Toward Certified Robustness Against Real-World Distribution Shifts

Abstract—We consider the problem of certifying the robustness of deep neural networks against real-world distribution shifts. To do so, we bridge the gap between hand-crafted specifications and realistic deployment settings by considering a neural-symbolic verification framework in which generative models are trained to learn perturbations from data and specifications are defined with respect to the output of these learned models. A pervasive challenge arising from this setting is that although S-shaped activations (e.g., sigmoid, tanh) are common in the last layer of deep generative models, existing verifiers cannot tightly approximate S-shaped activations. To address this challenge, we propose a general meta-algorithm for handling S-shaped sigmoid activations which leverages classical notions of counter-example-guided abstraction refinement. The key idea is to “lazily” refine the abstraction of S-shaped sigmoid functions to exclude spurious counter-examples found in the previous abstraction, thus guaranteeing progress in the verification process while keeping the state-space small. For networks with sigmoid activations, we show that our technique outperforms state-of-the-art verifiers on certifying robustness against both canonical adversarial perturbations and numerous real-world distribution shifts. Furthermore, experiments on the MNIST and CIFAR-10 datasets show that distribution-shift-aware algorithms have significantly higher certified robustness against distribution shifts.

Index Terms—certified robustness, distribution shift, generative models

I. INTRODUCTION

Despite remarkable performance in various domains, it is well-known that deep neural networks (DNNs) are susceptible to seemingly innocuous variation in their input data. Indeed, recent studies have conclusively shown that DNNs are vulnerable to a diverse array of changes ranging from norm-bounded perturbations [1, 7] to distribution shifts in weather conditions in perception tasks [8, 12]. To address these concerns, there has been growing interest in using formal methods to obtain rigorous verification guarantees for neural networks with respect to particular specifications [13–50]. A key component of verification is devising specifications that accurately characterize the expected behavior of a DNN in realistic deployment settings. Designing such specifications is crucial for ensuring that the corresponding formal guarantees are meaningful and practically relevant.

By and large, the DNN verification community has focused on specifications described by simple analytical expressions. This line of work has resulted in a set of tools which cover specifications such as certifying the robustness of DNNs against norm-bounded perturbations [13–31, 44, 51]. However, while such specifications are useful for certain applications, such as protecting against malicious security threats [52], there are many other applications where real-world distribution shifts, such as change in weather conditions, are more relevant, and these often cannot be described via a set of simple equations. While progress has been made toward broadening the range of specifications [53–56], it remains a crucial open challenge to narrow the gap between formal specifications and distribution shifts.

One promising approach for addressing this challenge involves incorporating neural network components in the specifications [56–58]. Such an approach has been used in the past to verify safety properties of neural network controllers [56] as well as robustness against continuous transformations between images [57]. More recently, Xie et al. [58] generalize this approach by proposing a specification language which can be used to specify complex specifications that are otherwise challenging to define. In this paper, we leverage and extend these insights to obtain a neural-symbolic (an integration of machine learning and formal reasoning) approach for verifying robustness against real-world distribution shifts. The key idea is to incorporate deep generative models that represent real-world distribution shifts [8, 9, 59, 60] in the formal specification.

To realize this idea, there remains one important technical challenge: all the previous work [56–58] assumes that both the neural networks being verified and the generative models are piecewise-linear. This assumption is made in order to leverage off-the-shelf neural network verifiers [56, 58], which focus on piecewise-linear activation functions such as ReLU. However, in practice the majority of (image) generative models [61–65] use S-shaped activation functions transcendental activation functions such as sigmoid and tanh in the output layer. While there are a few existing methods for verifying neural networks with sigmoidal activations [31, 34, 50, 51, 66], they all rely on a one-shot abstraction of the sigmoidal activations and suffer from lack of further progress if the verification fails on the abstraction. To bridge this gap and enable the use of a broad range of generative model architectures in the neural symbolic verification approach, we propose a novel abstraction-refinement algorithm for handling transcendental activation functions. We show that this innovation significantly boosts verification precision when compared to existing approaches.

Our neural-symbolic approach has the obvious limitation that the quality of the formal guarantee depends on the quality of the neural network used in the specification. However, the approach also possesses several pragmatically useful features that mitigate this limitation: if the verification fails, then a counter-example is produced; by examining the counter-
example, we can determine whether it is a failure of the model being tested or a failure of the generative model. In either case, this gives us valuable information to improve the verification framework. For example, the failure of the generative model might point us to input regions where the generative model is under-trained and data-augmentation is needed. On the other hand, if the verification succeeds, then the generative model represents a large class of inputs for which we know the model is robust.

**Contributions.** We summarize our contributions as follows:

- We describe a framework for verifying DNNs against real-world distribution shifts by incorporating deep generative models that capture distribution shifts—e.g., changes in weather conditions or lighting in perception tasks—as verification specifications.
- We propose a novel counter-example-guided abstraction refinement strategy for verifying networks with transcendental activation functions.
- We show that our verification techniques are significantly more precise than existing techniques on a range of challenging real-world distribution shifts on MNIST and CIFAR-10, as well as on canonical adversarial robustness benchmarks.

**II. PROBLEM FORMULATION**

In this section, we formally define the problem of verifying the robustness of DNN-based classifiers against real-world distribution shifts. The key step in our problem formulation is to propose a unification of logical specifications with deep generative models which capture distribution shifts.

**Neural network classification.** We consider classification tasks where the data consists of instances \( x \in \mathbb{X} \subseteq \mathbb{R}^{d_0} \) and corresponding labels \( y \in [k] := \{1, \ldots, k\} \). The goal of this task is to obtain a classifier \( C_f : \mathbb{R}^d \rightarrow [k] \) such that \( C_f \) can correctly predict the label \( y \) of each instance \( x \) for each \((x, y)\) pair. In this work, we consider classifiers \( C_f(x) \) defined by

\[
C_f(x) = \arg\max_{j \in [k]} f_j(x),
\]

where we take \( f : \mathbb{R}^{n_0} \rightarrow \mathbb{Y} \subseteq \mathbb{R}^{d_L} \) (with \( d_L = k \)) to be an \( L \)-layer feed-forward neural network with weights and biases \( W^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}} \) and \( b^{(i)} \in \mathbb{R}^{d_i} \) for each \( i \in [L] \) respectively. More specifically, we let \( f(x) = n^{(L)}(x) \) and recursively define

\[
n^{(i)}(x) = W^{(i)} \left( \hat{n}^{(i-1)}(x) \right) + b^{(i)}, \quad \hat{n}^{(i)}(x) = \rho \left( n^{(i)}(x) \right), \quad \text{and} \quad \hat{n}^{(0)}(x) = x.
\]

Here, \( \rho \) is a given activation function (e.g., ReLU, sigmoid, etc.) and \( n^{(i)} \) and \( \hat{n}^{(i)} \) represent the pre- and post-activation values of the \( i \)-th layer of \( f \) respectively.

**Perturbation sets and logical specifications.** The goal of DNN verification is to determine whether or not a given logical specification regarding the behavior of a DNN holds in the classification setting described above. Throughout this work, we use the symbol \( \Phi \) to denote such logical specifications, which define relations between the input and output of a DNN. That is, given input and output properties \( \Phi_{in} \) and \( \Phi_{out} \) respectively, we express logical specifications \( \Phi \) in the following way:

\[
\Phi := (\Phi_{in}(x) \Rightarrow \Phi_{out}(y)).
\]

For example, given a fixed instance-label pair \((x, y)\), the specification

\[
\Phi := (||x - \overline{x}||_p \leq \epsilon \implies C_f(x) = y)
\]

captures the property of robustness against norm-bounded perturbations by checking whether all points in an \( \ell_p \)-norm ball centered at \( x \) are classified by \( C_f \) as having the label \( y \).
Although the study of specifications such as \(\Phi := (x \in S(x)) \Rightarrow C_f(x) = \bar{y}\) has resulted in numerous verification tools, there are many problems which cannot be described by this simple analytical model, including settings where data varies due to distribution shifts. For this reason, it is of fundamental interest to generalize such specifications to capture more general forms of variation in data. To do so, we consider abstract perturbation sets \(S(x)\), which following [9] are defined as “a set of instances that are considered to be equivalent to \([a fixed instance] x\).” An example of an abstract perturbation set is illustrated in Figure 1, wherein each instance in \(S(x)\) shows the same street sign with varying levels of snow. Ultimately, as in the case of norm-bounded robustness, the literature surrounding abstract perturbation sets has sought to train classifiers to predict the same output for each instance in \(S(x)\) [8, 9, 59].

**Learning perturbation sets from data.** Designing abstract perturbation sets \(S(x)\) which accurately capture realistic deployment settings is critical for providing meaningful guarantees. Recent advances in the generative modeling community have shown that distribution shifts can be *provably* captured by deep generative models. The key idea in this line of work is to parameterize perturbation sets \(S(x)\) in the latent space \(Z\) of a generative model \(G(x, z)\), where \(G\) takes as input an instance \(x\) and a latent variable \(z \in Z\). Prominent among such works is [9], wherein the authors study the ability of conditional variational autoencoders (CVAEs) to capture shifts such as variation in lighting and weather conditions in images. In this work, given a CVAE parameterized by \(G(x, \mu(x) + z\sigma(x))\), where \(\mu(x)\) and \(\sigma(x)\) are neural networks, the authors consider abstract perturbation sets of the form

\[
S(x) := \{G(x, \mu(x) + z\sigma(x)) : ||z|| \leq \delta\}. 
\]  

(5)

In particular, \(\mu(x)\) denotes a neural network that maps each instance \(x\) to the mean of a normal distribution over the latent space \(Z\) conditioned on \(x\). Similarly, \(\sigma(x)\) maps \(x\) to the standard deviation of this distribution in the latent space.\(^{1}\) It’s noteworthy that in this framework, the perturbation upper bound \(\delta\) is scaled by \(\sigma(x)\), meaning that different instances \(x\) and indeed different models \(G\) will engender different relative certifiable radii.

Under favorable optimization conditions, the authors of [9] prove that CVAEs satisfy two statistical properties which guarantee that the data belonging to learned perturbation sets in the form of (5) produce realistic approximations of the true distribution shift (c.f. Assumption 1 and Thms. 1 and 2 in [9]). In particular, the authors of [9] argue that if a learned perturbation set \(S(x)\) in the form of (5) has been trained such that the population-level loss is bounded by two absolute constants, then \(S(x)\) well-approximates the true distribution shift in the following sense: with high probability over the latent space, for any clean and perturbed pair \((x, \tilde{x})\) corresponding to a real distribution shift, there exists a latent code \(z\) such that

\[
||z||_2 \leq \alpha \text{ and } ||G(x, \mu(x)z - \sigma(x)) - \tilde{x}|| \leq \beta \text{ for two small constants } \alpha \text{ and } \beta \text{ depending on the CVAE loss.}
\]

To further verify this theoretical evidence, we show that this framework successfully captures real-world shifts on MNIST and CIFAR-10 in Figure 2.

**Verifying robustness against learned distribution shifts.** To bridge the gap between formal verification methods and perturbation sets which accurately capture real-world distribution shifts, our approach in this paper is to incorporate perturbation sets parameterized by deep generative models into verification routines. We summarize this setting in the following problem statement.

**Problem II.1.** Given a DNN-based classifier \(C_f(x)\), a fixed instance-label pair \(\{x, y\}\), and an abstract perturbation set \(S(x)\) in the form of (5) that captures a real-world distribution shift, our goal is to determine whether the following neural-symbolic specification holds:

\[
\Phi := (x \in S(x)) \Rightarrow C_f(x) = \bar{y}
\]  

(6)

In other words, our goal is to devise methods which verify whether a given classifier \(C_f\) outputs the correct label \(y\) for each instance in a perturbation set \(S(x)\) parameterized by a

---

\(^{1}\)This notation is consistent with [9], wherein the authors use the same parameterization for conditional VAEs.

---

(a) MNIST samples. From top to bottom, the distribution shifts are scale, brightness, contrast, and Gaussian blur.

(b) CIFAR-10 samples. The distribution shifts for these sets are brightness (top) and fog (bottom).
generative model $G$.

III. TECHNICAL APPROACH AND CHALLENGES

The high-level idea of our approach is to consider the following equivalent specification to (6), wherein we absorb the generative model $G$ into the classifier $C$:

$$\Phi = (\|\| \leq \delta \implies CQ_{z}(\mathbf{x}) = \mathbf{y})$$  \hspace{1cm} (7)

In this expression, we define

$$Q_{z}(x) = (f \circ G)(x, \mu(x) + z\sigma(x))$$  \hspace{1cm} (8)

Reflexion strategy for S-shaped activations. For piecewise-linear activations, there is a natural refinement strategy for S-shaped activations and there are trade-offs between how precise the over-approximation is and how efficient it is to reason about the over-approximation. For example, a piecewise-linear over-approximation can be more precise than the linear over-approximation (in this case linear) is typically computationally easier than reasoning about the concrete representation, and moreover, if the specification holds on the abstraction then it actually holds on the original network.

In the wider verification literature, there are a handful of verification techniques that can handle S-shaped sigmoid functions \[18, 31, 34, 51, 66, 67\]. Each of these methods rely on abstraction, which builds an over-approximation of the network behavior. However, abstraction-based methods generally suffer from imprecision depending on the tightness of the abstraction, especially when verifying a neural network on large input domains. Each of these methods is based on abstraction, which conservatively approximates the behavior of the neural network. We define abstraction formally in Sec. IV but illustrate the key ideas here in Fig. 3 which shows the popular sigmoid activation function

$$\rho(x) = \frac{1}{1 + e^{-x}}. \hspace{1cm} (9)$$

Suppose the input $x$ is bounded between $l$ and $u$, the pre- and post- sigmoid values can be precisely described as $D = \{(x, y) \mid y = \rho(x) \land l \leq x \leq u\}$. However, instead of using this precise representation, we could over-approximate $D$ as $D' = \{(x, y) \mid y \leq ax + b \land y \geq cx + d \land l \leq x \leq u\}$ (the yellow convex region in Fig. 3), where $ax + b$ and $cx + d$ are the two lines crossing the sigmoid function at $(l, \rho(l))$ and $(u, \rho(u))$. $D'$ over-approximates $D$ because the latter is a subset of the former. Abstraction-based methods typically over-approximate all non-linear connections in the neural network and check whether the specification holds on the over-approximation. The benefit is that reasoning about the over-approximation (in this case linear) is typically computationally easier than reasoning about the concrete representation, and moreover, if the specification holds on the abstraction then it actually holds on the original network.

Previous work has studied different ways to over-approximate S-shaped activations and there are trade-offs between how precise the over-approximation is and how efficient it is to reason about the over-approximation. For example, a piecewise-linear over-approximation can be more precise than the linear over-approximation (in Fig. 3) but reasoning about the former is more computationally challenging. However, whichever over-approximation one uses, all abstraction-based methods suffer from imprecision: if a counter-example to the specification is found on the over-approximation (which means the over-approximation violates the specification), we cannot conclude that the original network also violates the specification. This is because the counter-example maybe spurious as it violates the constraints imposed by the precise, unabstracted neural network (as shown red in Fig. 3).

This spurious behavior demonstrates that there is a need for refinement of abstraction-based methods to improve the precision. For piecewise-linear activations, there is a natural refinement step: performing case analysis on the activation phases. However, when dealing with S-shaped sigmoid activations, it is less clear how to perform this refinement, because if the refinement is performed too aggressively, the state space may explode and exceed the capacity of current verifiers. To address this technical challenge, we propose a counter-example guided refinement strategy for S-shaped sigmoid activation functions.

![Fig. 3: An abstraction of the sigmoid activation function.](image-url)
Algorithm 1 VNN-CEGAR($M := (V, \mathcal{X}, \mathcal{Y}, \phi_{\text{aff}}, \phi_{p}), \Phi)$

1: function VNN-CEGAR($M := (V, \mathcal{X}, \mathcal{Y}, \phi_{\text{aff}}, \phi_{p}), \Phi$)
2:     $M' \leftarrow \text{Abstract}(M)$
3:     while true do
4:         $\langle \alpha, \text{proven} \rangle \leftarrow \text{Prove}(M', \Phi)$  \textbf{▷ Try to prove.}
5:         if proven then return true  \textbf{▷ Property proved.}
6:         $\langle M', \text{refined} \rangle \leftarrow \text{Refine}(M', \Phi, \alpha)$  \textbf{▷ Try to refine.}
7:         if $\neg$refined then return false  \textbf{▷ Counter-example is real.}

which is based on the CEGAR approach \cite{68}. Our main idea is to limit the scope of the refinement to the region around a specific counter-example. In the next section, we formally describe our proposed framework and show that it can be extended to other transcendental activation functions (e.g., softmax).

IV. A CEGAR Framework for S-shaped Sigmoid Activations

In this section, we formalize our meta-algorithm for precisely reasoning about DNNs with S-shaped sigmoid activations, which is based on the CEGAR framework \cite{69}. We first present the general framework and then discuss concrete instantiations of the sub-procedures.

Verification preliminaries. Our procedure operates on tuples of the form $M := (V, \mathcal{X}, \mathcal{Y}, \phi_{\text{aff}}, \phi_{p})$. Here, $V$ is a set of real variables with $\mathcal{X}, \mathcal{Y} \subseteq V$ and $\phi_{\text{aff}}$ and $\phi_{p}$ are sets of formulas over $V$ (when the context is clear, we also use $\phi_{\text{aff}}$ and $\phi_{p}$ to mean the conjuncts of the formulas in those sets). A variable assignment $\alpha : V \mapsto \mathbb{R}$ maps variables in $V$ to real values. We consider properties of the form $\Phi := (\Phi_{\text{in}}(\mathcal{X}) \Rightarrow \Phi_{\text{out}}(\mathcal{Y}))$, where $\Phi_{\text{in}}(\mathcal{X})$ and $\Phi_{\text{out}}(\mathcal{Y})$ are linear arithmetic formulas over $\mathcal{X}$ and $\mathcal{Y}$, and we say that $\Phi$ holds on $M$ if and only if the formula

$$\psi := \phi_{\text{aff}} \land \phi_{p} \land \Phi_{\text{in}}(\mathcal{X}) \land \neg \Phi_{\text{out}}(\mathcal{Y})$$

is unsatisfiable. We use $M \models \Phi$ to denote that $\Phi$ holds ($\psi$ is unsatisfiable), $M[\alpha] \models \neg \Phi$ to denote that $\Phi$ does not hold and is falsified by $\alpha$ ($\psi$ can be satisfied with assignment $\alpha$), and $M[\alpha] \models \Phi$ to denote that $\Phi$ is not falsified by $\alpha$ ($\alpha$ does not satisfy $\psi$). Given this notation, we define a sound abstraction as follows:

Definition IV.1 (Sound abstraction). Given a tuple $M := (V, \mathcal{X}, \mathcal{Y}, \phi_{\text{aff}}, \phi_{p})$ and a property $\Phi := (\Phi_{\text{in}}(\mathcal{X}) \Rightarrow \Phi_{\text{out}}(\mathcal{Y}))$, we say the tuple $M' := (V', \mathcal{X}, \mathcal{Y}, \phi_{\text{aff}}, \phi_{p})$ is a sound abstraction of $M$ if $M' \models \Phi$ implies that $M \models \Phi$.

Verifying DNNs. Given a DNN $f$, we construct a tuple $M_f$ as follows: for each layer $i$ in $f$, we let $v^{(i)}$ be a vector of $d_i$ variables representing the pre-activation values in layer $i$, and let $\hat{v}^{(i)}$ be a similar vector representing the post-activation values in layer $i$. Let $\mathcal{V}^{(0)}$ be a vector of $n_0$ variables from $\mathcal{V}$ representing the inputs. Then, let $V$ be the union of all these variables, and let $\mathcal{X}$ and $\mathcal{Y}$ be the input and output variables, respectively; that is, $\mathcal{X}$ consists of the variables in $\psi^{(0)}$, and $\mathcal{Y}$ contains the variables in $v^{(L)}$. $\phi_{\text{aff}}$ and $\phi_{p}$ capture the affine and non-linear (i.e., activation) transformations in the neural network, respectively. In particular, for each layer $i$, $\phi_{\text{aff}}$ contains the formulas $v^{(i)} = W^{(i)} \psi^{(i-1)} + b^{(i)}$, and $\phi_{p}$ contains the formulas $\psi^{(i)} = \rho(v^{(i)})$.

Algorithm \cite{1} presents a high-level CEGAR loop for checking whether $M \models \Phi$. It is parameterized by three functions. The Abstract function produces an initial sound abstraction of $M$. The Prove function checks whether $M' \models \Phi$. If so (i.e., the property $\Phi$ holds for $M'$), it returns with proven set to true. Otherwise, it returns an assignment $\alpha$ which constitutes a counter-example. The final function is Refine, which takes $M$ and $M'$, the property $\Phi$, and the counterexample $\alpha$ for $M'$ as inputs. Its job is to refine the abstraction until $\alpha$ is no longer a counter-example. If it succeeds, it returns a new sound abstraction $M'$. It fails if $\alpha$ is an actual counter-example for the original $M$. In this case, it sets the return value refined to false. Throughout its execution, the algorithm maintains a sound abstraction of $M$ and checks whether the property $\Phi$ holds on the abstraction. If a counter-example $\alpha$ is found such that $M'[\alpha] \models \neg \Phi$, the algorithm uses it to refine the abstraction so that $\alpha$ is no longer a counter-example. The following theorem follows directly from Def. IV.1.

Theorem IV.2 (CEGAR is sound). Algorithm \cite{7} returns true only if $M \models \Phi$.

A. Choice of the underlying verifier and initial abstraction

The Prove function can be instantiated with an existing DNN verifier. The verifier is required to (1) handle piecewise-linear constraints; and (2) produce counter-examples. There are many existing verifiers that meet these requirements \cite{14,31,44,51}. To ensure that these two requirements are sufficient, we also require that $\phi_{\text{aff}}$ and $\phi_{p}$ only contain linear and piecewise-linear formulas.

The Abstract function creates an initial abstraction. For simplicity, we assume that all piecewise-linear formulas are unchanged by the abstraction function. For S-shaped sigmoid activations, we use piecewise-linear over-approximations. In principle, any sound piecewise-linear over-approximation of the S-shaped sigmoid function could be used. One approach is to use a fine-grained over-approximation with piecewise-linear bounds \cite{70}. While this approach can arbitrarily reduce over-approximation error, it might easily lead to an explosion of the state space when reasoning about generative models due to the large number of transcendental activations (equal to the dimension of the generated image) present in the system. One key insight of CEGAR is that it is often the case that most of the complexity of the original system is unnecessary for proving the property and eagerly adding it upfront only increases the computational cost. We thus propose starting with a coarse (e.g., convex) over-approximation and only refining with additional piecewise-linear constraints when necessary. Suitable candidates for the initial abstraction of a S-shaped sigmoid function include the abstraction proposed in
which considers the convex relaxation of the S-shaped sigmoid activation.

B. Abstraction Refinement for the S-shaped sigmoid activation function

We now focus on the problem of abstraction refinement for models with S-shaped sigmoid activation functions. Suppose that an assignment \( \alpha \) is found by Prove such that \( M'[\alpha] = \neg \Phi \), but for some neuron with S-shaped sigmoid activation \( \rho \), represented by variables \((v, \hat{v}), \alpha(\hat{v}) \neq \rho(\alpha(v))\). The goal is to refine the abstraction \( M' \), so that \( \alpha \) is no longer a counter-example for the refined model. Here we present a refinement strategy that is applicable to any sound abstraction of the S-shaped sigmoid functions. We propose using two linear segments to exclude spurious counter-examples. The key insight is that this is always sufficient for ruling out any counter-example. We assume that \( \phi_{\text{aff}} \) includes upper and lower bounds for each variable \( v \) that is an input to an S-shaped sigmoid function. In practice, bounds can be computed with bound-propagation techniques [31, 34, 55].

**Lemma IV.3.** Given an interval \((l, u)\), an S-shaped sigmoid function \( \rho \), and a point \((p, q) \) \( \in \mathbb{R}^2 \), where \( p \in (l, u) \) and \( q \neq \rho(p) \), there exists a piecewise-linear function \( h : \mathbb{R} \rightarrow \mathbb{R} \) that 1) has two linear segments; 2) evaluates to \( \rho(p) \) at \( p \); and 3) separates \( \{(p, q)\} \) and \( \{(x, y) | x \in (l, u) \wedge y = \rho(x)\} \).

Leveraging this observation, given a point \((p, q) = (\alpha(v), \alpha(\hat{v}))\), we can construct a piecewise-linear function \( h \) of the following form:

\[
h(x) = \rho(p) + \begin{cases} 
\beta(x - p) & \text{if } x \leq p \\
\gamma(x - p) & \text{if } x > p 
\end{cases}
\]

that separates the counter-example and the S-shaped sigmoid function. If \( q > \rho(p) \), we add the formula \( \hat{v} \leq h(v) \) to the abstraction. And if \( q < \rho(p) \), we add \( \hat{v} \geq h(v) \).

The values for the slopes \( \beta \) and \( \gamma \) should ideally be chosen to minimize the over-approximation error while maintaining soundness. Additionally, they should be easily computable. Table I presents a general recipe for choosing \( \beta \) and \( \gamma \) when the spurious counter-example point is below the S-shaped sigmoid function. Choosing \( \beta \) and \( \gamma \) when the counter-example is above the S-shaped sigmoid function is symmetric (details are shown in App. ??). \( \eta \) denotes the inflection point of the S-shaped sigmoid function.

Note that in case 5, \( \beta \) is the same as \( \gamma \), meaning that a linear bound (the tangent line to \( \rho \) at \( p \)) suffices to exclude the counter-example. In terms of optimality, all but the \( \gamma \) value in case 1 and the \( \beta \) value in case 3 maximally reduce the over-approximation error among all valid slopes at \((p, \rho(p))\). In those two cases, linear or binary search techniques [34, 51] could be applied to compute better slopes, but the formulas shown give the best approximations we could find without using search.

**Lemma IV.4 (Soundness of slopes).** Choosing \( \beta \) and \( \gamma \) using the recipe shown in Table \ref{table:refine} results in a piecewise-linear function \( h \) that satisfies the conditions of Lemma IV.3.

---

### Algorithm 2 Refine

```plaintext
1: refined ← 0
2: for \((v, \hat{v}) \in \text{AllSigmoidal}(V')\) do
3:   if \( \alpha(\hat{v}) = \rho(\alpha(v)) \) then continue \hfill \# Skip satisfied activations.
4:   refined ← refined + 1
5:   \( \beta, \gamma \leftarrow \text{getSlopes}(l(v), u(v), \alpha(v), \alpha(\hat{v})) \) \# Compute slopes.
6:   \( \phi'_{\text{aff}} \leftarrow \phi_{\text{aff}} \cup \text{addPLBound}(\beta, \gamma, v, \hat{v}, \alpha) \) \hfill \# Refine the abstraction.
7:   if stopCondition(refined) then break \hfill \# Check termination condition.
8: return \( \langle V', X, Y, \phi'_{\text{aff}}, \phi'_{\rho}, \Phi, \alpha : Y \rightarrow X \rangle \)
```

---

An instantiation of the Refine function for neural networks with S-shaped sigmoid activation is shown in Alg. 2. It iterates through each S-shaped sigmoid activation function. For the ones that are violated by the current assignment, the algorithm computes the slopes following the strategy outlined above with the getSlopes function and adds the corresponding piecewise-linear bounds (e.g., \( \hat{v} \geq h(v) \)) with the addPLBound function. Finally, we also allow the flexibility to terminate the refinement early with a customized stopCondition function. This is likely desirable in practice, as introducing a piecewise-linear bound for each violated activation might be too aggressive. Furthermore, adding a single piecewise-linear bound already suffices to exclude \( \alpha \). We use an adaptive stopping strategy where we allow at most \( m \) piecewise-linear bounds to be added in the first invocation of Alg. 2. And then, in each subsequent round, this number is increased by a factor \( k \). For our evaluation, below, we used \( m = 30 \) and \( k = 2 \), which were the values that performed best in an empirical analysis.

**Theorem IV.5 (Soundness of refinement).** Given a sound abstraction \( M' \) of tuple \( M \), a property \( \Phi \), and a spurious counter-example \( \alpha \) s.t. \( M'[\alpha] = \neg \Phi \) and \( M[\alpha] = \Phi \), Alg. 2 produces a sound abstraction of \( M \), \( M'' \), s.t. \( M''[\alpha] = \Phi \).

V. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our proposed verification framework. We begin by investigating the effectiveness of our CEGAR-based approach on boosting the verification accuracy of existing approaches based on one-shot over-approximation. We evaluate on both robustness queries on real-world distribution shifts (§V-A) and existing benchmarks on robustness against norm-bounded perturbations [66, 71] (§V-C). Next, we benchmark the performance of our verifier on a range of challenging distribution shifts (§V-D). Finally, we use our method to show that robust training tends to result in higher levels of certified robustness against distribution shifts (§V-E).

**Datasets for distribution shifts.** We consider a diverse array of distribution shifts on the MNIST [72] and CIFAR-10 [73] datasets. The code used to generate the perturbations is adapted from [10].
Training algorithms. For each distribution shift we consider, we train a CVAE using the framework outlined in [9]. All generators use sigmoid activations in the output layer and ReLU activations in the hidden layers. This is a typical architecture for generative models. For each dataset, the number of sigmoid activations used in the CVAE is the same as the (flattened) output dimension; that is, 784 (28 × 28) sigmoids for MNIST and 3072 (3 × 32 × 32) sigmoids for CIFAR-10. Throughout this section, we use various training algorithms, including empirical risk minimization (ERM) [74], invariant risk minimization (IRM) [75], projected gradient descent (PGD) [2], and model-based dataset augmentation (MDA) [8].

Implementation details. We use the DeepPoly/CROWN [31, 34] method, which over-approximates the sigmoid output with two linear inequalities (an upper and a lower bound), to obtain an initial abstraction (the Abstract function in Alg. 1) for each sigmoid and instantiate the Prove function with the Marabou neural network verification tool. [14, 17] All experiments are run on a cluster equipped with Intel Xeon E5-2637 v4 CPUs running Ubuntu 16.04 with 8 threads and 32GB memory.

A. Evaluation of CEGAR-based verification procedure

We first compare the performance of our proposed CEGAR procedure to other baseline verifiers that do not perform abstraction refinement. To do so, we compute the largest perturbation δ that each verifier can certify. In our comparison, we consider three distinct configurations: (1) DP, which runs the DeepPoly/CROWN abstract interpretation procedure without any search; (2) DP+BaB, which runs a branch-and-bound procedure (Marabou) on an encoding where each sigmoid is abstracted with the DeepPoly/CROWN linear bounds and the other parts are precisely encoded; and (3) DP+CEGAR, which is the CEGAR method proposed in this work. [7] For each verifier, we perform a linear search for the largest perturbation bound each configuration can certify. Specifically, starting from δ = 0, we repeatedly increase δ by 0.02 and check whether the configuration can prove robustness with respect to S(x) within a given time budget (20 minutes). The process terminates when a verifier fails to prove or disprove a given specification.

For this experiment, we consider the shear distribution shift on MNIST and the fog distribution shift on the CIFAR-10 dataset (see Figure 2). All classifiers are trained using ERM. To provide a thorough evaluation, we consider several generator and classifier architectures; details can be found in App. D. Our results are enumerated in Table I, which shows the mean and standard deviation of the largest δ each configuration is able to prove for the first 100 correctly classified test images. We also report the average runtime on the largest δ proven robust by DP+BaB and DP+CEGAR, as well as the average number of abstraction refinement rounds by DP+CEGAR on those δ values. Across all configurations, our proposed technique effectively improves the verifiable perturbation bound with moderate runtime overhead. This suggests that the counter-example guided abstraction refinement scheme can successfully boost the precision when reasoning about sigmoid activations by leveraging existing verifiers.

B. Effect of hyper-parameters m and k

The refinement procedure (Alg. 2) has two numerical hyper-parameters, m and k, which together control the number of new bounds introduced at a given refinement round (at round i, at most m × k i new bounds are introduced). If too few new bounds are introduced, then there is not enough refinement and the number of refinement rounds needed to prove the property increases. On the other hand, if too many bounds are introduced, then the solver’s runtime on checking the abstraction might unnecessarily increase, resulting in timeouts.

We note that our framework is general and can be used with other abstractions and solvers. To motivate more efficient ways to encode and check the refined abstraction, we describe in App. E how to encode the added piecewise-linear bounds as LeakyRelUs. This leaves open the possibility to further leveraging neural network verifiers that support LeakyRelUs.

TABLE I: Choice of the slopes of the piece-wise linear abstraction refinement. The figures illustrate the refinement on the sigmoid function.
the largest \( \delta \) (we run all combinations of \( m \) and \( k \) consistently decreases as \( m \) increases. For example, \((50, 2)\) solves one less MNIST benchmark than \((30, 2)\). On the other hand, if the strategy is too “lazy” (e.g., \( m \) is too small), the increased number of refinement rounds can also result in runtime overhead. For example, on the CIFAR10 benchmarks, when \( k = 1.5 \), the average runtime decreases as \( m \) increases, even though all three configurations solve the same number of instances.

Overall, this study suggests that the optimal values of \( m \) and \( k \) vary across different benchmarks and adaptive heuristics for choosing these hyper-parameters is a promising direction to boost the runtime performance of the proposed algorithm.

To study the effect of \( m \) and \( k \) more closely, we evaluate the runtime performance of \( \text{DP+CEGAR} \) instantiated with different combinations of \( m \) and \( k \). In particular, we run all combinations of \( m \in \{10, 30, 50\} \) and \( k \in \{1.5, 2\} \) on the first 10 verification instances for the first two generator-classifier pairs in Table II \((\text{MLP}_G, \text{MLP}_C) \) and \((\text{CONV}_G, \text{CONV}_C) \). The perturbation bound is the largest \( \delta \) proven robust for each instance.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( m )</th>
<th>( k )</th>
<th># solved</th>
<th>time(s)</th>
<th># ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>10</td>
<td>1</td>
<td>139.5</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>1.5</td>
<td>52.4</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>1.5</td>
<td>39.8</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>113.6</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>2</td>
<td>121.0</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>2</td>
<td>64.6</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>CIFAR</td>
<td>10</td>
<td>1.5</td>
<td>77.0</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>1.5</td>
<td>60.4</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>1.5</td>
<td>38.2</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
<td>55.3</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>2</td>
<td>100.7</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>2</td>
<td>82.2</td>
<td>1.6</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE III:** Effect of \( m \) and \( k \) on the runtime performance of CEGAR.

C. Further evaluation of CEGAR on adversarial robustness benchmarks.

To better understand the effectiveness of our abstraction-refinement technique on boosting the verification accuracy over one-shot approximation, we consider a different initial abstraction, PRIMA [66], a more recently proposed abstraction that considers convex relaxations over groups of activation functions and is empirically more precise than DeepPoly/CROWN. In particular, we focus on the same sigmoid benchmarks used in [66]. We use the PRIMA implementation in the artifact associated with the paper and run a configuration \( \text{PRIMA+CEGAR} \) which is the same as \( \text{DP+CEGAR} \), except that we run PRIMA instead of DeepPoly/CROWN for the initial abstraction. Each job is given 16 threads and a 30 minute wall-clock time limit. Table [IV] shows the number of verified instances and the average runtime on verified instances for the two configurations. Our configuration is able to consistently solve more instances with only a moderate increase in average solving time. This suggests that the meta-algorithm that we propose can leverage the tighter bounds derived by existing abstraction-based methods and boost the state of the art in verification accuracy on sigmoid-based networks.

**TABLE IV:** Comparison with PRIMA [66] on the same benchmarks used in [66]

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc.</th>
<th>( \epsilon )</th>
<th>PRIMA robust</th>
<th>time(s)</th>
<th>PRIMA+CEGAR robust</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x100</td>
<td>99</td>
<td>0.015</td>
<td>52</td>
<td>106.5</td>
<td>65</td>
<td>119.5</td>
</tr>
<tr>
<td>9x100</td>
<td>99</td>
<td>0.015</td>
<td>57</td>
<td>136.0</td>
<td>96</td>
<td>323.7</td>
</tr>
<tr>
<td>6x200</td>
<td>99</td>
<td>0.012</td>
<td>65</td>
<td>197.9</td>
<td>75</td>
<td>260.7</td>
</tr>
<tr>
<td>ConvSmall</td>
<td>99</td>
<td>0.014</td>
<td>56</td>
<td>100.5</td>
<td>63</td>
<td>157.8</td>
</tr>
</tbody>
</table>

We have also evaluated our techniques on the sigmoid benchmarks used in VNN-COMP 2021 (there are no sigmoid benchmarks in VNN-COMP 2022), and we are also able to boost the verification precision over other SoTA solvers such as \( \alpha - \beta \)-CROWN and VeriNet [51] with our approach. Details can be found in App. [E]
Interestingly, MDA, which is perturbation-aware, outperforms MLP and verify the targeted robustness of the classifier against the select the first 500 correctly classified images in its dataset perturbation generative models. For each classifier, we then shears distribution shifts on the MNIST dataset. To this end, we classifiers trained using ERM, IRM, PGD, and MDA against the ERM against different perturbations.

We next use our proposed verification procedure to evaluate the robustness of classifiers trained using ERM against a wide range of distribution shifts. We select the first 100 correctly classified test images from the respective dataset for each perturbation and verify the robustness of the classifier against the perturbation set. Three values of the perturbation variable $\delta$ are considered: 0.1, 0.2, and 0.5. The architectures we consider for MNIST are MLP_GEN2 and MLP_CLS3. For CIFAR-10 we use CONV_GEN2 and CONV_CLS2. The verification results are shown in Figure 4. The “robust” columns show the number of instances that our verification procedure is able to certify within a 20 minute timeout. As one would expect, the robustness of each classifier deteriorates as the perturbation budget $\delta$ increases. For instance, for the shear transformation, the classifier is robust on 76 out of the 100 instances when $\delta = 0.1$, but is only certified robust on 4 instances when $\delta$ increases to 0.2. Information like this could help system developers to identify perturbation classes for which the network is especially vulnerable and potentially retrain the network accordingly.

### D. Benchmarking our approach on an array of real-world distribution shifts

We next use our proposed verification procedure to evaluate the robustness of classifiers trained using ERM against a wide range of distribution shifts. We select the first 100 correctly classified test images from the respective dataset for each perturbation and verify the robustness of the classifier against the perturbation set. Three values of the perturbation variable $\delta$ are considered: 0.1, 0.2, and 0.5. The architectures we consider for MNIST are MLP_GEN2 and MLP_CLS3. For CIFAR-10 we use CONV_GEN2 and CONV_CLS2. The verification results are shown in Figure 4. The “robust” columns show the number of instances that our verification procedure is able to certify within a 20 minute timeout. As one would expect, the robustness of each classifier deteriorates as the perturbation budget $\delta$ increases. For instance, for the shear transformation, the classifier is robust on 76 out of the 100 instances when $\delta = 0.1$, but is only certified robust on 4 instances when $\delta$ increases to 0.2. Information like this could help system developers to identify perturbation classes for which the network is especially vulnerable and potentially retrain the network accordingly.

### E. Verification for various robust training algorithms

Finally, we compare the robustness and accuracy of classifiers trained using ERM, IRM, PGD, and MDA against the shear distribution shifts on the MNIST dataset. To this end, we measure the accuracy on the entire test set under the learned perturbation generative models. For each classifier, we then select the first 500 correctly classified images in its dataset and verify the targeted robustness of the classifier against the perturbation. The architectures we use are MLP_GEN2 and MLP_CLS3.

Accuracy and robustness results are presented in Figure 5. Interestingly, MDA, which is perturbation-aware, outperforms the other three perturbation-agnostic training methods, on both test accuracy and robustness, suggesting that knowing what type of perturbation to anticipate is highly useful. Notice that accuracy on the perturbation set is not necessarily a good proxy for robustness: while the IRM-trained classifier has similar accuracy as the PGD-trained classifier, the former is significantly less robust on the perturbation set with $\delta = 0.1$. This further supports the need for including formal verification in the systematic evaluation of neural networks and training algorithms.

### VI. Related Work

**Beyond norm-bounded perturbations.** While the literature concerning DNN verification has predominantly focused on robustness against norm-bounded perturbations, some work has considered other forms of robustness, e.g., against geometric transformations of data [53–55]. However, the perturbations considered are hand-crafted and can be analytically defined by simple models. In contrast, our goal is to verify against real-world distribution shifts that are defined via the output set of a generative model. Our approach also complements recent work which has sought to incorporate neural symbolic components into formal specifications [58].

**Existing verification approaches.** Existing DNN verification algorithms broadly fall into one of two categories: search-based methods [13–30] and abstraction-based methods [31–50]. While several existing solvers can handle sigmoid activation functions [18, 31, 34, 51, 66], they rely on one-shot abstraction and lack a refinement scheme for continuous progress. On the other hand, a separate line of work has shown that verifying DNNs containing a single layer of logistic activations is decidable [76], but the decision procedure proposed in this work is computationally prohibitive. To overcome these limitations, we propose a meta-algorithm inspired by counter-example-guided abstraction refinement [68] that leverages existing verifiers to solve increasingly refined abstractions. We notice a concurrent work [77] on formal reasoning of example-guided abstraction refinement that leverages existing verifiers to solve increasingly refined abstractions.

CEGAR [68] is a classical idea. Our main contribution w.r.t. CEGAR lies in investigating the choices of its parameters (i.e., Abstract, Prove, and Refine) when it comes to verifying neural networks with S-shaped activations.
Verification against distribution shifts. The authors of [9] also considered the task of evaluating DNN robustness to real-world distribution shifts; in particular, the approach used in [9] relies on randomized smoothing [78]. This scheme provides probabilistic guarantees on robustness, whereas our approach (as well as the aforementioned approaches) provides deterministic guarantees. In a separate line of work, several authors have sought to perform verification of deep generative models [56, 57]. However, each of these works assumes that generative models are piece-wise linear functions, which precludes the use of state-of-the-art models.

VII. Conclusion

In this paper, we presented a framework for certifying robustness against real-world distribution shifts. We proposed using provably trained deep generative models to define formal specifications and a new abstraction-refinement algorithm for verifying them. Experiments show that our method can certify against larger perturbation sets than previous techniques.

Limitations. We now discuss some limitations of our framework. First, like many verification tools, the classifier architectures that our approach can verify are smaller than popular architectures such as ResNet [79] and DenseNet [80]. This stems from the limited capacity of existing DNN verification tools for piecewise-linear functions, which we invoke in the CEGAR loop. Given the rapid growth of the DNN verification community, we are optimistic that the scalability of verifiers will continue to grow rapidly, enabling their use on larger and larger networks. Another limitation is that the quality of our neural symbolic specification is determined by how well the generative model captures real-world distribution shifts. The mismatch between formal specification and reality is in fact common (and often unavoidable) in formal verification. And while [9] shows that under favorable conditions, CVAEs can capture distribution shifts, these assumptions may not hold in practice. For this reason, we envision that in addition to these theoretical results, a necessary future direction will be to involve humans in the verification loop to validate the shifts captured by generative models and the produced counterexamples. This resembles how verification teams work closely with product teams to continually re-evaluate and adjust the specifications in existing industrial settings. In general, we believe that closing this verification loop (verify, debug, verify again, etc.) is a very interesting future research direction. Finally, when applying our techniques to real industrial settings, another challenge is that collecting training data corresponding to a new distribution shifts may be costly (e.g., collecting the same street view under different times of day). This cost would mostly likely pay off in safety-critical domains.

REFERENCES


[69] M. Mann, A. Irfan, A. Griggio, O. Padon, and C. Barrett, “Counterexample-guided prophecy for model checking modulo the theory of arrays,” in International Confer-


APPENDIX A

CHOICES OF SLOPES (CONT.)

We present the general recipe of choosing \( \beta \) and \( \gamma \) in the case when the violation point is above the \( \text{S-shaped sigmoid} \) function.

APPENDIX B

Proofs

Proof. Lemma [IV.3] This can be proved by construction using the \( \beta \) and \( \gamma \) values in Table [I] and Table [V]. We next prove that those choices are sound in Lemma [IV.4].

Before proving Lemma [IV.4] we first state the following definitions and facts. \( ^5 \)

Definition B.1 (Tangent line). The tangent line at \( a \) to the function \( f \), denoted with \( \text{TanLine}_{f,a}(x) \), is defined as:

\[
\text{TanLine}_{f,a}(x) = f(a) + f'(a)(x-a).
\]

Definition B.2 (Secant line). Definition 2.2. Given \( a,b \in \mathbb{R} \), the secant line at \( [a,b] \) to a function \( f \), denoted with \( \text{SecLine}_{f,a,b}(x) \), is defined as:

\[
\text{SecLine}_{f,a,b}(x) = \frac{f(a)-f(b)}{a-b} \cdot (x-a) + f(a).
\]

Proposition B.3. Let \( f \) be a twice differentiable univariate function. If \( f''(x) \geq 0 \) for all \( x \in [l,u] \), then for all \( a, x \in [l,u] \), \( \text{TanLine}_{f,a}(x) \leq f(x) \), and for all \( a, b, x \in [l,u] \), where \( a < b \) and \( a \leq x \leq b \), \( \text{SecLine}_{f,a,b}(x) \geq f(x) \).

Proposition B.4. Let \( f \) be a twice differentiable univariate function. If \( f''(x) \leq 0 \) for all \( x \in [l,u] \), then for all \( a, x \in [l,u] \), \( \text{TanLine}_{f,a}(x) \geq f(x) \), and for all \( a, b, x \in [l,u] \), where \( a < b \) and \( a \leq x \leq b \), \( \text{SecLine}_{f,a,b}(x) \leq f(x) \).

Proposition B.5. Let \( f \) be a differentiable univariate function with non-negative derivative at each point. For all \( \gamma \) if \( \gamma \leq f'(x) \) for all \( x \in [l,u] \), then \( f(l) + \gamma(x-l) \leq f(x) \) for all \( x \in [l,u] \).

Proof. Lemma [IV.4] Cond. 1 and Cond. 2 hold trivially. Since \( q < h(p) \), for Cond. 3, it suffices to show that \( \forall (m,n) \in \{ (x,y) \mid x \in [l,u] \land y = \rho(x) \} \), \( n \geq h(m) \).

More concretely, we show that \( a \), \( n \geq \rho(p) + \beta(m-p) \) for all \( m \in [l,p) \), and \( b \), \( n \geq \rho(p) + \gamma(m-p) \) for all \( m \in (p,u] \). We prove this is true for each case in Table [I].

- **Case 1:** The segment corresponding to \( \beta \) is \( \text{TanLine}_{p,p} \), and Cond. a holds by Prop. [B.3]. On the other hand, the choice \( \gamma \) is such that \( \gamma \leq \rho'(x) \), for all \( x \in [p,u] \). Thus Cond. b holds by Prop. [B.5].

- **Case 2:** The segment corresponding to \( \beta \) is \( \text{TanLine}_{p,p} \), and Cond. a holds by Prop. [B.3]. The segment corresponding to \( \gamma \) is \( \text{SecLine}_{p,p,u} \), and Cond. b holds by Prop. [B.4].

- **Case 3:** For Cond. a, we further break it into 2 cases: \( m \leq \eta \) and \( m > \eta \). In the former case, the line \( \rho(p) + \beta(m-p) \) is below the line \( \rho(l) + \min(\rho'(l), \rho'(u))(x-l) \), which by Prop. [B.5] is below \( \rho \). When \( m > \eta \), \( \rho(p) + \beta(m-p) \) is below the secant line \( \text{SecLine}_{p,p,u} \) which by Prop. [B.4] is below \( \rho \). On the other hand, the segment corresponding to \( \gamma \) is \( \text{SecLine}_{p,p,u} \), and Cond. b holds by Prop. [B.4].

- **Case 4:** The segments are both secant lines, \( \text{SecLine}_{p,l,p} \) and \( \text{SecLine}_{p,p,u} \), thus the conditions hold by Prop. [B.4].

\( ^5 \)These are partially adapted from “Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions” by Cimatti et al. We will cite the paper in the revised version of the paper.
• **Case 5:** The segments are both tangent lines, \( \text{TanLine}_{p,p} \), thus the conditions hold by Prop. B.3.

The proof of the soundness of the slope choices in Table V.

**Proof. Theorem IV.5** We can prove the soundness of \( M'' \) by induction on the number of invocations of the `ADDPLBOUND` method. If it is never invoked, then \( M'' = M' \) which is a sound abstraction. In the inductive case, it follows from Lemma IV.4 that adding an additional piecewise-linear bound does not exclude variable assignments that respect the precise sigmoid function. On the other hand, when \( M[\alpha] \models \Phi \), the `ADDPLBOUND` method will be invoked at least once, which precludes \( \alpha \) as a counter-example with respect to \( M'' \). That is, \( M''[\alpha] \models \Phi \).

**APPENDIX C**

**ENCODING PIECE-WISE LINEAR REFINEMENT USING LEAKYReLU**

We observe that it is possible to encode the piecewise-linear bounds that we add during abstraction refinement using LeakyReLU functions. While we do not leverage this fact, we lay out the reduction to LeakyReLU in this section and leave it as future work to further leverage verification tools supporting LeakyReLUs.

A LeakyReLU \( r_{\alpha} \) is a piecewise linear function with two linear segments:

\[
r_{\alpha}(x) = \begin{cases} 
\alpha \cdot x & \text{if } x \leq 0 \\
x & \text{if } x > 0 
\end{cases}
\]

where \( \alpha \geq 0 \) is a hyper-parameter.

Given a piecewise linear function with two linear segments:

\[
h(x) - \rho(p) = \begin{cases} 
\beta(x - p) & \text{if } x \leq p \\
\gamma(x - p) & \text{if } x > p 
\end{cases}
\]

We can rewrite \( h \) as the following:

\[
h(x) = \gamma \cdot r_{\alpha}(x - p) + \rho(p), \text{ where } \alpha := \frac{\beta}{\gamma}
\]

Note that the \( \alpha \) value is always valid (\( \geq 0 \)) because both \( \beta \) and \( \gamma \) that we choose are positive. This means that we can potentially encode the piecewise linear bounds as affine and leaky relu layers. For example, the piecewise linear upper bound \( y \leq h(x) \) for a sigmoid \( y = \rho(x) \) can be encoded as

\[
a_1 = x - p \\
a_2 = r_{\alpha}(a_1) \\
a_3 = a_2 + \rho(p) \\
y = a_3 + a_4 \\
a_4 \leq 0
\]

where \( a_1, a_2, a_3, a_4 \) are fresh auxiliary variables. Eqs. a) and c) can be modeled by feed-forward layers. Eq. b) can be modeled by a leaky relu layer. If we treat \( a_4 \) as an input variable to the neural network, Eq. d) can be modeled as a residual connection. This suggest that we could in principle express the abstraction as an actual piecewise-linear neural networks (with bounds on the input variables e.g., \( a_4 \)), making it possible to leverage verifiers built on top of Tensorflow/Pytorch.

**APPENDIX D**

**DETAILS ON TRAINING AND CVAEs**

A. **Dataset**

We consider the well-known MNIST and CIFAR-10 dataset. The MNIST dataset contains 70,000 grayscale images of handwritten digits with dimensions \( 28 \times 28 \), where we used 60,000 images for training and held 10,000 for testing. The CIFAR-10 dataset contains 60,000 colored images of 10 classes with dimensions \( 3 \times 32 \times 32 \), where we used 50,000 images for training and held 10,000 for testing.

To perturb the images, we adapt the perturbations implemented in [10]. When training and testing the models, we sample images from the dataset and randomly perturb each image with a strength parameter \( c \) that is sampled uniformly from the ranges given in Table IV.

The code for our experiments is available at [https://github.com/hendrycks/robustness/blob/master/ImageNet-C/create_0/make_cifar_c.py](https://github.com/hendrycks/robustness/blob/master/ImageNet-C/create_0/make_cifar_c.py)
**B. Architecture**

On each dataset, we train a conditional variational encoder (CVAE) with three components: prior network, encoder network, and a decoder network (generator). We also train a set of classifiers. In this section, we detail the architecture of these networks. The architecture of MNIST networks are shown in Tables VII[XIII]. Those of the CIFAR networks are shown in Tables XIV[XIX]. The output layers of the generators are activated with sigmoid function. All hidden-layers use ReLU activations.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Perturbation</th>
<th>Range of $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>brightness</td>
<td>$[0.0, 0.5]$</td>
</tr>
<tr>
<td></td>
<td>rotation</td>
<td>$[-60, 60]$</td>
</tr>
<tr>
<td></td>
<td>gaussian blur</td>
<td>$[-15, 75]$</td>
</tr>
<tr>
<td></td>
<td>shear</td>
<td>$[1.0, 1.0]$</td>
</tr>
<tr>
<td></td>
<td>contrast</td>
<td>$[0.0, 0.4]$</td>
</tr>
<tr>
<td></td>
<td>translate</td>
<td>$[1.0, 5.0]$</td>
</tr>
<tr>
<td></td>
<td>scale</td>
<td>$[0.5, 0.9]$</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>brightness</td>
<td>$[0.05, 0.3]$</td>
</tr>
<tr>
<td></td>
<td>contrast</td>
<td>$[-1.5, 1.5]$</td>
</tr>
<tr>
<td></td>
<td>gaussian blur</td>
<td>$[-4, 1]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters/Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$28 \times 28$</td>
</tr>
<tr>
<td>Dense</td>
<td>$784 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$300 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$8 \times 2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters/Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$28 \times 28 + 8$</td>
</tr>
<tr>
<td>Dense</td>
<td>$200 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$784 \times 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters/Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$28 \times 28$</td>
</tr>
<tr>
<td>Dense</td>
<td>$32 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$32 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$10 \times 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters/Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$28 \times 28$</td>
</tr>
<tr>
<td>Dense</td>
<td>$64 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$32 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$10 \times 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameters/Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$32 \times 32 \times 3$</td>
</tr>
<tr>
<td>Dense</td>
<td>$3072 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$300 \times 1$</td>
</tr>
<tr>
<td>Dense</td>
<td>$8 \times 2$</td>
</tr>
</tbody>
</table>

**C. Optimization**

We implement our models and training in PyTorch. The CVAE implementation is adapted from that in [9]. On both datasets, we trained our CVAE networks for 150 epochs using the ADAM optimizer with a learning rate of $10^{-4}$ and forgetting factors of 0.9 and 0.999. In addition, we applied cosine annealing learning rate scheduling. Similar to [3], we increase $\beta$ linearly from $\beta = 0$ at epoch 1 to $\beta = 0.01$ at epoch 40, before keeping $\beta = 0.01$ for the remaining epochs. We use a batch size of 256.

The ERM classifiers on the MNIST dataset are trained with the ADAM optimizer with a learning rate of $10^{-3}$ for 20 epochs. The classifiers for the CIFAR-10 dataset are trained with the ADAM optimizer with learning rate $10^{-3}$ for 200 epochs. The classifiers in Sec. V-E are trained also all trained with the ADAM optimizer with learning rate $10^{-2}$ for 20 epochs. For PGD, we use a step size of $\alpha = 0.1$, a perturbation budget of $\epsilon = 0.3$, and we use 7 steps of projected gradient ascent. For IRM, we use a small held-out validation set to select $\lambda \in \{0.1, 1, 10, 100, 1000\}$. For MDA, a step size of $\alpha = 0.1$, a perturbation budget of $\epsilon = 1.0$, and we use 10 steps of projected gradient ascent.

**D. Computing resources**

The classifiers used in Subsec. V-E were trained using a single NVIDIA RTX 5000 GPU. The other networks were trained using 8 AMD Ryzen 7 2700 Eight-Core Processors.
where a counter-example can be found using PGD attack, and evaluate on the remaining 35 benchmarks. In particular, we run a sequential portfolio approach where we first attempt to solve the query with $\alpha$-$\beta$-CROWN \cite{18, 34, 44} (competition version), and if the problem is not solved, we run PRIMA+CEGAR. Table IV shows the results. As a point of comparison, we also report the numbers of the top three performing tools \cite{18, 31, 44, 51, 66} during VNN-COMP-21 on these benchmarks. While $\alpha$-$\beta$-CROWN is already able to solve 29 of the 35 benchmarks, with the abstraction refinement scheme, we are able to solve 1 additional benchmark. We note that during the competition $\alpha$-$\beta$-crown did not use up the 5 minute per-instance timeout limitation on any of these benchmarks. This suggests that the solver was not able to make further progress once the analysis is inconclusive on the one-shot abstraction of the sigmoid activations. On the other hand, our technique provides a viable way to make continuous progress if the verification attempt by the state-of-the-art fails.

APPENDIX F
EVALUATION OF AN EAGER REFINEMENT STRATEGY

We also compare the lazy abstraction refinement strategy with an eager approach where piecewise-linear bounds are added for each sigmoid from the beginning instead of added lazily as guided by counter-examples. In particular, we attempt to add one piecewise-linear upper-bound and one piecewise-linear lower-bound with $K$ linear segments for each sigmoid activations. The segment points are even along the x-axis. We try 2 and 3 for the value of $K$. We than evaluate on the same MNIST benchmarks as in Table II. The results are shown in Table XXI. While the two strategies are still able to improve upon the perturbation bounds found by the pure abstract-interpretation-based approach DP, the means of the largest certified $\delta$ are significantly smaller than those of the CEGAR-based configuration we propose. Interestingly, while $K=3$ uses a finer-grained over-approximation compared with $K=2$, the bounds the former can certify is only larger on one of the six benchmark sets. This suggests that the finer-grained abstraction increases the overhead to the solver and is overall not effective at excluding spurious counter-examples on the set of benchmarks that we consider, which supports the need of a more informed abstraction refinement strategy such as the one we propose.

APPENDIX G
LICENSES

The MNIST and CIFAR-10 datasets are under The MIT License (MIT). The Marabou verification tool is under the terms of the modified BSD license (https://github.com/NeuralNetworkVerification/Marabou/blob/master/COPYING).
TABLE XX: Comparison on the VNN-COMP-21 benchmarks

<table>
<thead>
<tr>
<th>Model</th>
<th># Bench.</th>
<th>α-β-CROWN</th>
<th>VeriNet</th>
<th>ERAN</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x200</td>
<td>35</td>
<td>29</td>
<td>12.9</td>
<td>20</td>
<td>2.5</td>
</tr>
</tbody>
</table>

TABLE XXI: Evaluation results of the eager approach. We also report again the results of DP+CEGAR, which is the same as Table II.