feature neighborhoods. Unlike previous splitting techniques, which perform uniform splitting [32,3] or branch-and-bound [7,48,6,35,52,30,19], our splitting relies on active learning: the success or failure of previous splits determines the size of future splits. The key idea is to phrase the verification task as a process, where each step picks an unproven part of the neighborhood and submits it to a robustness analyzer. The analyzer either succeeds in proving robustness or fails. Our goal is to compute the optimal split. An optimal split is one where the number of failed steps is minimal, the size of each proven part is maximal, and the execution time is minimal. Predicting an optimal split requires estimating the exact robustness boundary of the neighborhood, which is challenging.

Splitting by predicting the analyzer's velocity and sensitivity We present VeeP (for **v**erification **p**redictor), a learning algorithm, treating the robustness analyzer as the oracle, which dynamically defines the splitting. VeeP defines the next step by predicting the next optimal diameters. To this end, it approximates the analyzer's *sensitivity* and *velocity* for the unproven part. Informally, the sensitivity is a function of the diameters quantifying how certain the robustness analyzer is that the neighborhood is robust. A positive sensitivity means the analyzer determines the neighborhood is robust, while a non-positive sensitivity means the analyzer fails. The velocity is a function of the diameters quantifying the speed of the robustness analyzer. VeeP predicts the diameters of the next step by solving a constrained optimization problem: it looks for the diameters maximizing the velocity such that its sensitivity is positive. VeeP relies on parametric regression to approximate the velocity and sensitivity functions of the current step. It terminates either when it succeeds verifying robustness for the given target

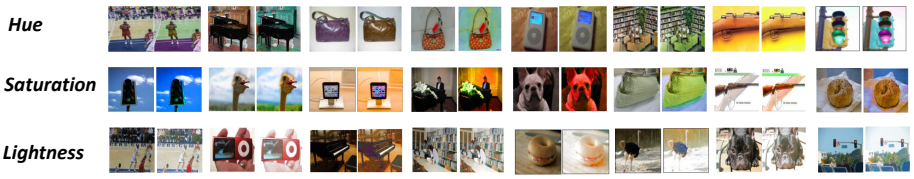

Fig. 1: Examples of ImageNet images and maximally perturbed images in the neighborhoods that VeeP verified robust, for an AlexNet model.

diameters or when it fails to prove robustness for too small parts. It is thus a sound and precise verifier, up to a tunable precision level.

We implemented VeeP in a system, which relies on GPUPoly [34] as the robustness analyzer (the oracle). We evaluate VeeP on different kinds of architectures, including ResNet models for CIFAR-10 and AlexNet models for ImageNet. Our experiments focus on several semantic features: brightness, contrast, and HSL (hue, saturation, lightness). Results show that, when given a 90 minute timeout, VeeP almost perfectly closes the gap between the maximal certified feature neighborhoods and the minimal feature adversarial examples: the verified diameters that VeeP computes are, on average, at least 96% of the maximal certifiable diameter. On average, VeeP completes in 29 minutes. We compare to branch-and-bound, which computes 74% of the maximal diameters in 54 minutes, and to uniform splitting, which computes 73% of the maximal diameters in 62 minutes. We study the acceleration rate of VeeP over branch-and-bound and uniform splitting by running an experiment without a timeout. Results show that VeeP reduces the execution time of branch-and-bound by 4.4x and of uniform splitting by 10.2x. We also compare to the theoretical optimal greedy baseline that "knows" the optimal diameter of every step. We show that VeeP's time overhead is only 1.2x more than this theoretical optimal baseline. Figure 1 illustrates how large the neighborhoods that VeeP verifies. It shows pairs of original ImageNet images and the maximally perturbed image in the neighborhood that VeeP verified robust, for an AlexNet model. In these examples, every neighborhood is defined by a different feature (hue, saturation, and lightness), and the target diameter submitted to VeeP is determined by computing a minimal adversarial feature example along the corresponding feature.

To conclude, our main contributions are:

- A learning algorithm, called VeeP, to verify robustness of feature neighborhoods. VeeP computes an optimal split of the neighborhood, each part is verified by a robustness analyzer. To predict the next split, VeeP approximates the analyzer's velocity and sensitivity using parametric regression.
- An evaluation of VeeP on MNIST, Fashion MNIST, CIFAR-10 and ImageNet over fully-connected, convolutional, ResNet, and AlexNet models. Our evaluation focuses on neighborhoods defined using brightness, contrast, and HSL. Results show that VeeP provides a significant acceleration over branch-and-bound and uniform splitting.

2 Preliminaries

In this section, we provide the background on neural network classifiers, verification of feature neighborhoods, and existing splitting approaches.

Neural network classifiers Given an input domain $\mathbb{R}^d$ and a set of classes $C = \{1, \ldots, c\}$, a classifier is a function mapping inputs to a score vector over the possible classes $D : \mathbb{R}^d \to \mathbb{R}^c$. A fully-connected network consists of L layers. The first layer takes as input a vector from $\mathbb{R}^d$, denoted i, and it passes the input as is to the next layer. The last layer outputs a vector, denoted $o^D(i)$, consisting of a score for each class in C. The classification of the network for input i is the class with the highest score, $c' = \text{argmax}(o^D(i))$. When it is clear from the context, we omit the superscript D. The layers are functions, denoted $h_1, h_2, \ldots, h_L$, each takes as input the output of the preceding layer. The network's function is the composition of the layers: $o(i) = D(i) = h_L(h_{L-1}(\cdots(h_1(i))))$. The function of layer m is defined by a set of processing units called neurons, denoted $n_{m,1}, \ldots, n_{m,k_m}$. Each neuron takes as input the outputs of all neurons in the preceding layer and outputs a real number. The output of the layer m is the vector $(n_{m,1}, \ldots, n_{m,k_m})^T$ consisting of all its neurons' outputs. A neuron $n_{m,k}$ has a weight for each input $w_{m,k,k'}$ and a single bias $b_{m,k}$. Its function is computed by first computing the sum of the bias and the multiplication of every input by its respective weight: $\hat{n}_{m,k} = b_{m,k} + \sum_{k'=1}^{k_{m-1}} w_{m,k',k} \cdot n_{m-1,k'}$. This output is then passed to an activation function φ to produce the output $n_{m,k} = \varphi(\hat{n}_{m,k})$. Activation functions are typically non-linear functions. In this work, we focus on the ReLU activation function, $\text{ReLU}(x) = \max(0, x)$. We note that, for simplicity's sake, we explain our approach for fully-connected networks, but it extends to other architectures, e.g., convolutional and residual networks.

Local robustness A safety property for neural networks that has drawn a lot of interest is *local robustness*. Its meaning is that a network does not change its classification for a given input under a given type of perturbation. Formally, given an input x, a neighborhood containing x, $I(x) \subseteq \mathbb{R}^d$, and a classifier D, we say D is robust in $I(x)$ if $\forall x' \in I(x)$, $\text{argmax}(D(x')) = \text{argmax}(D(x))$. We focus on feature neighborhoods, consisting of perturbations of an input x along a set of features $f_1, \ldots, f_T$. The perturbation of an input along a feature f is a function $f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$, mapping an input x and a diameter δ to the perturbation of x along the feature f by δ. To abbreviate, we call the perturbation function the feature f, similarly to [32]. For all features f and inputs x, we assume $f(x, 0) = x$. Given a feature f, a diameter $\bar{\delta}$, and an input x, the feature neighborhood $I_{f,\bar{\delta}}(x)$ is the set of all perturbations of x along f by up to diameter $\bar{\delta}$: $I_{f,\bar{\delta}}(x) = \{f(x, \delta) \mid 0 \leq \delta \leq \bar{\delta}\}$. We extend this definition to a set of features by considering a diameter for every feature. Given a set of features $f_1, \ldots, f_T$, their diameters $\bar{\delta}_1, ..., \bar{\delta}_T$, and an input x, we define:

$$I_{f_1,\bar{\delta}_1,\ldots,f_T,\bar{\delta}_T}(x) = \{f_T(...f_2(f_1(x, \delta_1), \delta_2)..., \delta_T) \mid 0 \leq \delta_1 \leq \bar{\delta}_1, ..., 0 \leq \delta_T \leq \bar{\delta}_T\}$$

3 Verification of Feature Neighborhoods: Motivation

There are many verifiers for analyzing robustness of neural networks [52,12,24,48,34,38,54,16,42,13,47]. Most of them analyze box neighborhoods, where each input entry is bounded by an interval $[l, u]$ (for $l, u \in \mathbb{R}$). In particular, they can technically reason about feature neighborhoods: first, one has to over-approximate a feature neighborhood $I_{f_1,\bar{\delta}_1,\ldots,f_T,\bar{\delta}_t}(x)$ to a bounding box neighborhood, and then pass the box neighborhood to any of these verifiers. However, this approach loses the dependency between the input entries, imposed by the features, and may result in spurious counterexamples. To capture the dependencies, a recent work proposes to encode features as a layer and add it to the network as the first layer [32]. This has been shown to be effective for various features, such as brightness, hue, saturation, and lightness. However, for deep networks and large feature neighborhoods, encoding the dependency is not enough to prove robustness: either the analysis time is too long or the analyzer loses too much precision and fails. Because feature neighborhoods have low dimensionality (every feature introduces a single dimension), divide-and-conquer is a natural choice for scaling the analysis [32,3,42].

Divide-and-conquer for feature neighborhoods Divide-and-conquer is highly effective for scaling the analysis of feature neighborhoods. The key challenge is computing a useful split. A branch-and-bound approach (BaB) computes the split lazily [7,48,6,35,52,30,19]. To illustrate, consider a single feature neighborhood $I_{f,\bar{\delta}}(x)$. A BaB approach begins by analyzing $I_{f,\bar{\delta}}(x)$. If the analysis fails, it splits the neighborhood into two neighborhoods, $I_{f,\delta}(x)$ and $I_{f,\bar{\delta}-\delta}(f(x,\delta))$. Then, it analyzes each neighborhood separately and continues to split neighborhoods upon failures. As a result, it tends to waste a lot of time on analyzing too large neighborhoods until reaching to suitable-sized neighborhoods. A uniform splitting approach determines a number m and splits the neighborhood into $I_{f,\bar{\delta}/m}(x), \ldots, I_{f,\bar{\delta}/m}(f(x, \bar{\delta} \cdot (m-1)/m))$ [32,3,42]. This approach may still fail for some neighborhoods, due to timeouts or loss in precision, or waste too much time on verifying too small neighborhoods. This raises the question: *can we dynamically determine a split that minimizes the execution time of the verification?*

4 Problem Definition: Time-Optimal Feature Verification

In this section, we define the problem of robustness verification of feature neighborhoods minimizing the execution time. To simplify notation, the definitions assume a single feature, but they easily extend to multiple features.

We view the robustness analysis of feature neighborhoods as a process. Given a feature neighborhood, the verifier executes a series of steps, dynamically constructed, until reaching the maximal diameter for which the network is robust. Our verification process relies on a box analyzer $\mathcal{A}$, which can determine the robustness of box neighborhoods. Every verification step determines the next (sub)neighborhood to verify and invokes the analyzer. The analyzer $\mathcal{A}$ need

not be complete and may fail due to overapproximation error. That is, given a network and a box neighborhood, $\mathcal{A}$ returns *robust*, *non-robust*, or *unknown*. Since the goal of the feature verifier is to compute a maximal neighborhood, if $\mathcal{A}$ returns *unknown*, it splits the last neighborhood into smaller neighborhoods. To guarantee that the verification process terminates, if $\mathcal{A}$ fails to verify a feature neighborhood with a diameter up to a predetermined threshold δ_{MIN}, we assume that this neighborhood is not robust. Because the feature verifier terminates when reaching the maximal diameter, the challenge is not to improve its precision but rather to keep its execution time minimal. We next provide formal definitions.

Definition 1 (Verification Step). *Given a box analyzer $\mathcal{A}$, a classifier D, and a feature neighborhood defined by f, $\bar{\delta}$ and x, a verification step is a pair (δ_x, δ), such that $0 \leq \delta_x < \bar{\delta}$ and $0 < \delta \leq \bar{\delta}$. The result of a verification step (δ_x, δ) is $\mathcal{A}$'s result for D and $I_{f,\delta}(f(x, \delta_x))$, which is* robust, not robust *or* unknown.

We next define feature verification sequence, consisting of verification steps.

Definition 2 (Feature Verification Sequence). *Given a box analyzer $\mathcal{A}$, a precision level δ_{MIN}, a classifier D, and a feature neighborhood defined by f, $\bar{\delta}$, and x, a feature verification sequence is a sequence of verification steps $s_1, \ldots, s_m$ that verify the maximally robust neighborhood up to $\bar{\delta}$, i.e., either:*

- *there is no step whose result is* not robust *and, for every $\delta_y \in [0, \bar{\delta}]$, there is a step $s = (\delta_x, \delta)$, where $\delta_x \leq \delta_y \leq \delta_x + \delta$, for which $\mathcal{A}$ returns* robust. *That is, the verification steps cover all inputs in $I_{f,\bar{\delta}}(x)$, or*
- *there is no step whose result is* not robust*, except perhaps the last step $s_m = (\delta_{m,x}, \delta_m)$ whose result is* unknown *or* not robust *and $\delta_m = \delta_{MIN}$. For every $\delta_y \in [0, \delta_{m,x}]$, there is a step $s = (\delta_x, \delta)$, where $\delta_x \leq \delta_y \leq \delta_x + \delta$, for which $\mathcal{A}$ returns* robust. *That is, the verification steps cover all inputs in $I_{f,\delta_{m,x}}(x)$ and we assume there is an adversarial example in $I_{f,\delta_{MIN}}(f(x, \delta_{m,x}))$.*

Finally, we define the problem of time-optimal feature verification. To this end, we introduce a notation. Given a verification step s, we denote by $t(s)$ the execution time of the analyzer $\mathcal{A}$ on the neighborhood defined by step s. We note that we assume that the time to define a verification step $s = (\delta_x, \delta)$ is negligible with respect to $t(s)$. Given a feature verification sequence $S = (s_1, \ldots, s_m)$, its execution time is the sum of its steps' execution times: $t(S) = \Sigma_{i=1}^{m} t(s_i)$. Our goal is to compute a feature verification sequence minimizing the execution time.

Definition 3 (Time-Optimal Feature Verification). *Given a box analyzer $\mathcal{A}$ and a feature neighborhood defined by f, $\bar{\delta}$ and x, a time-optimal feature verification sequence S is one that minimizes the execution time: $argmin_S t(S)$.*

This problem is challenging because divide-and-conquer algorithms have the execution time of a verification step only *after* they invoke $\mathcal{A}$ on that step's neighborhood. Thus, constructing a verification sequence is bound to involve suboptimal choices. However, we show that it is possible to *predict* the execution time of a (new) verification step based on the execution times of the previous steps. We note that although we focus on analysis of deep neural networks, we believe that predicting verification steps based on prior steps is a more general concept which is applicable to analysis of other machine learning models.

5 Prediction by Proof Velocity and Sensitivity

In this section, we present the key concepts on which we build to predict the verification steps: proof velocity and sensitivity. We show that these can be modeled by parametric functions. We then explain how these functions can be used to predict optimal steps by solving a constrained optimization problem.

Proof velocity To minimize the execution time of the verification process, we wish to maximize the *proof velocity*. Proof velocity is the ratio of the neighborhood's *certified diameter* and the time to verify it by the box analyzer $\mathcal{A}$. In the following, we denote the execution time of step $s = (\delta_x, \delta)$ by $t(s) = t_{\mathcal{A}}(I_{f,\delta}(f(x, \delta_x)))$. The certified diameter of this step's neighborhood, denoted $\delta^s_{\mathcal{A}}$, is equal to δ, if $\mathcal{A}$ returns *robust*, and 0, if $\mathcal{A}$ returns *non-robust* or *unknown*.

Definition 4 (Proof Velocity). *Given a box analyzer $\mathcal{A}$, a classifier D, a feature neighborhood defined by f, $\bar{\delta}$, and x, and a verification step $s = (\delta_x, \delta)$, the proof velocity of s is:* $V_{\mathcal{A}}(I_{f,\delta}(f(x, \delta_x))) = \frac{\delta^s_{\mathcal{A}}}{t(I_{f,\delta}(f(x,\delta_x)))}$.

The velocity is either a positive number, if $\mathcal{A}$ returns *robust*, and 0 otherwise. A zero velocity means that the feature verifier has to split this neighborhood and that we have not gained from this analysis. Empirically, we observe that if $\mathcal{A}$ relies on linear approximations to analyze the network robustness, the proof velocity can be modeled as a function of the certified diameter. For small networks or neighborhoods, the velocity is approximately a linear function of the diameter, because the analysis time is, in practice, constant. The larger the network or the neighborhood, the longer the analysis time because the overapproximation error increases, and thus the analyzer $\mathcal{A}$ executes more refinement steps (e.g., back-substitution [42] or solving linear programs [48]). We empirically observe that when the network or the neighborhood are large enough to trigger refinement steps, the execution time is approximately exponentially related to the diameter: $t(\delta) \propto exp(\beta \cdot \delta)$, for some parameter β. Consequently, $V(\delta) \propto \delta \cdot exp(-\beta \cdot \delta)$. Note that, for $\beta = 0$, the proof velocity is linear in δ. Thus, this function captures both cases of small network/neighborhood and large network/neighborhood. We illustrate this relation in Figure 2, showing the measured proof velocity (the blue dots) as a function of the diameter δ, across different models and three box analyzers relying on different linear approximations. The figure also shows the function we use to approximate the proof velocity (the red curve). The figure shows how close the approximation is. We next summarize this observation.

Observation 1. *For every verification step $s = (\delta_x, \delta)$, if $\delta^s_{\mathcal{A}} > 0$, the velocity can be approximated by: $V(\delta) = \alpha_V \cdot \delta \cdot exp(-\beta_V \cdot \delta)$ for $\beta_V \geq 0$ and $\alpha_V \in \mathbb{R}$.*

We can use this observation to predict time-optimal verification steps. To this end, at the beginning of every verification step, we require to (1) estimate the parameters of the velocity's function and (2) predict the maximal δ^s_{MAX} for which the analyzer $\mathcal{A}$ returns *robust*. With these values, we can define the next step by computing $\delta \in (0, \delta^s_{\text{MAX}}]$ maximizing the proof velocity. In order to predict the maximal value δ^s_{MAX}, we define the concept of *neighborhood sensitivity*.

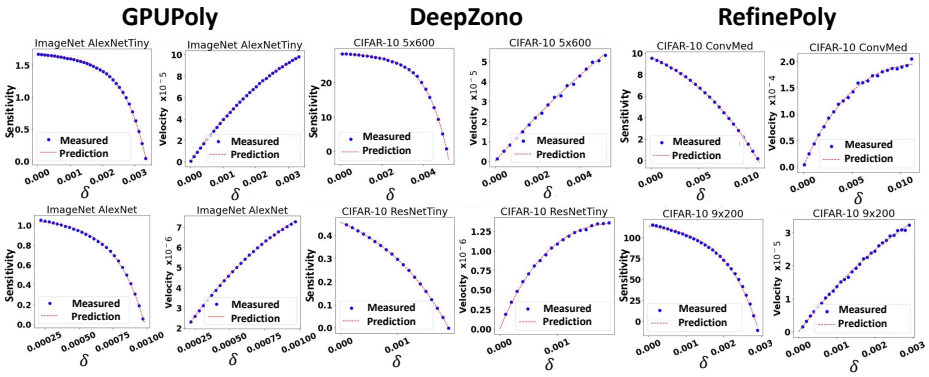

Fig. 2: Velocity and sensitivity as functions of the diameter δ, for different models and three box analyzers: GPUPoly [42], DeepZono [41], and RefinePoly [40]. Blue dots show the measured values and red curves show our function approximations.

Neighborhood sensitivity The *sensitivity* concept builds on the commonly known concept *network confidence*. Given a classifier D and an input x, the confidence of the classifier in class j is the output $o_j^D(x)$, i.e., the score that D assigns for j on input x. Based on this term, we define the *sensitivity* of x as the difference between the confidence in j and the highest confidence in a class different from j:

$$S^D(x,j) = o_j^D(x) - \text{argmax}_{j' \neq j}(o_{j'}^D(x))$$

If $S^D(x,j) > 0$, then D classifies x as j, and the higher $S^D(x,j)$ the more certain the classifier is in its classification of x as j. We extend this term to neighborhoods. We define the neighborhood sensitivity as the minimal sensitivity of its inputs: $S^D(I,j) = \min\{S^D(x',j) \mid x' \in I\}$. From this definition, we get few observations. First, for any $I \subseteq I'$, we have $S^D(I',j) \leq S^D(I,j)$. That is, extending a neighborhood with more inputs may decrease the neighborhood sensitivity in j. Second, if $S^D(I,j) \leq 0$, then I is not robust to j. Third, if $\mathcal{A}$ is precise, then for every verification step $s = (\delta_x, \delta)$, we have $\delta_{\mathcal{A}}^s = \delta$ if and only if the sensitivity $S^D(I_{f,\delta}(f(x,\delta_x)),j)$ is positive. In practice, we rely on an imprecise analyzer $\mathcal{A}$ and we cannot compute the exact neighborhood sensitivity. However, we can approximate a neighborhood's sensitivity by relying on the analysis of $\mathcal{A}$. Since most incomplete analyzers compute, for every output neuron k, real-valued bounds $[l_k, u_k]$, we can approximate the neighborhood sensitivity:

$$S_{\mathcal{A}}^D(I_{f,\delta}(f(x,\delta_x)),j) = l_j - \max_{j' \neq j} u_{j'}$$

Thus, to compute the maximal δ_{MAX}^s whose neighborhood can be proven robust by $\mathcal{A}$, we can compute the maximal δ_{MAX}^s for which $S_{\mathcal{A}}^D(I_{f,\delta_{\text{MAX}}^s}(f(x,\delta_x)),j) > 0$. The remaining question is how to approximate the sensitivity function. Empirically, we observe that if $\mathcal{A}$ relies on linear approximations to analyze the network robustness, the neighborhood sensitivity has an exponential relation to the diameter. This

is demonstrated in Figure 2, for different models and linear approximations. The figure shows how close the approximation is (red curves) to the measured sensitivity (the blue dots). We next summarize this observation.

Observation 2. *For every verification step, the neighborhood sensitivity can be approximated by:* $S_{\mathcal{A}}(\delta) = \alpha_S + \beta_S \cdot exp(\gamma_S \cdot \delta)$*, where* $\alpha_S, \beta_S, \gamma_S \in \mathbb{R}$*.*

This exponential relation can be explained by considering the effect of linear approximations on non-linear computations. At a high-level, the exponential relation is linked to the number of non-linear neurons being approximated. We exemplify this relation in the extended version of this paper [23, Appendix A].

Time-optimal feature verification via proof velocity and sensitivity Given the functions of the velocity and sensitivity, we can state our problem as a constrained optimization. Given an analyzer $\mathcal{A}$, a feature neighborhood defined by f, $\bar{\delta}$ and x, and the currently maximal certified diameter δ_x, the δ of the optimal verification step $s = (\delta_x, \delta)$ is a solution to the following optimization problem:

$$\max V^D(I_{f,\delta}(f(x,\delta_x))) \text{ such that } S^D_{\mathcal{A}}(I_{f,\delta}(f(x,\delta_x)), c_x) > 0$$

Here, c_x is the classification of x. Because both functions are convex, the global maximum can be computed as standard. First, we compute the feasible region of δ by comparing $S^D(I_{f,\delta}(f(x,\delta_x)), c_x)$ to zero. Second, we compute the derivative of $V^D(I_{f,\delta}(f(x,\delta_x)))$, compare to zero, and compute the optimal δ. If the optimal δ is not feasible, we take the closest feasible value. Therefore, if we know the parameters of the velocity and sensitivity functions, we can compute an optimal verification step. The challenge is to approximate these parameters, for every step. In the next section, we explain how to predict them from the previous steps.

6 VeeP: A System for Time-Optimal Feature Verification

In this section, we present our system, called VeeP, for computing time-optimal verification steps. VeeP builds on the ideas presented in Section 5 and dynamically constructs the verification steps by solving the constrained optimization problem. The challenge is predicting the parameters of the velocity and sensitivity functions. The key idea is to treat the analyzer as an *oracle*, whose responses to previous verification steps are used to define the next step. Conceptually, VeeP builds on active learning, where it acts as the learner for optimal verification steps and the analyzer acts as the oracle. Throughout execution, VeeP tracks the accumulated verified diameters of the robust neighborhood. If a verification step succeeds, the robust neighborhood is extended and the verified diameters increase. If a step fails, the next predicted diameters will be smaller, up to a minimal value δ_{MIN}. Thus, although VeeP predicts the diameters, which may be too small or large, its overall analysis is sound and precise up to δ_{MIN}. It is sound because it employs divide-and-conquer and relies on a sound analyzer. It is precise because if a step fails for diameters greater than δ_{MIN}, then VeeP attempts again to extend the

robust neighborhood by predicting smaller diameters. We begin this section by explaining how VeeP reasons about neighborhoods defined by a single feature and then extend it to general feature neighborhoods.

6.1 VeeP for Single Feature Neighborhoods

In this section, we describe VeeP for analyzing neighborhoods defined by a single feature. VeeP takes as inputs a classifier D, a feature f, a diameter $\bar{\delta}$, and an input x. During its execution, it maintains in δ_x the sum of the certified diameters. It returns the maximal $\delta_x \leq \bar{\delta}$ for which the neighborhood is robust, up to precision δ_{MIN}. VeeP operates iteratively, where the main computation of every iteration is determining a verification step $s_k = (\delta_x, \delta_k)$ to submit to the analyzer $\mathcal{A}$.

Defining a verification step The goal of a verification step is to increase the accumulated certified diameter δ_x by a diameter δ_k. VeeP aims at choosing δ_k such that (1) the sensitivity of $I_{f,\delta_k}(f(x,\delta_x))$, as determined by the box analyzer $\mathcal{A}$, is positive, and (2) $I_{f,\delta_k}(f(x,\delta_x))$ maximizes the proof velocity. VeeP leverages Observation 1 and 2 and approximates them as $S_k(\delta) = \alpha_S + \beta_S \cdot exp(\gamma_S \cdot \delta)$ and $V_k(\delta) = \alpha_V \cdot \delta \cdot exp(-\beta_V \cdot \delta)$. It solves two parametric regression problems to determine $\theta_S^k = (\alpha_S, \beta_S, \gamma_S)$ and $\theta_V^k = (\alpha_V, \beta_V)$. This requires to obtain examples: $e_S^1 = (\delta^1, S(\delta^1)), ..., e_S^M = (\delta^M, S(\delta^M))$ and $e_V^1 = (\delta^1, V(\delta^1)), ..., e_V^M = (\delta^M, V(\delta^M))$. The minimal number of examples is three for $S_k(\delta)$ and two for $V_k(\delta)$. Given the examples, the parameters are determined by minimizing a loss:

$$\theta_S^k = \underset{\alpha_S,\beta_S,\gamma_S}{\text{argmin}}\, L(\alpha_S, \beta_S, \gamma_S, e_S^1, \ldots, e_S^M) \qquad \theta_V^k = \underset{\alpha_V,\beta_V}{\text{argmin}}\, L(\alpha_V, \beta_V, e_V^1, \ldots, e_V^M)$$

For the loss, VeeP uses the least squares error. Given the parameters, VeeP solves the optimization problem (Section 5) to approximate the optimal value of δ_k:

$$\max V_{\theta_V^k}(\delta) \text{ such that } S_{\theta_S^k}(\delta) > 0$$

The remaining question is how to obtain examples. A naive approach is to randomly select $\delta^1, \ldots, \delta^M$ and for each δ^i run the analyzer $\mathcal{A}$ on $I_{f,\delta^i}(f(x,\delta_x))$, to find the sensitivity and velocity. However, these M calls to $\mathcal{A}$ are highly time consuming, especially because their only goal is to predict the next diameter to analyze. Instead, VeeP relies on previous steps to estimate examples by leveraging two empirical observations. First, the function $V_k(\delta)$ is similar to previous $V_{k-i}(\delta)$, for small values of i. Thus, VeeP can use as examples $(\delta_{k-i}, V_{k-i}(\delta_{k-i}))$, for small values of i. Second, the function $S_k(\delta)$ is similar to $S_{k-i}(\delta)$, for small values of i, up to a small alignment term: $S_k(0) - S_{k-i}(0)$. Thus, VeeP can use as examples $(\delta_{k-i}, S_{k-i}(\delta_{k-i}) + S_k(0) - S_{k-i}(0))$, for small values of i. Note that computing $S_k(0)$ does not require to run $\mathcal{A}$, because the sensitivity of $I_{f,0}(f(x,\delta_x))$ is exactly the sensitivity of the input $f(x,\delta_x)$, which can be computed by running it through the classifier D. Based on these observations, VeeP obtains examples as follows. Its first example is $(0, S_k(0))$. Since the velocity of this step's neighborhood is zero, it is not used to approximate $V_k(\delta)$. The next $M-1$ examples are defined

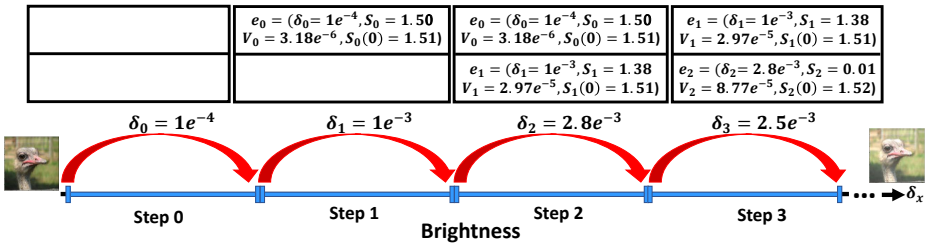

Fig. 3: Analysis for the brightness feature, an ImageNet image, and AlexNetTiny.

as described by the previous $M-1$ predicted diameters, which have already been submitted to $\mathcal{A}$. Note that the examples are defined from previous steps regardless of whether their neighborhoods have been proven robust or not. When VeeP begins its computation and has no previous steps, it executes $M-1$ steps whose diameters are some small predetermined values.

Example Figure 3 shows an example of VeeP's analysis for a brightness neighborhood with $\bar{\delta} = 0.2$, an ImageNet image x (the image on the left) and an AlexNetTiny classifier D. We assume $M = 3$. The first two steps rely on predetermined small diameters $\delta_0 = 10^{-4}$ and $\delta_1 = 10^{-3}$. VeeP begins by submitting to $\mathcal{A}$ the neighborhood $I_{f,\delta_0}(x)$ and $\mathcal{A}$ returns *robust*. VeeP thus updates the accumulated diameter $\delta_x = 10^{-4}$ and constructs the example e_0. The example consists of the sensitivity S_0 and velocity V_0 (computed from $\mathcal{A}$'s analysis), and the sensitivity $S_0(0)$ at $\delta_x = 0$ (computed by running x through D). The next verification step submits to $\mathcal{A}$ the neighborhood $I_{f,\delta_1}(f(x, 10^{-4}))$ and $\mathcal{A}$ returns *robust*. VeeP thus updates $\delta_x = 1.1 \cdot 10^{-3}$ and constructs the example e_1, consisting of the sensitivity S_1 and velocity V_1 (computed from $\mathcal{A}$'s analysis) and the sensitivity $S_1(0)$ (computed by running $f(x, 10^{-4})$ through D). To predict the next diameter δ_2, VeeP relies on e_0, e_1 and $S_2(0)$ (computed by running $f(x, 1.1 \cdot 10^{-3})$ through D). Its examples are: $e_S^0 = (0, 1.52)$, $e_S^1 = (10^{-4}, S_0 + S_2(0) - S_0(0))$, $e_S^2 = (10^{-3}, S_1 + S_2(0) - S_1(0))$, and $e_V^0 = (10^{-4}, V_0)$, $e_V^1 = (10^{-3}, V_1)$. Given the examples, it minimizes the MSE loss to compute the parameters θ_S^2 and θ_V^2. Afterwards, it solves the constrained optimization function to compute δ_2. The result is $\delta_2 = 2.8 \cdot 10^{-3}$. VeeP submits to $\mathcal{A}$ the neighborhood $I_{f,\delta_2}(f(x, 1.1 \cdot 10^{-3}))$ and $\mathcal{A}$ returns *robust*. VeeP updates δ_x and constructs the example e_2, as described before. VeeP predicts the next diameter δ_3, by repeating this process using the examples e_1 and e_2. It continues until reaching the target diameter $\bar{\delta} = 0.2$. The most perturbed image in this neighborhood is shown on the right of Figure 3.

Overall operation The operation of VeeP is summarized in Figure 4 and mostly follows the description above, up to few modifications to guarantee termination. Initially, VeeP sets $\delta_x = 0$ and generates the first $M-1$ steps using predetermined diameters. Every verification step predicts the next diameter based on previous iterations, as described before (steps 1–4 in Figure 4). Then, to avoid certification failures and guarantee termination, VeeP performs three corrections to the

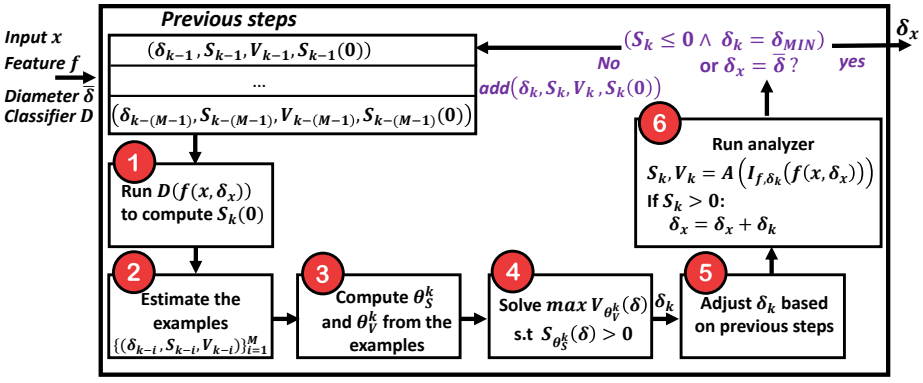

Fig. 4: The VeeP System.

predicted diameter δ_k (step 5). First, it checks whether in the last M steps, there has been a step i with a smaller predicted diameter, $\delta_i < \delta_k$, that failed. If so, VeeP sets δ_k to the minimal value between the last verified diameter (if exists) and the last failed one from the last M steps. Second, it subtracts a small constant from δ_k. Third, it guarantees that δ_k is not below the precision level by setting $\delta_k = \max(\delta_k, \delta_{\text{MIN}})$. These refinements, along with the prediction based on recent examples, aim at mitigating predicting too large or too small diameters. The neighborhood defined by (δ_x, δ_k) is submitted to the analyzer $\mathcal{A}$ (step 6), which returns the real-valued bounds of the output neurons. Accordingly, VeeP computes the sensitivity S_k and velocity V_k. If $S_k > 0$, then the neighborhood is robust and thus δ_x is increased by δ_k. Afterwards, VeeP checks the termination conditions. The first condition is $S_k \leq 0$ and $\delta_k = \delta_{\text{MIN}}$, indicating that the neighborhood is maximal. The second condition is $\delta_x = \bar{\delta}$, indicating that VeeP certified the target diameter. If the conditions are not met, VeeP constructs the example of this step and continues to another iteration.

Correctness analysis We next discuss the time overhead of VeeP and its correctness. The first lemma analyzes the time overhead of VeeP. The overhead is the additional time that VeeP requires compared to an oblivious splitting approach. The overhead of every step consists of the call to the classifier D (to compute $S_k(0)$) and the time to solve the regression problems (to approximate S_k and V_k). The time overhead also includes the $M-1$ initial calls to the analyzer $\mathcal{A}$.

Lemma 1. *The total overhead is* $n \cdot (T_D + T_R) + \Sigma_{i=1}^{M-1} T_{\mathcal{A},i}$, *where* T_D *is the time to run a single input in the classifier* D, T_R *is the time to solve a regression problem from* M *examples,* n *is the number of verification steps and* $T_{\mathcal{A},i}$ *is the execution time of* $\mathcal{A}$ *on the* i^{th} *initial step.*

In practice, T_D and T_R are significantly shorter than the time to run the analyzer $\mathcal{A}$. Since the value of M is small (we pick $M = 3$ or $M = 4$), the overhead of the initial queries to the analyzer is negligible when compared to the total

execution time of VeeP. As a result, we observe that the execution time of VeeP is very close to the optimal greedy baseline that "knows" the optimal diameter of every step. We continue with a lemma guaranteeing termination and a theorem guaranteeing soundness and precision (up to δ_{MIN}). Proofs are provided in the extended version of this paper [23, Appendix B].

Lemma 2. *Given a classifier D, an input x, a feature f and a diameter $\bar{\delta}$, if $\mathcal{A}$ is guaranteed to terminate, then VeeP is guaranteed to terminate.*

Theorem 1. *Given a classifier D, an input x, a feature f, a diameter $\bar{\delta}$, and a precision level δ_{MIN}, if $\mathcal{A}$ is sound (but may be incomplete), then VeeP is:*
- *sound: if it returns $I_{f,\delta_x}(x)$, then this neighborhood is robust, and*
- *precise up to δ_{MIN}: if it returns δ_x smaller than $\bar{\delta}$, then we assume there is $\hat{\delta} \in (\delta_x, \delta_x + \delta_{MIN}]$ such that $x' = f(x, \hat{\delta})$ is an adversarial example.*

6.2 VeeP for Multi-feature Neighborhoods

In this section, we present VeeP's algorithm to verify neighborhoods defined by multiple features $f_1, f_2, \ldots, f_T$. VeeP computes a sequence of verification steps that cover the maximal robust T-dimensional hyper-rectangle neighborhood. The sequence is constructed such that VeeP computes the maximal diameters feature-by-feature. To compute the maximal diameter of the i^{th} feature, VeeP computes the maximal robust i-dimensional neighborhood of the first i features. Similarly to Section 6.1, a verification step is a pair of an offset vector $(\delta_1, \ldots, \delta_T)$ (instead of δ_x) and a diameter δ. A verification step thus corresponds to a hyper-cube neighborhood $I_{f_1,\delta,\ldots,f_T,\delta}(x_0)$, where x_0 is the perturbation of x as determined by the features and offsets $(x_0 = f_T(\ldots(f_2(f_1(x, \delta_1), \delta_2), \ldots), \delta_T))$. While VeeP could predict a different diameter for each feature, this would increase the prediction's complexity by a factor of T. Besides this, the analysis is similar to Section 6.1 but generalizes it to high dimension, resulting in few differences. First, computing the offsets is more subtle than computing δ_x. Second, the examples used for prediction also leverage the *closest* examples. Third, computing the accumulated verified diameters, required for checking the termination conditions, involves obtaining the *vertices* of the certified region. We next explain all these differences, then exemplify VeeP's operation, and finally present the algorithm.

Offsets Initially, all offsets are zero. Recall that VeeP computes the maximal diameters feature-by-feature, and, for every f_i, it computes the maximal robust i-dimensional neighborhood of $f_1, ..., f_i$. After every verification step, VeeP computes the next offsets. Assume VeeP is currently at feature f_i. If a step fails for $\delta > \delta_{\text{MIN}}$, the offsets of the next step are identical. If a step fails for $\delta = \delta_{\text{MIN}}$ or reaches $\bar{\delta}_i$, VeeP computes the initial offsets of f_{i+1}, as shortly described. Otherwise, VeeP computes the next offsets based on a feature-by-feature order (from 1 to i). The order, defined in [23, Appendix C], guarantees that VeeP covers the entire i-dimensional neighborhood. We later exemplify it on a running example. Upon starting a feature f_j, VeeP computes the initial offsets based

on the already certified neighborhoods. This is obtained by finding the earliest step forming a vertex on the j-dimensional boundary of the certified region, such that the vertex's j^{th} offset is within $(0, \bar{\delta}_j)$. This leverages the already certified neighborhoods: since the steps define hyper-cube neighborhoods, as a byproduct of their analysis, there is also progress in the direction of the succeeding, not yet analyzed, features. The complete computation is provided in [23, Appendix C].

Examples The diameter of a verification step is predicted by $M + 1$ examples: $(0, S_k(0))$, $M - 1$ (adapted) recent examples and, to increase the prediction accuracy, the *closest* example, with respect to the Euclidean distance. The $M - 1$ recent examples are used only if they (aim to) advance the diameter of the same feature as the current step does. If not all of them advance the same feature, VeeP completes the missing examples with closest examples or initialization examples.

Termination VeeP terminates when it reaches all target diameters or all maximal diameters. These conditions generalize the termination conditions presented in Section 6.1. To check the first condition, VeeP maintains an array `ds` of the certified diameters, which are updated after every verification step. The diameters are computed from the vertices bounding the certified region. Although the region induced by the maximal diameters is a hyper-rectangle, the certified region may form other shapes. During the analysis, VeeP computes the vertices of the certified region. To update `ds`, it selects the maximal bounded hyper-rectangle, with respect to the Euclidean norm. To check the second condition, VeeP checks whether it has failed for T consecutive iterations for a neighborhood whose diameter is δ_{MIN}. Correctness follows from the the operation of VeeP: upon failure of a neighborhood with diameter δ_{MIN}, it proceeds to the next feature. Thus, T consecutive failures imply that VeeP has reached all maximal diameters.

Example We next exemplify VeeP for a neighborhood defined by brightness and contrast, where $\bar{\delta}_1 = \bar{\delta}_2 = 0.08$ and $M = 3$ (Figure 5). VeeP computes the maximal diameters one by one: first the brightness's diameter and then the contrast's diameter. Figure 5(a) visualizes the verification steps that compute the maximal diameter of brightness. The sequence begins from the offset $(0, 0)$ (i.e., $x_0 = x$), and the computation is similar to Section 6.1. When VeeP reaches $\bar{\delta}_1$, it continues to the contrast feature. It begins by finding the earliest verification step forming a vertex on the 2-dimensional boundary of the certified region, such that the vertex's second offset is within $(0, \bar{\delta}_2)$. This is the first step and the vertex is $(0, 0.018)$ (since this step's diameter is 0.018). Thus, the initial offset of contrast is $(0, 0.018)$. During the analysis of the contrast feature, VeeP computes verification steps feature-by-feature. Thus, after initializing the offsets, VeeP advances the brightness's offset, until reaching its maximal certified diameter (rightmost square, top row, Figure 5(b)). Then, by the order VeeP follows for the verification steps, it (again) looks for the earliest step forming a vertex on the 2-dimensional boundary of the certified region, such that the vertex's second offset is within $(0, \bar{\delta}_2)$. This is the leftmost square, top row, Figure 5(b). Thus, it sets the next offset (i.e., of the leftmost square, top row, Figure 5(c)) to that vertex's offsets. The rest of the

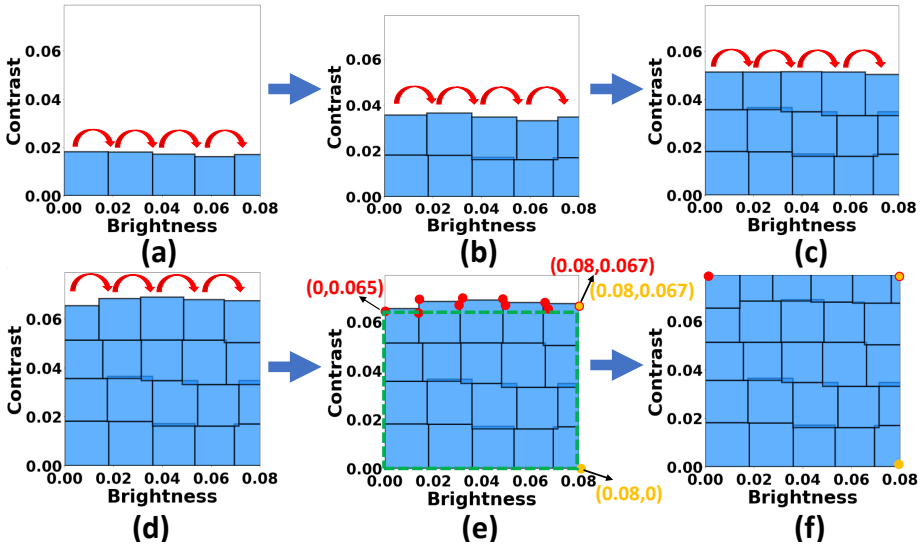

Fig. 5: Example of VeeP's analysis to certify a neighborhood defined by brightness and contrast, for an MNIST image, on a fully-connected network.

computation continues similarly (Figure 5(c), (d), and (f)). We next illustrate the different sets of examples used for the prediction (besides $(0, S_k(0))$). Consider Figure 5(b). The examples used by the middle step at the top row are the two leftmost squares at the top row and the middle square at the row below. The examples used by the leftmost square at the top row are the three closest examples – the three leftmost squares at the bottom row – since there are no steps advancing the contrast's diameter. After every verification step, VeeP constructs for each feature the vertices of the certified neighborhood. Figure 5(e) shows the vertices after completing the verification steps of Figure 5(d): ten red vertices for contrast and two yellow vertices for brightness. Figure 5(f) shows the vertices after completing all verification steps. Given the vertices, VeeP computes the accumulated verified diameter of each feature, which is the minimum coordinate of its vertices. For example, in Figure 5(e), the verified diameter of brightness is 0.08, which is the minimum of the first coordinates of $(0.08, 0)$ and $(0.08, 0.067)$, and similarly, the verified diameter of contrast is 0.065. VeeP updates the current maximal diameters to these diameters if they form a larger hyper-rectangle than the current ones. Note that if VeeP terminates after reaching all target diameters (e.g., Figure 5(f)), the certified region is a hyper-rectangle and is thus returned.

Overall operation Algorithm 1 summarizes the operation of VeeP. VeeP begins by initializing `ds`, the maximal diameters array, the first $M - 1$ examples (as described in Section 6.1), the `offset` array and a counter `count_min`, tracking the number of consecutive failures. Then, it enters a loop, where each iteration computes a single verification step. An iteration of the loop begins by determining

Algorithm 1: Multi-feature-VeeP (D, x, f_1, $\bar{\delta}_1$,..., f_T, $\bar{\delta}_T$)

Input: A classifier D, input x, features $f_1, \ldots, f_T$ and diameters $\bar{\delta}_1, \ldots, \bar{\delta}_T$.
Output: Diameter array ds s.t. $I_{f_1, ds[1], \ldots, f_T, ds[T]}(x)$ is maximally robust.

1 $ds = [0, \ldots, 0]$
2 Ex = InitExamples(M)
3 offsets = $[0, \ldots, 0]$
4 count_min = 0
5 **while** $\exists ds[i] < \bar{\delta}_i \wedge count\_min < T$ **do**
6 $x_0 = perturb(x, f_1, \ldots, f_T, \text{offsets})$
7 $S_0 = D(x_0)$
8 $\delta = \text{predict}(\text{Ex}, x_0, S_0)$
9 $t_0 = time()$
10 $\{l_{o,j}, u_{o,j}\}_{j=1}^{c} = \mathcal{A}(D, I_{f_1, \delta, \ldots, f_T, \delta}(x_0))$
11 $t_1 = time()$
12 $S = l_{c_x} - \max_{j \neq c_x} u_j$
13 $V = S > 0 \ ? \ \frac{\delta}{t_1 - t_0} : 0$
14 Ex = Ex $\cup \{(\delta, S, V, S_0, \text{offsets})\}$
15 offsets = compute_next_offsets($Ex, \bar{\delta}_1, \ldots, \bar{\delta}_T$)
16 $BV_1, \ldots, BV_T$ = compute_certified_neighborhood_vertices(Ex)
17 $ds\_curr = [0, \ldots, 0]$
18 **for** $i = 1$; $i \leq T$; $i++$ **do**
19 $ds\_curr[i] = min\{v_i \mid v \in BV_i\}$
20 **if** *vectorNorm(ds_curr) > vectorNorm(ds)* **then** ds = ds_curr
21 count_min = $(S \leq 0 \wedge \delta == \delta_{\text{MIN}})$? count_min + 1 : 0
22 **return** ds

x_0 from the offsets (Line 6). Then, it progresses as described in Section 6.1 (Lines 7–14): it computes x_0's sensitivity, predicts δ, submits to $\mathcal{A}$, computes the velocity and sensitivity, and adds this verification step as a new example. After that, it computes the new offsets (Line 15). Next, the maximal diameters are computed. To this end, VeeP constructs, for each feature, the vertices of the certified region (Line 16). Computing the vertices is a technical computation determined from the set of examples. We omit the exact computation. Given the vertices, VeeP computes the current verified diameters `ds_curr`. The current verified diameter of feature i is the minimum i^{th} coordinate of its vertices (Lines 17–19). Then, if the Euclidean norm of `ds_curr` is greater than that of `ds`, it updates `ds` (Line 20). Lastly, the counter `count_min` is increased, if $\mathcal{A}$ failed, or resets, otherwise (Line 21). The loop continues as long as VeeP has not reached all target diameters and has not failed during the last T iterations (Line 5).

Correctness We next present the correctness guarantees of Algorithm 1. Proofs are provided in the extended version of this paper [23, Appendix B].

Lemma 3. *Given a classifier D, an input x, features $f_1, \ldots, f_T$ and diameters $\bar{\delta}_1, \ldots, \bar{\delta}_T$, if $\mathcal{A}$ is guaranteed to terminate, then VeeP is guaranteed to terminate.*

Lastly, we show that VeeP is sound and precise, up to precision of δ_{MIN} for each feature's maximal diameter.

Theorem 2. *Given a classifier D, an input x, features $f_1, \ldots, f_T$ and diameters $\bar{\delta}_1, \ldots, \bar{\delta}_T$, if $\mathcal{A}$ is sound (but may be incomplete), then:*
- *VeeP is sound: at the end of the algorithm $I_{f_1,ds[1],\ldots,f_T,ds[T]}(x)$ is robust.*
- *VeeP is precise up to δ_{MIN} for each feature's maximal diameter.*

7 Evaluation

In this section, we evaluate VeeP. We begin with implementation aspects and optimizations and then present our experiments.

Implementation We implemented VeeP in Python[1]. It currently supports neighborhoods defined by one or two features. For the analyzer, it relies on GPUPoly [34]. It further builds on the idea of Semantify-NN [32] that encodes features as input layers with the goal of encoding pixel relations to reduce overapproximation errors. Semantify-NN encodes features using fully-connected and convolutional layers. For some features, this approach is infeasible for high-dimensional datasets because of the high memory overhead. To illustrate, denote the input dimension by $h \times w \times 3$. The HSL input layers, as defined in Semantify-NN, map an (R,G,B) triple into a single value in the feature domain, resulting in a perturbed output of $h \times w$. This output is then translated back to the input domain. Namely, a fully-connected layer requires $(h \times w) \times (h \times w \times 3)$ weights. For ImageNet, where $h = w = 224$, this layer becomes too large to fit into a standard memory (over 60GB). Instead, we observe that for some features the feature layer's weights are mostly zeros and thus this layer can be implemented using sparse layers [37,2]. Our implementation sets $\delta_{\text{MIN}} = 10^{-5}$ and $M = 3$. As optimization, it does not keep all previous examples, but only the required ones, which are dynamically determined. For example, for the neighborhood in Figure 5, VeeP keeps only the examples at the top two rows.

Evaluation setup We trained models and ran the experiments on a dual AMD EPYC 7742 server with 1TB RAM and eight NVIDIA A100 GPUs. We evaluated VeeP on four image datasets: MNIST [28] and Fashion-MNIST [53], with images of size 28×28, CIFAR-10 [25], with images of size 32×32×3, and ImageNet [11], with images of size 224×224×3. We considered fully-connected, convolutional [29], ResNet [18], and AlexNet [26] models. For MNIST and Fashion-MNIST, we used FC-5000x10, a fully-connected network with 50k neurons. For MNIST, we also used a convolutional network SuperConv with 88k neurons (from ERAN's repository[2]). For CIFAR-10, we used ResNetTiny with 311k neurons (from ERAN) and ResNet18 with 558k neurons. For ImageNet, we used AlexNetTiny with 444k and AlexNet with 600k neurons. The last four models were trained with PGD [31].

[1] https://github.com/ananmkabaha/VeeP
[2] https://github.com/eth-sri/eran

Since GPUPoly currently does not support MaxPool layers, we replaced them in AlexNet with convolutional ones (justified by [44]). The CIFAR-10 models were taken from ERAN's repository, and we trained the other models.

Baseline approaches We compare VeeP to popular splitting approaches: branch-and-bound (BaB) [7,48,6,35,52,30,19] and uniform splitting [32,3,42]. Any BaB technique starts by attempting to certify the robustness of the given neighborhood. If it fails, it splits the verification task into two parts and attempts to certify the robustness of each separately. If the certification fails again, BaB repeats the splitting process until all parts certify the original neighborhood. The difference between BaB techniques is what neurons they can split and how they choose what to split. For example, some rely on heavy computations, such as solving a linear program [6,35]. For our setting, where the split focuses on the input neurons and the input has low dimensionality, the *long-edge* approach, which splits the input neuron with the largest interval, has been shown to be efficient [6]. We thus compare to this approach. Uniform splitting splits a neighborhood into smaller neighborhoods of the same size, sufficiently small so the analyzer can certify them. Thus, it requires a pre-determined split size (unlike VeeP and BaB which adapt it during the execution). For a fair comparison, we need to carefully determine this size: providing a too small size will result in too long execution times (biasing our results), while providing a too large size will result in certification failures. Thus, we estimate the maximal split size which will enable the uniform splitting to certify successfully. To this end, before running the experiments, we run the following computation. For each neighborhood, we define several smaller neighborhoods. For each, we look for the maximal ϵ which can be verified by GPUPoly without splitting. Finally, we determine the split size of the uniform splitting to be the minimal value of ϵ across all these smaller neighborhoods. For a fair comparison, both baseline approaches were integrated in our system, i.e., they rely on GPUPoly and the feature layers described before.

Experiments We run two experiments: one limits the execution time with a timeout and measures the maximal certified diameter, and the other one measures execution time as a function of the certified diameter. In each experiment, we run multiple problem instances. In each instance, we provide each approach a network, an image, one or two features, and a target diameter (if there are two features, both have the same target diameter). We define the target diameter to be the diameter of the minimal feature adversarial example δ_{adv} (computed by a grid search). That is, we provide each approach an upper bound on the maximal certified diameter. We measure how close is the returned certified diameter to δ_{adv}. Note that our problem instances are challenging because the feature neighborhoods we consider are the largest possible.

Maximal certified diameter given a timeout In the first experiment, we evaluate the maximal certified diameter of all approaches, given a timeout. The evaluated feature neighborhoods are defined by brightness (a linear feature) and contrast and HSL (non-linear features). The contrast feature defines the brightness difference

Table 1: VeeP vs. branch-and-bound and uniform splitting over brightness, contrast, hue, saturation, and lightness neighborhoods, averaged over 50 images.

Dataset	Model			VeeP		BaB		Uniform	
			δ_{adv}	δ_f%	$t[m]$	δ_f%	t[m]	δ_f%	t[m]
MNIST	SuperConv	Brightness	0.61	100	0.5	100	1.16	98	4.1
MNIST	SuperConv	B&C	0.56	99	26.1	98	35.2	81	77.3
MNIST	FC 5000x10	Brightness	0.15	100	1.9	100	11.5	100	13.4
MNIST	FC 5000x10	B&C	0.134	94	54.5	59	86.4	62	81.8
F-MNIST	FC 5000x10	Brightness	0.3	100	3.5	100	15.5	100	27.9
CIFAR-10	ResNetTiny	Brightness	0.42	100	7.9	100	32.1	89	60.6
CIFAR-10	ResNetTiny	B&C	0.3	96	73.4	49	144.6	30	164.1
CIFAR-10	ResNetTiny	Hue	3.36	99	27.5	62	59.1	77	48.94
CIFAR-10	ResNetTiny	Saturation	0.83	98	5.6	100	21.0	96	68.8
CIFAR-10	ResNetTiny	Lightness	0.39	100	10.8	100	45.9	76	32.6
ImageNet	AlexNetTiny	Brightness	0.22	95	68.8	59	87.6	59	82.7
ImageNet	AlexNetTiny	Hue	0.99	78	40.6	25	67.4	37	68.1
ImageNet	AlexNetTiny	Saturation	0.39	97	27.7	79	69.0	71	74.9
ImageNet	AlexNetTiny	Lightness	0.16	93	64.8	17	83.4	52	71.4

between light and dark areas of the image, and the HSL features are color space transformations, where hue defines the position in the color wheel, saturation controls the image's colorfulness and lightness the perceived brightness. We run VeeP, BaB, and uniform splitting over the different models. For most networks and neighborhoods, we let each splitting approach run on a single GPU for 1.5 hours. For ResNet18, AlexNet, and the brightness and contrast (B&C) neighborhoods of TinyResNet, we let each splitting approach run on eight GPUs for 3 hours. We measure the execution time in minutes $t[m]$ and the maximal certifiable diameter δ_f. We compare δ_f to the diameter of the closest adversarial example in the feature domain δ_{adv} (for B&C, we compare to (δ_{adv},δ_{adv})). Table 1 reports our results for the smaller models. Each result is averaged on 50 images. The results indicate that VeeP proves on average at least 96% of the maximal certifiable diameters in 29 minutes. The maximal diameters computed by the baselines are 74%, for BaB, and 73%, for uniform splitting. Their execution times are 54 minutes, for BaB, and 62 minutes, for uniform splitting. Table 2 reports our results for the two largest models, ResNet18 and AlexNet. Because of the long timeout, we focus on ten images and compare only to BaB. Our results show that VeeP proves at least 96% of the maximal diameters, while BaB proves 44%. VeeP's execution time is 98 minutes, whereas BaB is 160 minutes.

Execution time as a function of the certified diameter In the second experiment, we measure the execution time of every approach as a function of the certified diameter. In this experiment, there is no timeout and thus we focus on two models, ResNetTiny and AlexNetTiny, and two features: brightness and saturation. For

Table 2: VeeP vs. branch-and-bound over large models, averaged over 10 images.

Dataset	Model			VeeP		BaB	
			δ_{adv}	δ_f%	$t[m]$	δ_f%	t[m]
CIFAR-10	ResNet18	Brightness	0.41	100	88.4	58	150
CIFAR-10	ResNet18	Saturation	0.85	98	45.2	98	123
ImageNet	AlexNet	Brightness	0.42	92	130	6	180
ImageNet	AlexNet	Saturation	0.56	100	67.3	52	165
ImageNet	AlexNet	Lightness	0.32	93	162	3	180

each network and a feature, we consider 50 images. For each network, image, and a feature, the target diameter is the diameter of the closest adversarial example δ_{adv}. We run all approaches until completion. During the execution of each approach, we record the intermediate progress, that is, the required time for certifying $r \cdot \delta_{adv}$ of the neighborhood, for ratio $r \in \{0.1, 0.2, \ldots, 0.8, 0.9, 0.95, 0.98\}$.

Figure 6 shows the results of this experiment. It depicts the execution time in minutes of each approach as a function of r, i.e., the ratio of the certified diameter and the target diameter δ_{adv}. Our results indicate that VeeP provides acceleration of 4.4x compared to BaB and acceleration of 10.2x compared to uniform splitting. The figure demonstrates the main drawbacks of uniform splitting and branch-and-bound. On the one hand, choosing a large step size for uniform splitting can certify smaller ratios of the target diameter more quickly. On the other hand, for larger ratios, uniform splitting must use a smaller step size, which significantly increases the execution time. The results also show that BaB wastes a lot of time on attempts to certify too large neighborhood until converging to a certifiable split size. We note that both baseline approaches are sub-optimal since they do not attempt to compute the optimal split size. In contrast, VeeP predicts the split sizes that minimize the execution time and thus performs better than the baselines. We validate VeeP's optimality by comparing it to a theoretical greedy optimal baseline. The theoretical baseline "knows" (without any computation) the optimal step size for every verification step. To simulate it, before every verification step of the optimal baseline, we compute the optimal step size by running a grid search over the remaining diameter (i.e., $\bar{\delta} - \delta_x$). We then let the optimal baseline pick the diameter determined by the grid search. Note that this baseline is purely theoretical: we do not consider the execution time of running the grid searches as part of its execution time. Our results indicate that VeeP's performance is very close to the theoretical baseline's performance, VeeP is slower by only a factor of 1.2x. The additional overhead of VeeP stems from several factors: (1) the time to estimate the predictors, (2) the time to run the network on $f(x, \delta_x)$, and (3) the inaccuracies of our predictors and correction steps.

Lastly, we exemplify how large the feature neighborhoods that VeeP certifies are. Figure 7 shows four certified neighborhoods, defined by different features. For each, the figure shows the features, the range of the certified diameters, and

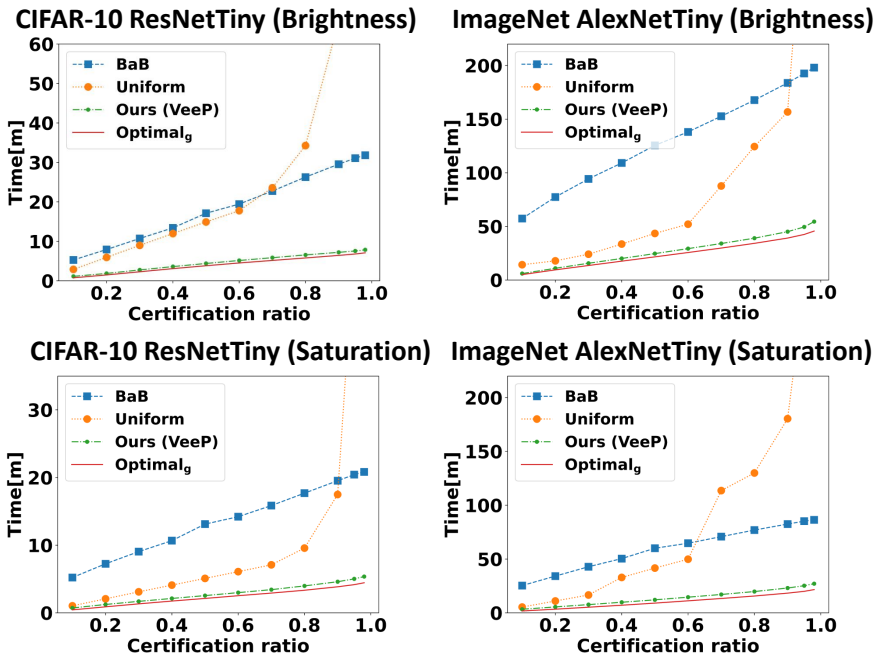

Fig. 6: Comparison of VeeP to uniform splitting, branch-and-bound, and a greedy optimal baseline, averaged over 50 images.

several images generated by uniformly sampling from the certified range. The images are organized across the diameter axis, where the original image x is at the origin. These examples demonstrate that the certified feature neighborhoods contain images that are visually different compared to the original image. Being able to certify large feature neighborhoods allows network designers understand the robustness level of their networks to feature perturbations.

8 Related Work

In this section, we discuss the most closely related work to VeeP.

Network robustness and feature verification Many works introduce verifiers analyzing the robustness of L_∞-balls, where each pixel is bounded by an interval [52,12,24,48,34,38,54,16,42,13,47]. Other works consider feature verification [32,3,42,49]. Earlier works on feature verification, focusing on rotations, brightness and contrast, translate feature neighborhoods into L_∞ neighborhoods and then analyze them with existing verifiers [42,49]. Recent works encode the feature constraints into the verifier. One work relies on Monte Carlo sampling to overapproximate geometric feature constraints by convex linear bounds [3]. The bounds are refined by solving an optimization problem and then submitted to an

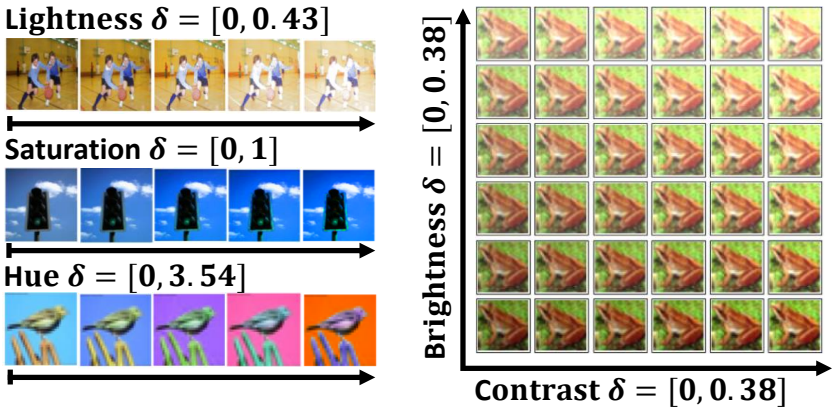

Fig. 7: Examples of images in feature neighborhoods, certified by VeeP.

existing verifier. Other work proposes an input layer that encodes the feature and is added to the original network [32]. All works also employ uniform splitting.

Splitting techniques To increase precision and scalability, many verifiers rely on uniform splitting [32,3,42] or branch-and-bound (BaB) [7,48,6,35,52,30,19]. Long-edge is a common BaB technique that splits the input with the largest interval [7,52]. Smart-Branching (BaBSB) [7] and Smart-ReLU (BaBSR) [6,48] rely on a fast computation to estimate the expected improvement of splitting an input or a neuron and then split the one maximizing the improvement. Filtered Smart Branching (FSB) extends BaBSR by bound propagation to estimate multiple candidates of BaBSR [35,48]. Another work relies on an indirect effect analysis to estimate the neuron splitting gain [19]. Others suggest to train GNNs via supervised learning to obtain a splitting strategy [30]. However, building the dataset and training the GNNs can be time consuming. In contrast to BaB, which lazily splits inputs or neurons, VeeP dynamically predicts the optimal split.

Feature attacks Several adversarial attacks rely on semantic feature perturbations. One work relies on HSV color transformations (which is close to HSL) [21]. Other works link adversarial examples to PCA features [56,4,8]. Other feature attacks include facial feature perturbations [15], colorization and texture attacks [5], features obtained using scale-invariant feature transform (SIFT) [51], and semantic attribute perturbations using multi-attribute transformation models [22].

Learning Our approach is related to several learning techniques. It is mainly related to active learning, where a learner learns a concept by querying an oracle [1]. Active learning is suitable for tasks in which labeling a dataset is expensive [55], for example real-life object detection [17], crowd counting [57], and image segmentation [39]. Similarly, in our setting, querying the analyzer to obtain examples is expensive. Our setting is also related to online learning, where new data gradually becomes available. Online learning typically addresses tasks with

time-dependent data [20], e.g., visual tracking [33], stock price prediction [50], and recommendation systems [9]. In contrast, VeeP's examples are not time-dependent. Our approach is also related to CEGIS and CEGAR. Counterexample-guided inductive synthesis (CEGIS) synthesizes a program by iteratively proposing candidate solutions to an oracle [43]. The oracle either confirms or returns a counterexample. Counterexample-guided abstraction-refinement (CEGAR) is a program verification technique for dynamically computing abstractions capable of verifying a given property [10]. It begins from some abstraction to the program and iteratively refines it as long as there are spurious counterexamples. In contrast, VeeP relies on recent examples, not necessarily counterexamples.

9 Conclusion

We presented VeeP, a system for verifying the robustness of deep networks in neighborhoods defined by a set of features. Given a neighborhood, VeeP splits the verification process into a series of verification steps, each aiming to verify a maximal part of the given neighborhood in a minimal execution time. VeeP defines the next verification step by constructing velocity and sensitivity predictors from previous steps and by considering recent failures. VeeP is guaranteed to terminate and is sound and precise up to a parametric constant. We evaluate VeeP over challenging experiments: deep models for MNIST, Fashion-MNIST, CIFAR-10 and ImageNet, and large feature neighborhoods, defined by the closest feature adversarial example. Results show that the average diameter of the neighborhoods that VeeP verifies is at least 96% of the maximal certifiable diameter. Additionally, VeeP provides a significant acceleration compared to existing splitting approaches: up to 10.2x compared to uniform splitting and 4.4x compared to branch-and-bound.

Acknowledgements. We thank the reviewers for their feedback. This research was supported by the Israel Science Foundation (grant No. 2605/20).

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. In Inf. Comput. (1987)
2. Ardakani, A., Condo, C., Gross, W.J.: Sparsely-connected neural networks: Towards efficient VLSI implementation of deep neural networks. In ICLR (2017)
3. Balunovic, M., Baader, M., Singh, G., Gehr, T., Vechev, M.T.: Certifying geometric robustness of neural networks. In NeurIPS (2019)
4. Bhagoji, A.N., Cullina, D., Sitawarin, C., Mittal, P.: Enhancing robustness of machine learning systems via data transformations. In CISS (2018)
5. Bhattad, A., Chong, M.J., Liang, K., Li, B., Forsyt, D.A.: Unrestricted adversarial examples via semantic manipulation. In ICLR (2020)
6. Bunel, R., Lu, J., Turkaslan, I., Torr, P.H.S., Kohli, P., Kumar, M.P.: Branch and bound for piecewise linear neural network verification. J. Mach. Learn. Res. (2020)

7. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In NeurIPS (2018)
8. Carlini, N., Wagner., D.A.: Adversarial examples are not easily detected: Bypassing ten detection methods. In AISec (2017)
9. Chen, N., Hoi, S.C.H., Li, S., Xiao, X.: Mobile app tagging. In WSDM (2016)
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In CAV (2000)
11. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In CVPR (2009)
12. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In CAV (2020)
13. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In SP (2018)
14. Goodfellow, I.J., Shlens, J., Szegedy., C.: Explaining and harnessing adversarial examples. In ICLR (2015)
15. Goswami, G., Ratha, N.K., Agarwal, A., Singh, R., Vatsa., M.: Unravelling robustness of deep learning based face recognition against adversarial attacks. In AAAI (2018)
16. Gowal, S., et al.: Scalable verified training for provably robust image classification. In ICCV (2019)
17. Haussmann, E., et al.: Scalable active learning for object detection. In IV (2020)
18. He, K., Zhang, X., Ren, S., Sun., J.: Deep residual learning for image recognition. In CVPR (2016)
19. Henriksen, P., Lomuscio, A.: DEEPSPLIT: an efficient splitting method for neural network verification via indirect effect analysis. In IJCAI (2021)
20. Hoi, S.C., Sahoo, D., Lu, J., Zhao, P.: Online learning: A comprehensive survey. In Neurocomputing (2021)
21. Hosseini, H., Poovendran, R.: Semantic adversarial examples. In CVPR Workshops (2018)
22. Joshi, A., Mukherjee, A., Sarkar, S., Hegde, C.: Semantic adversarial attacks: Parametric transformations that fool deep classifiers. In ICCV (2019)
23. Kabaha, A., Drachsler-Cohen, D.: Boosting robustness verification of semantic feature neighborhoods. In https://arxiv.org/abs/2209.05446 (2022)
24. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer., M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In CAV (2017)
25. Krizhevsky, A.: Learning multiple layers of features from tiny images. (2009)
26. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In NeurIPS (2012)
27. Kurakin, A., Goodfellow, I.J., Bengio, S.: Adversarial machine learning at scale. In ICLR (2017)
28. Lecun, Y., Bottou, L., Bengio, Y., Haffner., P.: Gradient-based learning applied to document recognition. In Proc. IEEE 86(11): 2278-2324 (1998)
29. LeCun, Y., et al.: Backpropagation applied to handwritten zip code recognition. In Neural Comput (1989)
30. Lu, J., Kumar, M.P.: Neural network branching for neural network verification. In ICLR (2020)
31. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In ICLR (2018)
32. Mohapatra, J., Weng, T., Chen, P., Liu, S., Daniel, L.: Towards verifying robustness of neural networks against A family of semantic perturbations. In CVPR (2020)

33. M.Y., A., et al.: A survey on online learning for visual tracking. In Vis Comput 37, 993–1014 (2021)
34. Müller, C., Serre, F., Singh, G., Püschel, M., Vechev, M.: Scaling polyhedral neural network verification on gpus. In MLSYS (2021)
35. Palma, A.D., et al.: Improved branch and bound for neural network verification via lagrangian decomposition. arXiv:2104.06718 (2021)
36. Papernot, N., McDaniel, P.D., Goodfellow, I.J., Jha, S., Celik, Z.B., Swami, A.: Practical black-box attacks against machine learning. In AsiaCCS (2017)
37. Richter, O., Wattenhofer, R.: Treeconnect: A sparse alternative to fully connected layers. In ICTAI (2018)
38. Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A.M., Vechev, M.T.: Scalable polyhedral verification of recurrent neural networks. In CAV (2021)
39. Saidu, I.C., Csató, L.: Active learning with bayesian unet for efficient semantic image segmentation. In J. Imaging (2021)
40. Singh, G., Ganvir, R., Püschel, M., Vechev., M.T.: Beyond the single neuron convex barrier for neural network certification. In NeurIPS (2019)
41. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In NeurIPS (2018)
42. Singh, G., Gehr, T., Püschel, M., T., M.: An abstract domain for certifying neural networks. In Proc. ACM Program. Lang (2019)
43. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In ASPLOS (2006)
44. Springenberg, J.T., Dosovitskiy, A., Brox, T., Riedmiller, M.A.: Striving for simplicity: The all convolutional net. In ICLR Workshop (2015)
45. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus., R.: Intriguing properties of neural networks. In ICLR (2014)
46. Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I.J., Boneh, D., D., P.: Ensemble adversarial training: Attacks and defenses. In ICLR (2018)
47. Tran, H., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. In CAV (2020)
48. Wang, S., et al.: Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In NeurIPS (2021)
49. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In NeurIPS (2018)
50. Wang, X., Yang, K., Liu, T.: Stock price prediction based on morphological similarity clustering and hierarchical temporal memory. In IEEE Access 10.1109/ACCESS.2021.3077004 (2021)
51. Wicker, M., Huang, X., Kwiatkowska., M.: Feature-guided black-box safety testing of deep neural networks. In TACAS (2018)
52. Wu, H., et al.: Parallelization techniques for verifying neural networks. In FMCAD (2020)
53. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. CoRR, abs/1708.07747 (2017)
54. Xu, K., et al.: Automatic perturbation analysis for scalable certified robustness and beyond. In NeurIPS (2020)
55. Zhan, X., Wang, Q., Huang, K., Xiong, H., Dou, D., Chan, A.B.: A comparative survey of deep active learning. CoRR abs/2203.13450 (2022)
56. Zhang, Y., Tian, X., Li, Y., Wang, X., Tao, D.: Principal component adversarial example. In IEEE Trans. Image Process (2020)
57. Zhao, Z., Shi, M., Zhao, X., Li, L.: Active crowd counting with limited supervision. In ECCV (2020)