Boosting DeepPoly with Abstraction Refinement for Neural Network Verification

Anonymous Author(s)

ABSTRACT

Neural networks have been increasingly applied in safety-critical applications (such as self-driving cars and medical diagnosis). Thus it is often necessary to formally verify their correctness against desired properties. Recently proposed approaches apply techniques such as abstract interpretation to tackle the problem. Among the proposed abstract domains, DeepPoly is shown to be both effective (e.g., supporting a range of activation functions) and efficient. However, one limitation of DeepPoly (as well as other abstract domains) is over-approximation may introduce false alarms, which prevent us from verifying otherwise valid properties. To overcome this limitation, we propose a lightweight approach to refine DeepPoly's abstraction whenever it is necessary. Intuitively, our approach identifies the neuron which is likely the most responsible for the false alarm and refines its abstraction. This abstraction refinement process may iterate multiple times until a termination condition is met. The experiment results on verifying local robustness show that our approach can (sometimes significantly) increase the verified robustness radiuses of 52% images on MNIST networks and 44% images on CIFAR10 networks compared to DeepPoly.

ACM Reference Format:
Anonymous Author(s). 2021. Boosting DeepPoly with Abstraction Refinement for Neural Network Verification. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recently, neural networks have inevitably expanded their application domains to those safety-critical systems such as self-driving cars [2] and medical diagnosis [15]. It is thus important to formally verify their correctness against desired properties of such neural networks. Towards this goal, the research community has identified and formalized several interesting properties for neural networks, such as robustness [9] and fairness [13], and proposed ever-more sophisticated methods to verify them. Intuitively, verifying local robustness requires us to show that a perturbation on an input within a certain norm does not change the prediction outcome, whereas fairness (defined as the absence of individual discriminative instances [32]) requires us to show that a prediction outcome does not change if only certain protected features (e.g., gender and race) of the input are changed.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Multiple methods and tools have been developed to tackle the problem of neural network verification, including constraint-solving based approaches such as Reluplex [14], DLV [12], and Neurify [28], and abstract-interpretation based approaches such as AI^2 [8], ReluVal [29], DeepZ [22], DeepPoly [24], RefineZono [23], and RefinePoly [21]. Among these approaches, abstract-interpretation based approaches are shown to be more scalable and more flexible (e.g., supporting more kinds of activation functions) [24]. Furthermore, different abstract interpretation approaches adopt different abstract domains, among which the DeepPoly abstract domain, which is the basis for DeepPoly (which shares the same name with the abstract domain) and RefinePoly, is shown to be both effective and efficient.

However, DeepPoly (not unlike other abstract domains) suffers from false alarms due to the over-approximation which is applied to every neuron in the network. One natural idea to improve DeepPoly is to refine the abstraction whenever the verification fails. However, the problem is that neural networks often contain many fully-connected neurons (which can be understood as a horrible pile of spaghetti code) and abstraction takes place typically at every neuron (including the hidden layers and the input layer). As a result, there are too many possible ways of refining the abstraction, i.e., exponential in the number of neurons even if only the simple ReLU activation function is applied.

There have been limited attempts on abstraction refinement for neural network verification, such as ReluVal [29] and Neurify [28]. Note that ReluVal and Neurify use a different abstract domain than DeepPoly and only work for neural networks with the ReLU activation function. Given that DeepPoly was designed to handle neural networks with different types of activation functions (i.e., ReLU, sigmoid, and tanh), it is not clear how or whether the above refinement strategy is applicable to DeepPoly.

In this work, we propose an abstraction refinement approach for the DeepPoly abstract domain. Our approach is based on DeepPoly's capability of capturing the violation of the property in the form of a linear expression over any layer of the neurons. Concretely speaking, we start with the ordinary DeepPoly approach for verifying a user-provided neural network against a reachability property. If DeepPoly fails to verify the property, we identify a neuron for refinement by measuring the 'contribution' of every neuron to the failure, which in turn is defined based on the abstraction constructed in DeepPoly. This refinement process can be applied multiple times until a termination condition (such as the property is verified, timeout, or a bound on the number of refinement iterations is reached) is satisfied. Our approach is not limited to the ReLU activation function (i.e., we support sigmoid and tanh). Our approach has been implemented in the Socrates framework [19]. The experiment results show that our approach increases the verified robustness radiuses of 52% images on MNIST networks and 44% images on CIFAR10 networks compared to DeepPoly. We further

show that our approach also improves RefinePoly as well as the verification of fairness.

To summarize, our contributions in this work are as follows. First, we propose an approach for abstraction refinement in the DeepPoly abstract domain for neural network verification. Second, we implement our approach in the Socrates framework and conduct extensive experiments to evaluate its effectiveness.

Organization The rest of this paper is organized as follows. Section 2 reviews preliminary knowledge. Section 3 presents our approach. Section 4 discusses implementation details and evaluation results. Section 5 reviews related work. Section 6 concludes and discusses potential future research directions.

2 PRELIMINARIES

In this section, we provide the background on neural networks and define our problem. Then we introduce the DeepPoly abstract domain and show how to apply it on an illustrative example.

2.1 Problem Definition

In general, a neural network can be viewed as a function $N : \mathbb{R}^p \rightarrow \mathbb{R}^q$ mapping an input $x \in \mathbb{R}^p$ (e.g., images) to an output $y \in \mathbb{R}^q$ (e.g., scores for labels in image classification). In case the network is used for classification, the output vector y is further processed using a labelling function L, which typically returns the index of the maximum value in y. In this work, we focus on feed-forward neural networks that can be organized using a layered architecture. We leave other forms of neural networks (such as recurrent neural networks) to future work.

Among the layers of a neural network, the first layer is the input layer; the last layer is the output layer and the remaining layers are the hidden layers. Normally, each hidden layer applies two transformations to its input, i.e., an affine transformation and an activation function transformation. First, the affine transformation applies an affine operation, i.e., $affine(x) = Ax + B$ where x is the input to the layer; A is a weight matrix and B is a bias vector. Then the activation function transformation applies a non-linear activation function σ neuron-wise. Widely used activation functions include ReLU, sigmoid, and tanh. For simplicity, we always assume that each hidden layer in the network is expanded into two separate layers in a pre-processing step, i.e., one for the affine operation and one for the activation function operation.

Example 2.1. Fig. 1 shows a simple feed-forward neural network with four layers, i.e., an input layer, two hidden layers, and an output layer. Each layer has two neurons. The input layer reads the input; each hidden layer processes its input via an affine transformation and a ReLU activation function; and the output layer yields an output vector. A label is then generated based on the largest value in the output vector. The values on the edges represent the coefficients of the weight matrix and the values beside the neurons are the biases for the affine transformation at each layer. Note that these values are floating-point numbers in practice. Here we use integers to simplify the presentation.

In the following, we fix a neural network N which consists of k layers with index from 0 to $k-1$, and each layer i contains d_i neurons. Each layer i of the network N is a function $f_i : \mathbb{R}^{d_{i-1}} \rightarrow \mathbb{R}^{d_i}$ mapping the output of layer $i-1$ to the output of layer i, and the neural network is a function $N : \mathbb{R}^{d_0} \rightarrow \mathbb{R}^{d_{k-1}}$, where d_0 is the dimension of the input and d_{k-1} is the dimension of the output. That is, $N = f_{k-1} \circ \cdots f_i \circ \cdots \circ f_0$. We assume the input layer only reads in data and does not perform any transformation. We write x to represent the original input vector; x^i to denote the vector obtained after the i-th layer; and x^i_j to denote the j-th value in the vector x^i. For simplicity, we may use y instead of x^{k-1} to denote the output vector, i.e., $y = x^{k-1} = N(x)$.

A verification task is a triple $\{\phi_I\}N\{\phi_O\}$ where ϕ_I and ϕ_O are quantifier-free predicates constituted by free variables as well as a set of pre-defined functions such as the distance function and the labelling function. Note that the free variables in ϕ_I and ϕ_O are implicitly universally quantified. Typically, the free variables represent the input x in ϕ_I. The triple is valid if and only if ϕ_O is satisfied for all inputs x such that $x \vDash \phi_I$. An input which satisfies ϕ_I but violates ϕ_O is referred to as a counterexample. The verification problem is thus: *given a triple $\{\phi_I\}N\{\phi_O\}$, check whether the triple is valid or not.* A verifier for a verification task is sound if the verifier reports the given triple is valid only if it is valid; and it is complete if the verifier always reports the triple is valid if it is valid.

We refer the readers to [19] for the exact syntax of ϕ_I and ϕ_O. Note that the local robustness property (with an infinity norm distance function) can be easily encoded as a verification task. Intuitively, local robustness means that a small perturbation on an input does not change its classification result. Formally, a neural network N is locally robust on a concrete input I with a radius r iff

$$\{||x - I||_\infty \leq r\}N\{L(N(x)) = L(N(I))\}$$

where $||x - x'||_\infty$ is defined as $max(|x_0 - x'_0|, \cdots, |x_n - x'_n|)$ and L is the labelling function which returns the index of the maximum value in a given vector.

Example 2.2. For the network shown in Example 2.1, consider a concrete input $I = (0, 0)$. The output vector given I is $(1, -4)$ and thus the label of I is 0. Suppose we want to verify the local robustness on I against all perturbations within a radius $r = 1$, the verification task can be encoded as follows.

$$\{||x - (0, 0)||_\infty \leq 1\}N\{L(y) = 0\}$$

An equivalent form for the above verification task is as follows.

$$\{-1 \leq x_0 \leq 1 \wedge -1 \leq x_1 \leq 1\}N\{y_0 > y_1\}$$

where we write y_j to denote the j-th value of the output vector.

Although our discussion focuses on local robustness hereafter, our approach works for other properties as well. One example is a (local) fairness property which is defined as follows.

$$\{x_0 \neq I_0 \wedge x_1 = I_1 \wedge \cdots \wedge x_{d_0-1} = I_{d_0-1}\}N\{L(N(x)) = L(N(I))\}$$

assuming the feature at index 0 is the only protected feature. Intuitively, the property requires that an input x which only differs from a concrete input I by some protected features (e.g., x_0) always has the same label as I.

2.2 Abstract Interpretation with DeepPoly

In this work, the primary means of tackling the above-defined verification problem is abstract interpretation [6]. Abstract interpretation

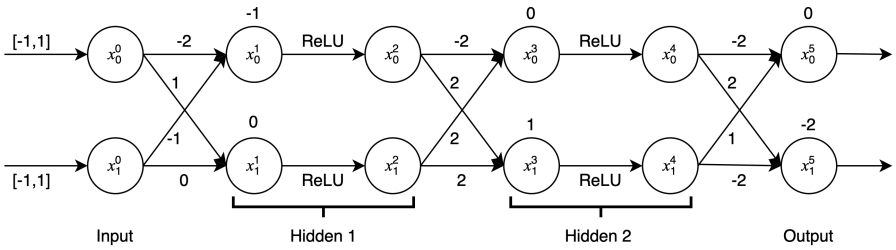


Figure 1: A feed-forward neural network

is a classic static program analysis technique. The idea is to soundly approximate and reason about a program's behaviors without executing it. When abstract interpretation is applied to verify neural networks, several abstract domains have been proposed such as symbolic interval [29], Zonotope [22], and DeepPoly [24], among which DeepPoly is considered the state of the art, as it balances effectiveness and efficiency. The DeepPoly abstract domain assigns to each neuron in the network a variable which represents its abstract state after the abstract transformation. Using our naming convention, we use x_j^i to denote the variable representing the j-th neuron in the i-th layer. The abstract state of each variable x_j^i is in the form of a 4-tuple $(ge_j^i, le_j^i, lw_j^i, up_j^i)$ where ge_j^i and le_j^i are two linear expressions constituted by variables from the previous layer (i.e., layer $i-1$) and lw_j^i and up_j^i are two concrete values (representing a concrete lower bound and upper bound). It is guaranteed that $ge_j^i \leq x_j^i \leq le_j^i$ and $lw_j^i \leq x_j^i \leq up_j^i$. Initially, we have $(ge_j^0, le_j^0, lw_j^0, up_j^0) = (I_j - r, I_j + r, I_j - r, I_j + r)$ for every variable x_j^0 in the input layer based on ϕ_I, i.e., the values of the concrete input I and the radius r[1]. DeepPoly abstract transformers then propagate, layer-by-layer, the abstract states from the input layer to the output layer. Based on the abstract state of the output layer, we may then verify ϕ_O.

In the following, we summarize DeepPoly's abstract transformers and refer the readers to [24] for details as well as the soundness proof. After having the abstract states of all the neurons up to layer i, first we consider the computation of 2 components ge_j^{i+1} and le_j^{i+1} for all neurons j in the layer $i+1$. If the layer $i+1$ is an affine layer, we have $ge_j^{i+1} = le_j^{i+1} = A_j^{i+1} \cdot x^i + B_j^{i+1}$ where A_j^{i+1} is the weight vector of neuron j at layer $i+1$; B_j^{i+1} is its bias; and $\cdot$ is the dot product. If the layer $i+1$ is a ReLU activation function layer, we have the following.

$$ge_j^{i+1} = \begin{cases} 0 & \text{if } up_j^i \leq -lw_j^i \\ x_j^i & \text{otherwise} \end{cases}$$

$$le_j^{i+1} = \begin{cases} 0 & \text{if } up_j^i \leq 0 \\ x_j^i & \text{if } lw_j^i \geq 0 \\ \dfrac{up_j^i \times (x_j^i - lw_j^i)}{(up_j^i - lw_j^i)} & \text{otherwise} \end{cases}$$

If the activation function is the sigmoid or tanh function, we have the following.

$$ge_j^{i+1} = \begin{cases} \sigma(lw_j^i) & \text{if } lw_j^i = up_j^i \\ \sigma(lw_j^i) + \lambda \times (x_j^i - lw_j^i) & \text{if } lw_j^i > 0 \\ \sigma(lw_j^i) + \lambda' \times (x_j^i - lw_j^i) & \text{otherwise} \end{cases}$$

$$le_j^{i+1} = \begin{cases} \sigma(up_j^i) & \text{if } lw_j^i = up_j^i \\ \sigma(up_j^i) + \lambda \times (x_j^i - up_j^i) & \text{if } up_j^i \leq 0 \\ \sigma(up_j^i) + \lambda' \times (x_j^i - up_j^i) & \text{otherwise} \end{cases}$$

where $\lambda = \dfrac{\sigma(up_j^i) - \sigma(lw_j^i)}{up_j^i - lw_j^i}$; $\lambda' = min(\sigma'(lw_j^i), \sigma'(up_j^i))$; σ is the respective activation function; and σ' is its first derivative.

For the other 2 components lw_j^{i+1} and up_j^{i+1}, if the layer $i+1$ is an activation function layer, we simply have:

$$lw_j^{i+1} = \sigma(lw_j^i) \qquad up_j^{i+1} = \sigma(up_j^i)$$

where σ is the activation function[2]. Otherwise, if it is an affine layer, DeepPoly applies back substitution to compute the lower bound and upper bound. That is, suppose we want to compute the lower bound lw_j^{i+1}, we begin with its ge_j^{i+1} by setting $ls_j^{i+1}(i+1) = A_j^{i+1} \cdot x^i + B_j^{i+1}$ and compute $ls_j^{i+1}(s)$ for $0 \leq s \leq i$. Note that at any step $s+1$, the value of $ls_j^{i+1}(s+1)$ can be expressed as a general linear expression $b + \sum_{h=0}^{d_s-1} c_h^s \times x_h^s$ where b and each c_h^s are constant values; and x_h^s represents a particular variable at layer s. Thus, we have the following.

$$ls_j^{i+1}(s) = b + \sum_{h=0}^{d_s-1} \left(max(c_h^s, 0) \times ge_h^s + min(c_h^s, 0) \times le_h^s \right)$$

Intuitively, we substitute each variable in the expression by their ge and le components layer-by-layer. The process continues until $s = 0$ and then we can set $lw_j^{i+1} = ls_j^{i+1}(0)$ due to the fact that $ls_j^{i+1}(0)$ is a constant.

Similarly, to compute the upper bound up_j^{i+1} for the variable x_j^{i+1}, we begin with its le_j^{i+1} by setting $us_j^{i+1}(i+1) = A_j^{i+1} \cdot x^i + B_j^{i+1}$. Then we have:

$$us_j^{i+1}(s) = b + \sum_{h=0}^{d_s-1} \left(min(c_h^s, 0) \times ge_h^s + max(c_h^s, 0) \times le_h^s \right)$$

[1]It is actually $clip(I_j - r)$ and $clip(I_j + r)$ where $clip$ is a clipping function so that the abstract states are kept inside the input domain.

[2]For the ReLU activation function layer, our rule guarantees $lw_j^{i+1} \geq 0$, which is slightly different and better than what is defined in [24].

before setting $up_j^{i+1} = us_j^{i+1}(0)$.

To verify the local robustness property, we need to prove that the label of x is always the same as the concrete input I's, say l. Based on the definition of the labelling function, it means at the output layer, we must always have $x_l^{k-1} > x_j^{k-1}$ for all label j such that $j \neq l$. To check whether that is true, at the output layer, the lower bound of x_l^{k-1} is compared with the upper bound of x_j^{k-1} for all $j \neq l$. If $lw_l^{k-1} > up_j^{k-1}$ for all $j \neq l$, the property is verified. Otherwise, for each j which fails the condition, DeepPoly creates a variable $x_j^k = x_l^{k-1} - x_j^{k-1}$ and applies back substitution to find its lower bound. If there is a x_j^k with a lower bound that is not greater than 0, DeepPoly fails to prove the property.

Example 2.3. Given the verification task in Example 2.2, we have the following abstraction for variable x_0^0 and x_1^0.

$$(ge_0^0, le_0^0, lw_0^0, up_0^0) = (ge_1^0, le_1^0, lw_1^0, up_1^0) = (-1, 1, -1, 1)$$

Applying the abstract transformers presented above, we propagate the abstraction from the input layer to other layers. The abstractions are as follows.

$$(ge_0^1, le_0^1, lw_0^1, up_0^1) = (-2x_0^0 - x_1^0 - 1, -2x_0^0 - x_1^0 - 1, -4, 2)$$
$$(ge_1^1, le_1^1, lw_1^1, up_1^1) = (x_0^0, x_0^0, -1, 1)$$
$$\dots$$
$$(ge_0^3, le_0^3, lw_0^3, up_0^3) = (-2x_0^2 + 2x_1^2, -2x_0^2 + 2x_1^2, -4, 2)$$
$$(ge_1^3, le_1^3, lw_1^3, up_1^3) = (2x_0^2 + 2x_1^2 + 1, 2x_0^2 + 2x_1^2 + 1, 1, 5)$$
$$\dots$$
$$(ge_0^5, le_0^5, lw_0^5, up_0^5) = (-2x_0^4 + x_1^4, -2x_0^4 + x_1^4, -1.67, 5)$$
$$(ge_1^5, le_1^5, lw_1^5, up_1^5) = (2x_0^4 - 2x_1^4 - 2, 2x_0^4 - 2x_1^4 - 2, -12, -1.33)$$

To prove the label of x is 0, we need to prove $x_0^5 > x_1^5$. To do that, we first compare the value of lw_0^5 and up_1^5. If $lw_0^5 > up_1^5$, we conclude $x_0^5 > x_1^5$ is satisfied. In this case, however, $lw_0^5 < up_1^5$ and so the verification fails. At this point, DeepPoly employs a last attempt by creating a variable $x_1^6 = x_0^5 - x_1^5$ and applies back substitution to compute its lower bound as follows.

$$ls_1^6(6) = x_0^5 - x_1^5 \quad (1)$$
$$ls_1^6(5) = -4x_0^4 + 3x_1^4 + 2 \quad (2)$$
$$ls_1^6(4) = -1.33x_0^3 + 3x_1^3 - 3.33 \quad (3)$$
$$ls_1^6(3) = 8.67x_0^2 + 3.33x_1^2 - 0.33 \quad (4)$$
$$ls_1^6(2) = ls_1^6(1) = ls_1^6(0) = -0.33 \quad (5)$$

Unfortunately, as shown above, the lower bound of x_1^6 is $lw_1^6 = -0.33 < 0$ and so DeepPoly fails to verify this example.

In the above example, DeepPoly fails to verify the property although it is valid, due to the unreachable states introduced by over-approximation via abstract transformers. One way to address this issue is to refine the abstraction, i.e., to reduce the over-approximation on certain neurons so that the property can be verified.

3 OUR APPROACH

In this section, we first present the details of our abstraction refinement strategy and verification algorithm. Then, we discuss its soundness and complexity.

3.1 Abstraction Refinement

As discussed above, the reason that DeepPoly fails to verify the robustness property of the neural network shown in Fig. 1 is the unreachable states that are introduced by the abstract transformers. One way to address this issue is to refine the abstraction, i.e., to identify one or more neurons x_j^i and refine their abstract states so that a tighter abstraction for subsequent neurons can be computed. Given that the number of neurons in real-world neural networks is often large, ranging from hundreds to millions, we need a systematic way of identifying neurons, which, once refined, potentially allows us to verify the property. We propose to choose the neurons for refinement based on a metric which measures their 'responsibility' for failing the verification. Although the discussion hereafter is based on the local robustness property, we remark that the approach works for the general reachability verification problem defined in Section 2.1.

For a local robustness property, ϕ_O is in the form of $x_l^{k-1} > x_j^{k-1}$ where l is the concrete input's label and j is any other label. Let $(ge_l^{k-1}, le_l^{k-1}, lw_l^{k-1}, up_l^{k-1})$ and $(ge_j^{k-1}, le_j^{k-1}, lw_j^{k-1}, up_j^{k-1})$ be the abstraction computed for x_l^{k-1} and x_j^{k-1} in the DeepPoly domain. Recall that in the case that DeepPoly fails to verify the property, it creates a variable $x_j^k = x_l^{k-1} - x_j^{k-1}$ and finds its lower bound through back substitution.

Our refinement is based on x_j^k. Recall that through back substitution, we can express the lower bound of x_j^k as a linear expression constituted by variables representing any layer (say layer i) of neurons (including the hidden layers and the input layer). That is, the lower bound of x_j^k can be captured as follows.

$$x_j^k \geq b + \sum_{h=0}^{d_i - 1} c_h^i \times x_h^i \quad (6)$$

Intuitively, the coefficient c_h^i of a variable x_h^i in the linear expression is an indication, to some extend, of the contribution of that neuron x_h^i towards not being able to establish $x_l^{k-1} - x_j^{k-1} > 0$. Based on this intuition, we design our refinement strategy as follows.

First, we identify the candidate neurons for refinement, which includes those neurons at the input layer and any hidden layer before the activation function layers (since the affine transformation is encoded exactly in the DeepPoly abstract domain). For instance, for the example shown in Fig. 1, these include the neurons at the input layer (i.e., layer 0) and layers 1 and 3. Furthermore, if the next layer is an activation function layer with the ReLU function, a neuron, represented as x_h^i in DeepPoly, is considered a candidate for refinement if and only if it satisfies $lw_h^i < 0$ and $up_h^i > 0$. The reason is that otherwise, the transformation is exact, i.e., there is no abstraction. For the other cases (i.e., the input layer or a layer whose following layer is the sigmoid or tanh activation function), we only consider a neuron x_h^i if it satisfies $lw_h^i \neq up_h^i$ for the same reason. Then, for each of these candidate neurons, we compute its

impact on failing the verification using the following formula.

$$impact(x_h^i) = max(|c_h^i \times lw_h^i|, |c_h^i \times up_h^i|)$$

For the remaining neurons, we simply set their impact to be 0. Note that we take into consideration the lower bound lw_h^i and upper bound up_h^i of x_h^i in the impact. Effectively, the impact is the magnitude of the term $c_h^i \times x_h^i$ calculated based on the constant lower bound and upper bound calculated by DeepPoly.

Afterwards, the impact of each variable is normalized based on the sum of all neurons' impact in the same layer. Intuitively, this can be explained as follows. Each term in the lower bound of x_j^k, as shown in (6), contributes to failing $x_l^{k-1} - x_j^{k-1} > 0$ and the more terms (6) has, the less each neuron of the layer contributes overall.

$$norm\_impact(x_h^i) = \frac{impact(x_h^i)}{\sum_{n=0}^{d_i-1} impact(x_n^i)}$$

We then choose the variable x_h^i with the greatest normalized impact for abstraction refinement. Refinement is performed by splitting its abstract state. In general, the refinement creates two new abstract states. That is, assume $x_h^i = (ge_h^i, le_h^i, lw_h^i, up_h^i)$, the following new abstract states are created to replace x_h^i.

$$x1_h^i = (ge_h^i, le_h^i, lw_h^i, val)$$
$$x2_h^i = (ge_h^i, le_h^i, val, up_h^i)$$

where val is the splitting point. Its value depends on the next abstract transformer. If the follow-up activation function is the ReLU function, $val = 0$. Otherwise, i.e., if the chosen variable is in the input layer or the follow-up activation function is the sigmoid or tanh function, $val = (lw_h^i + up_h^i)/2$.

Example 3.1. Continue with the previous example, after building the list of linear expressions representing $x_0^5 - x_1^5$ in form of the variables from the previous layers, i.e., (1) to (5), using the above-discussed approach, we identify that the variable x_0^3 has the greatest normalized impact, i.e., $norm\_impact(x_0^3) = 1$ while $norm\_impact(x)$ for any other variable x is 0. Because layer 4 is an activation function layer with the ReLU function, we split the abstract state of x_0^3 at 0 to obtain two new abstract states, i.e., the first one with bounds $[-4, 0]$, and the second one with bound $[0, 2]$.

3.2 Verification Algorithm

We are now ready to present our verification algorithm which incorporates the above-mentioned abstraction refinement as well as optimizations which reduce the workload whenever possible. The details are shown in Algorithm 1.

The algorithm maintains a tree with a maximum height which is the same as the number of layers. Intuitively, the nodes at height i represent the abstraction of layer i of the neural network. For instance, initially the tree has only a root node $\bigwedge_j (ge_j^0 \le x_j^0 \le le_j^0 \wedge lw_j^0 \le x_j^0 \le up_j^0)$, i.e., which is the constraint representing the input constraint. The tree branches only when there is an abstraction refinement. For instance, if we applying DeepPoly once without abstraction refinement, a tree like the one shown in Fig. 2a is constructed, i.e., a tree without any branches. A branch at height i is created only if a neuron at layer i is identified for refinement.

Algorithm 1: *verify*$(N, \phi_I, \phi_O, max\_ref)$

```
1  let T be a tree with an 'open' root
     ψ^0 ← ⋀_j (ge_j^0 ≤ x_j^0 ≤ le_j^0 ∧ lw_j^0 ≤ x_j^0 ≤ up_j^0);
2  let ref_cnt ← 0;
3  while there is a leaf which is 'open' do
4      pick any leaf ψ^p from T which is 'open', say at level p;
5      if ψ^p is at height k − 1 then
6          if φ_O is not verified based on ψ^p then
7              if ref_cnt = max_ref then
8                  return false;
9              identify a neuron, say x_h^i for refinement at val;
10             replace node ψ^i who is the ancestor of ψ^p at level i
                 with 2 new 'open' leaves ψ^i ∧ x_h^i ≤ val and
                 ψ^i ∧ x_h^i ≥ val;
11             ref_cnt ← ref_cnt + 1;
12         else
13             mark the leaf ψ^p as 'closed';
14     else
15         create a node ψ^{p+1} which abstracts layer p + 1 given ψ^p;
16         add ψ^{p+1} as a 'open' child of ψ^p;
17 return true;
```

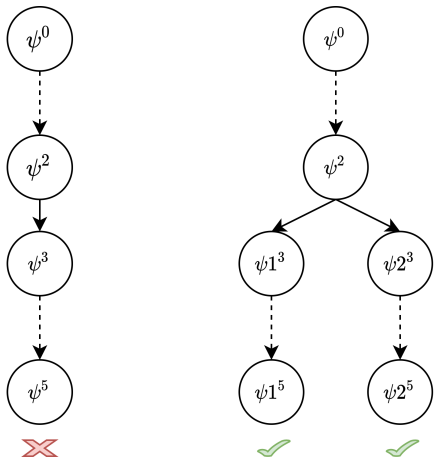
Furthermore, each leaf is marked as either 'open' (i.e., yet to be verified) or 'closed' (i.e., successfully verified). The goal is to close all leaves so that the property is verified.

In the following, we walk through the algorithm step-by-step. The inputs are the verification task $\{\phi_I\}N\{\phi_O\}$ and the maximum number of refinements. At line 1, the root node is created according to ϕ_I as discussed in Section 2.2. During the loop from lines 3 to 16, we grow the tree one node at a time. First a leaf ψ which is 'open', say at level p, is selected at line 4. At line 5, we check whether we have reached the output layer. If so, we check whether ϕ_O can be established as discussed in Section 2.2. If yes, the leaf is marked 'closed' at line 13. Otherwise, we check whether we reach the maximum number of refinements (lines 7-8). If yes, we stop without concluding the property is verified or not. Otherwise, a neuron x_h^i is identified for refinement as discussed in Section 3.1, say at a splitting point val. In such a case, the ancestor of ψ at height i, which represents the current abstraction of layer i, is replaced with two new nodes which represent the abstraction after refinement. In other words, the entire subtree with root ψ^i is to be re-constructed[3].

If the leave ψ is not yet at the output layer, at line 15, a new child node grows on ψ, which is constructed as discussed in Section 2.2. That is, a tuple $(ge_j^{p+1}, le_j^{p+1}, lw_j^{p+1}, up_j^{p+1})$ for each neuron at layer $p + 1$ is created based on ψ and ψ' is set to be $\bigwedge_j (ge_j^{p+1} \le x_j^{p+1} \le le_j^{p+1} \wedge lw_j^{p+1} \le x_j^{p+1} \le up_j^{p+1})$.

The algorithm terminates whenever all leaves are 'closed', in which case we conclude that the property is verified at line 17. Note that it is designed to reuse abstraction constructed in the previous abstraction refinement iterations as much as possible, i.e.,

[3]When the refinement is at layer 0, it creates two separate trees, which makes T become a forest. Each tree is then verified separately.



(a) Before refinement **(b) After refinement**

Figure 2: The verification tree before and after refinement

the abstraction of a neuron is re-computed only if the abstraction of a neuron in an earlier layer has been refined.

Tightening lower and upper bounds after refinement. Note that line 15 works slightly differently from DeepPoly, i.e., because a refinement at an earlier layer i introduces tighter concrete lower bound lw_h^i and concrete upper bound up_h^i for the h-th neuron, we can use them to obtain tighter bounds for subsequent neurons. That is, suppose layer p where $p > i$ is an affine layer and we would like to compute a tighter lw_j^p.

Recall that ls_j^p can be expressed as a linear expression composed of neurons of any previous layer s, written as $ls_j^p(s+1) = b + \sum_{h=0}^{d_s-1} c_h^s \times x_h^s$ where $0 \le s \le p-1$. Thus, we can compute a constant value $lb_j^p(s)$ based on $ls_j^p(s+1)$ as follows.

$$lb_j^p(s) = b + \sum_{h=0}^{d_s-1} \Big(max(c_h^s, 0) \times lw_h^s + min(c_h^s, 0) \times up_h^s \Big)$$

Intuitively, the new bounds obtained through refinement allow us to compute a new bound for lw_j^p which may be tighter than before. That is, we set the lower bound for neuron x_j^p as follows.

$$lw_j^p = max(lb_j^p(0), lb_j^p(1), \cdots, lb_j^p(p-1))$$

In contrast, DeepPoly always sets lw_j^p to be $lb_j^p(0)$ (which is equal to $ls_j^p(0)$) since it is guaranteed to be the maximum in the absence of refinement. Similarly, the upper bound up_j^p for the variable x_j^p can be computed based on the linear expressions $us_j^p(s+1)$ with $0 \le s \le p-1$. For each s, we compute a constant value $ub_j^p(s)$ as follows.

$$ub_j^p(s) = b + \sum_{h=0}^{d_s-1} \Big(min(c_h^s, 0) \times lw_h^s + max(c_h^s, 0) \times up_h^s \Big)$$

Then we can choose the tightest upper bound for x_j^p as follows.

$$up_j^p = min(ub_j^p(0), ub_j^p(1), \cdots, ub_j^p(p-1))$$

Example 3.2. Continue with the previous example, using DeepPoly abstract transformers, we build a tree as shown in Fig. 2a. As ϕ_O cannot be verified based on ψ^5, we refine the node ψ^3 (since x_0^3 has the maximum normalized impact as we show in Example 3.1) and create two new nodes at level 3 of the tree. Afterwards, DeepPoly is applied to grow the nodes one by one until the output layer is reached by both branches, and ϕ_O is verified in both cases.

As a result, the local robustness property of Example 2.2 is verified. Fig. 2b shows the complete verification tree in this example.

Soundness and Complexity. The soundness of Algorithm 1 follows easily from the soundness of DeepPoly and the fact that the two new tasks created at line 10 are equivalent to the old one. Assume that Q is the complexity of verifying one task with DeepPoly; R is the complexity of each refinement; and K is the number of refinements, we have the following observation about the complexity. With a maximum of K refinements, DeepPoly is invoked a maximum of $2K+1$ times. As a result, the complexity is $\mathcal{O}(K(Q+R))$. Note that the complexity of each refinement is linear with the size of the neural network.

4 IMPLEMENTATION AND EVALUATION

We have implemented our approach as an analysis engine in the Socrates framework [19]. Socrates provides a general *Parser* module, which we re-used to parse the neural networks and verification properties. We implemented our approach as a separate module and registered it with the analysis-engine library in the framework. We re-implemented DeepPoly's abstract transformers in Python and used several public libraries such as *numpy*, *autograd*, and *scipy*. In total, less than 500 lines of Python code are required for the implementation, thanks to many reusable facilities provided by the framework.

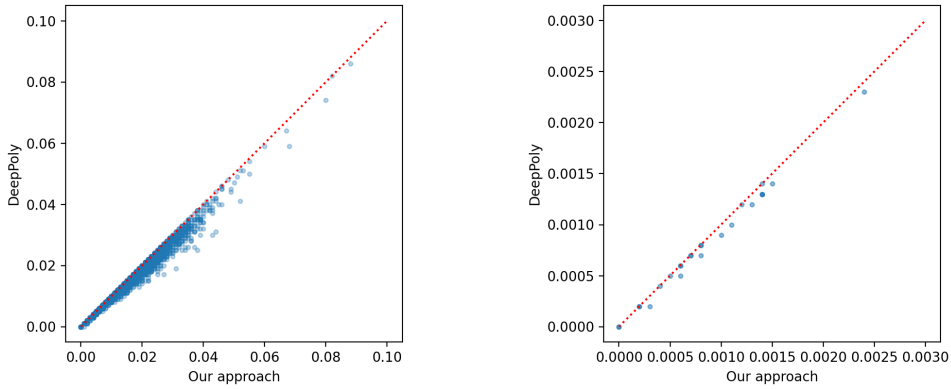
Given the complexity of neural network verification, our implementation features an additional step to quickly falsify a verification task before the Algorithm 1 is applied. That is, we formalize and solve the following optimization problem to check whether there is a concrete counterexample.

$$loss(N, x) = y_l - \max_{j \neq l} y_j \quad \text{where } y = N(x) \wedge x \vDash \phi_I$$

Recall that l is the label of the concrete input I. The loss function returns the difference between y_l and the maximum value of the remaining elements in the output vector. It is easy to see that when the loss function is non-positive, the value of x is a concrete counterexample. We employ the *scipy* library to minimize the loss function and check whether the returned minimal value is non-positive. Since neural networks are designed for optimization, this step often allows us to quickly falsify a verification task.

4.1 Experiment Settings

In the following, we conduct multiple experiments using a set of benchmark verification tasks to answer multiple research questions. All the experiment details, including the source code, are available at [1].



(a) on MNIST networks **(b) on CIFAR10 networks**

Figure 3: Comparing our approach with DeepPoly

Benchmarks. For local robustness property, we collected a benchmark of verification tasks on multiple neural network models trained on the MNIST dataset [17]. We systematically vary the model size to train multiple models, i.e., the number of hidden layers (before preprocessing) vary from 3 to 5, and the number of neurons for each hidden layer varies from 10 to 50 (with a step side of 10). The models are trained using different activation functions, including ReLU, sigmoid, and tanh. In the following, we denote a network with activation function f, k hidden layers, and n neurons in each hidden layer as *mnist_f_k_n* for simplicity. Besides our own trained models, we collect three big feed-forward neural network models which were used in the DeepPoly experiments from the DeepPoly repository, named *mnist_relu_3_1024*, *mnist_relu_5_100*, and *mnist_relu_8_200*, which makes a total of 48 networks. As in the DeepPoly experiments, we choose the first 100 inputs from the test set for robustness verification (due to its complexity, for *mnist_relu_3_1024*, only the first 10 inputs are used). For each network and each input which is classified correctly by the network, we verify its local robustness. In total, there are 4586 verification tasks for MNIST networks. Moreover, to show that our approach can apply to networks with high-dimensional inputs, we collect two more networks trained on the CIFAR10 dataset [16] from the DeepPoly repository, named *cifar_relu_6_100* and *cifar_relu_9_200*. Using the same experiment settings as above, we have 25 verification tasks for these two networks.

To apply our approach for fairness verification, we collect three feed-forward neural network models that are used in fairness testing [32]. Each network has 5 hidden layers (before preprocessing), built with the ReLU activation function. They are trained with three different datasets, named *bank*, *census*, and *credit* [32]. In essence, the networks try to classify people based on their characteristics collected in the dataset, which include protected features such as *age* and *gender*. Each dataset contains different number of protected features, e.g., 1 feature for *bank*, 2 features for *credit*, and 3 features for *census*. For the verification tasks, we use 100 first inputs chosen from each dataset. Again, we do not consider the inputs which are classified incorrectly. In total, we have 285 verification tasks.

Properties. The local robustness is defined based on the infinity norm distance [4]. Instead of fixing the radius r and checking whether we can verify the property (which could be deceiving as one can pick and choose the r value to maximize the effect), we gradually increase r with a step size of 10^{-3} for MNIST networks and 10^{-4} for CIFAR10 networks (because the CIFAR10 networks are more sensible of input permutation), and report the maximum value of r that our approach and alternative approaches manage to verify. The maximum verified value of r is denoted as r_{max}. For each task, the approach which achieves the maximum r_{max} value is the best. In this way, we can precisely assess the capacity of different approaches. For the fairness property, we use the definition as mentioned in Section 2.1.

Experimental Configuration All the experiments are conducted using a machine with 3.1Ghz 8-core CPU and 64GB RAM. We choose DeepPoly, RefinePoly, and Neurify as the baseline comparison and run the experiments with their default settings. For our approach, we set the maximum number of abstraction refinement iterations to be 100. The number is chosen to balance effectiveness and efficiency. It is expected that a larger number of refinements will give better verification results at the cost of more running time.

4.2 Experiment Results

In the following, we answer the research questions one by one based on the experiment results.

RQ1: Does our refinement strategy improve DeepPoly?. To answer the question, we systematically compare the performance of our approach with DeepPoly on all the benchmarks. Recall that without any refinement, our approach is the same as DeepPoly. The results are shown in Fig. 3. Each point in the figure represents one verification task. The x-axis and y-axis show the largest radiuses that can be verified by our approach and DeepPoly respectively. The middle red line helps to show the difference between the results of these tools. We roughly use colors to indicate the density of data points, in which the lighter color represents the region with fewer overlapping points. From the figure, we can see that no data

point lies above the red line, which indicates that the results of our approach are always equal to or better than the results of DeepPoly. For the details, our approach can prove the robustness with larger radiuses in 2365 tasks (i.e., 52%) compared to DeepPoly on MNIST networks. For CIFAR10 networks, our approach gives better results in 11 tasks (i.e., 44%). Given that the number of refinements is limited to 100 and the number of possible refinements is exponential in the number of neurons, the results suggest that we are often able to identify relevant neurons for refinement.

RQ2: How does our approach perform comparing with other refinement strategies?. To answer the question, we compare the performance of our approach with two different refinement strategies. The first one, named InputRange and inspired from [3], chooses to refine the input feature with the largest range (i.e., the one with the maximum $up_j^0 - lw_j^0$ for any j). The second one, named RandomChoice, chooses to refine either one input feature or the input of one neuron which contains an activation function randomly. Note that once the neuron is identified, the refinement is the same as in our approach. As before, the maximum number of abstraction refinement iterations is 100. We remark that the refinement strategy in [29] only works for networks with the ReLU function and is implemented in Neurify, which we will compare as a baseline, and thus is omitted here.

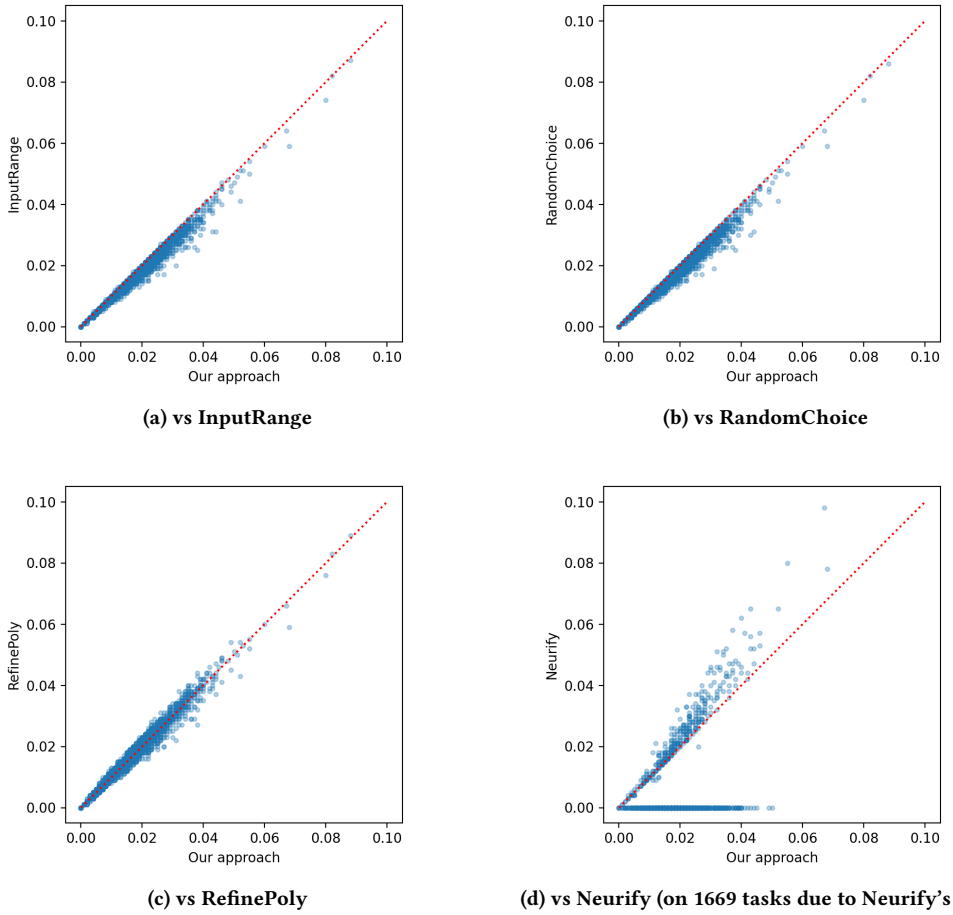
The comparison is performed with all robustness verification tasks on 50 networks. The results for MNIST networks are shown in Fig. 4a and 4b. From the figures, we can see that most of the data points lie on or below the red lines, which means our approach outperforms the other two refinement strategies - sometimes by a considerable amount. In particular, our approach has better results in 2182 tasks (i.e., 48%) and worse results in 7 tasks (i.e., 0.2%) compared to InputRange. Similarly, our approach has better results in 2243 tasks (i.e., 49%) and worse results in 5 tasks (i.e., 0.1%) compared to RandomChoice. We also observe that InputRange and RandomChoice have limited effect compared to DeepPoly, i.e., InputRange has better results in only 297 tasks (i.e., 6%) and RandomChoice has better results in only 200 tasks (i.e., 4%). For CIFAR10 networks, our approach is always better than InputRange and RandomChoice. In particular, our approach gives better results in 11 tasks (i.e., 44%) compared to InputRange and 10 tasks (i.e., 40%) compared to RandomChoice. Again, InputRange and RandomChoice have limited effect compared to DeepPoly on these networks, i.e., only RandomChoice can give better results than DeepPoly and in only 1 task (i.e., 4%). The results thus suggest that our refinement strategy works effectively in many cases.

RQ3: How does our approach perform comparing with other approaches?. The above-mentioned results show that our refinement strategy improves DeepPoly's performance. In the following, we compare our approach with alternative approaches. First, we compare our approach with RefinePoly [21], as it is a newer version from the group who developed DeepPoly. The key idea of RefinePoly is to approximate the output of multiple neurons in the same layer jointly, and so that it can capture the dependencies between these neurons. It should be noted that although RefinePoly may produce better verification results sometimes, it often takes much more time than DeepPoly in general and thus DeepPoly is still considered the state of the art. The results are shown in Fig. 4c on the verification tasks for local robustness on MNIST networks. We can observe that there are some data points above and below the red line, which means that RefinePoly and our approach are complementary to each other. In particular, our approach verifies larger radiuses than RefinePoly in 951 tasks (i.e., 21%), and RefinePoly verifies larger radiuses than our approach in 1420 tasks (i.e., 31%). For the CIFAR10 networks, our approach gives better results in 6 tasks (i.e., 24%) and RefinePoly gives better results in 8 tasks (i.e., 32%). The observation is consistent with the experiment on MNIST networks. The results show that the more expressive abstract domain in RefinePoly helps to get better results. Furthermore, integrating our approach with RefinePoly is likely to yield even better results. To verify the above conjecture, we integrated our refinement strategy into RefinePoly. Our experiments on three networks *mnist_relu_3_10*, *mnist_relu_4_10*, and *mnist_relu_5_10* show that with our refinement strategy, RefinePoly can further improve the results in 228 over 289 tasks (i.e., 79%).

Next, we compare our approach with Neurify [28]. Neurify is a constraint-solving based approach, instead of an abstract-interpretation based approach like DeepPoly and RefinePoly. It is included for comparison because it supports a nontrivial form of refinement and reportedly outperforms other tools such as Reluplex [14] and ReluVal [29]. Neurify chooses the neuron containing the ReLU activation function with the largest gradient based on the network output to refine by splitting the input of the chosen neuron at 0. It should be noted however that Neurify does not support neural networks with the sigmoid and tanh activation functions and its refinement strategy is limited to ReLU activation functions as well. Moreover, Neurify does not support CIFAR10 networks. In this experiment, we are thus restricted to MNIST networks built with ReLU, i.e., 18 networks with 1669 verification tasks. We notice that Neurify employs some solvers to check the network properties, so it is expected that it needs more time to run compared to other approaches. To be fair, we use the running time of our approach for each network with some more spare time (at least 30 minutes) as the time limit for Neurify. Fig. 4d shows the result. In 1318 tasks (i.e., 79%), our approach verifies larger radiuses than Neurify and Neurify verifies larger radiuses than our approach in 256 tasks (i.e., 15%). The results are expected considering that Neurify employs the solvers to verify the network properties instead of reasoning based on the abstract states as in abstract-interpretation based approaches. However, constraint-solving based approaches like Neurify often take significantly more time. In fact, we observe 1311 tasks (i.e., 79%) timeout during the experiment. For each task that does not have verification result due to timeout, we set $r_{max} = 0$. It explains the points at the bottom of the figure.

RQ4: How do different activation functions and sizes of the networks affect our approach?. To answer the question, we break down the results of 45 MNIST networks trained by ourselves in RQ1. The results are shown in Table 1.

As we can see, our approach gives the best results on ReLU networks. The results are encouraging considering that ReLU is the most popular activation function. For networks with sigmoid and tanh activation functions, our approach still improves the results in many cases, although not as many as that of ReLU networks. The



(a) vs InputRange **(b) vs RandomChoice**

(c) vs RefinePoly **(d) vs Neurify (on 1669 tasks due to Neurify's limit)**

Figure 4: Comparing our approach with other refinement strategies and approaches on MNIST networks

Function	#Tasks	#Improvement	Percentage
ReLU	1463	1238	85%
sigmoid	1440	601	42%
tanh	1477	424	29%

Table 1: Comparison between different activation functions

reason is that our refinement splits the abstract states of the ReLU function's inputs at 0, which makes the subsequent abstract transformations precise; while for sigmoid and tanh functions, splitting the abstract states at the middle does not guarantee the subsequent abstract transformations to be precise. Moreover, considering that with the same input range, the output range of the tanh function is larger than of the sigmoid function (i.e., the abstract states from the sigmoid function are more precise), it is expected that our approach achieves better results with sigmoid networks than with tanh networks.

It is also expected that the effectiveness of our approach reduces when the sizes of the networks increase (unless a larger number of refinement is allowed). Indeed, for 206 tasks on MNIST networks from the DeepPoly repository, 102 tasks (i.e., 50%) are improved. Moreover, our approach cannot improve the results for 10 tasks on the biggest network (i.e., *mnist_relu_3_1024*). We notice that RefinePoly, Neurify, as well as the other two refinement strategies cannot improve the results on this network either.

RQ5: Is our approach useful for verifying other properties?. To answer the question, we apply our approach to verify the fairness property. Note that the original DeepPoly implementation does not support verification of fairness and we compare our implementation with and without the refinement strategy to see the effect. To verify the fairness property, given a concrete input I, we set

$$(ge_j^0, le_j^0, lw_j^0, up_j^0) = \begin{cases} (M_j, X_j, M_j, X_j) & \text{if feature } j \text{ is protected} \\ (I_j, I_j, I_j, I_j) & \text{otherwise} \end{cases}$$

where M_j and X_j are the minimum value and the maximum value in the input domain of feature j respectively. After that, we apply the algorithm for verifying local robustness to verify fairness.

The results are shown in Fig. 5. We observe that, with refinement, more tasks can be verified compared to not using refinement. In

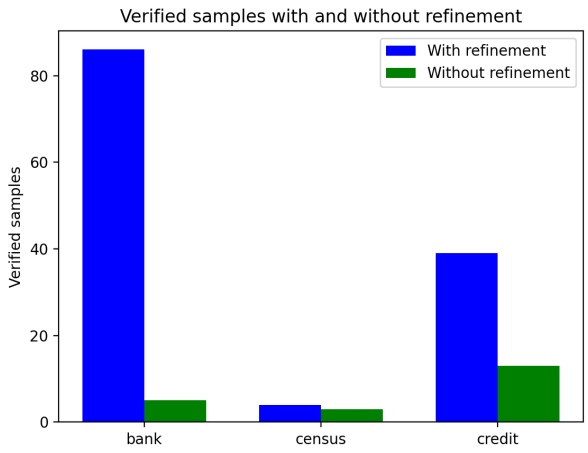


Figure 5: Fairness verification results

particular, the numbers of verified tasks with and without refinement are 86, 4, and 39 compared to 5, 3, and 13. We also notice that the number of verified tasks is more for *bank* and less for *census*. Our interpretation is that this is because the *bank* dataset has only 1 protected feature whereas the *census* dataset has 3. As a result, the latter constitutes a much larger input range.

RQ6: How efficient is our approach?. To answer the question, we report the time required to verify r_{max}. In total, we spent 557 min 33 sec on verifying 4586 tasks on MNIST networks, i.e., 7 seconds per task on average. For CIFAR10 networks, we spent 5 min 41 sec on verifying 25 tasks, i.e., 14 seconds per task on average. We consider that such time efficiency is encouraging. Furthermore, it implies that in practice, we can afford to set the limit on the number of refinements to a bigger number and potentially verify even larger radiuses.

We further analyze the number of refinements that are required to achieve r_{max}. For a total of 4611 tasks, we record that a total of 53811 refinements are performed. Among the tasks generated from refinements, 47384 of them are verified by invoking the DeepPoly algorithm and 11020 are removed later as a refinement at an earlier layer takes place. Considering that only less than 12 refinements on average are required to verify r_{max}, we believe that the choice of neuron for refinement is a reasonably good one in our approach. The falsification step that we perform before verification starts is also shown to be relevant, i.e., 184 tasks are falsified while we try to verify the robustness with $r_{max}+10^{-3}$ or $r_{max}+10^{-4}$. Furthermore, it only takes about 6 seconds in total to falsified these 184 tasks. Finally, we notice that we only need 2 min 50 sec to verify/falsify 285 fairness tasks.

5 RELATED WORK

This work is closely related to research from both the machine learning community and software verification community which aim to establish correctness of neural networks.

Existing efforts on neural network verification can be roughly grouped into two categories: exact methods and approximation methods. The exact methods consider the semantics of neural networks precisely. For example, [25] adopted mixed integer linear programming to tackle the problem, while [7, 14] proposed to solve the problem through SMT solving. In principle, these methods can achieve sound and complete guarantees in neural network verification, i.e., they can prove a property if it holds and they can disprove a property if it does not hold. However, these methods often have limited scalability and consequently applicability. Since neural networks are complex and non-linear functions, exactly capturing their semantics often requires a method to consider an exponential number of cases, which is computationally expensive. Thus these methods are often limited to small neural networks without using the sigmoid or tanh activation functions.

In comparison, approximation methods sacrifice the analysis precision and over-approximate neural network behaviors using a variety of well-studied abstract domains. For instance, the approaches in [30, 31] adopt linear approximation, whereas the approaches in [8, 22, 24] leverage or devise numerical abstract domains from existing abstract interpretation methods. Intuitively, approximation simplifies the presentation of the neural network behaviors through introducing unreachable states, i.e., abstracts the behaviors of non-linear functions using a convex linear shape, and thus can leverage existing techniques to solve the problem efficiently. AI^2 is the first approximation approach, which employs the Zonotope abstraction to verify neural networks, and then DeepZ improves its precision via better abstract transformers for the activation functions. Later, DeepPoly devises a new abstract domain based on polyhedra and intervals suitable for neural network verification. Recently, researchers propose the star-based approach [26, 27] to approximate neural network behaviors. Thanks to the approximation, these approaches enjoy more scalability than exact methods and can handle sigmoid and tanh activation functions. The problem is they are not complete, i.e., they may fail to verify a valid property.

To mitigate the precision loss due to approximation, one solution is to devise a more precise approximation such as DeepPoly proposed in [24]. The other is to refine the approximation if it is necessary, i.e., by splitting a verification problem into several verification tasks in small scales and verify them separately. All the works [7, 14, 18, 28, 29] and the approach proposed in this paper belong to this category. Our work differs from the other works by the approach we use to identify the neuron for refinement, which is shown to work effectively with different activation functions.

6 CONCLUSION AND FUTURE WORK

In this work, we propose an approach to verify neural networks through abstraction refinement. According to the evaluations, our approach could improve state-of-the-art approaches based on the same abstract domain, and outperform other refinement strategies.

In terms of future work, we would like to explore two directions. On the one hand, we plan to extend our approach to support more complicated neural networks, such as recurrent neural networks [5, 11] and capsule neural networks [10, 20], so as to further advance the field of neural network verification. Moreover, we would like to improve our approach to handle even larger neural networks and thus it can work on neural networks in more realistic scenarios.

REFERENCES

[1] Anonymous: Source and Benchmark (2021), https://figshare.com/s/bc5d0a430a99ed67b3d6

[2] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., et al.: End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316 (2016)

[3] Bunel, R.R., Turkaslan, I., Torr, P., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Advances in Neural Information Processing Systems. pp. 4790–4799 (2018)

[4] Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: 2017 ieee symposium on security and privacy (sp). pp. 39–57. IEEE (2017)

[5] Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259 (2014)

[6] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)

[7] Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: International Symposium on Automated Technology for Verification and Analysis. pp. 269–286. Springer (2017)

[8] Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 3–18. IEEE (2018)

[9] Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)

[10] Hinton, G.E., Sabour, S., Frosst, N.: Matrix capsules with em routing. In: International conference on learning representations (2018)

[11] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)

[12] Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: International Conference on Computer Aided Verification. pp. 3–29. Springer (2017)

[13] Joseph, M., Kearns, M.J., Morgenstern, J.H., Roth, A.: Fairness in learning: Classic and contextual bandits. In: Lee, D.D., Sugiyama, M., von Luxburg, U., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. pp. 325–333 (2016), http://papers.nips.cc/paper/6355-fairness-in-learning-classic-and-contextual-bandits

[14] Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)

[15] Kourou, K., Exarchos, T.P., Exarchos, K.P., Karamouzis, M.V., Fotiadis, D.I.: Machine learning applications in cancer prognosis and prediction. Computational and structural biotechnology journal **13**, 8–17 (2015)

[16] Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)

[17] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)

[18] Lu, J., Kumar, M.P.: Neural network branching for neural network verification. arXiv preprint arXiv:1912.01329 (2019)

[19] Pham, L.H., Li, J., Sun, J.: Socrates: Towards a unified platform for neural network verification. arXiv preprint arXiv:2007.11206 (2020)

[20] Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. arXiv preprint arXiv:1710.09829 (2017)

[21] Singh, G., Ganvir, R., Püschel, M., Vechev, M.: Beyond the single neuron convex barrier for neural network certification. In: Advances in Neural Information Processing Systems. pp. 15098–15109 (2019)

[22] Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Advances in Neural Information Processing Systems. pp. 10802–10813 (2018)

[23] Singh, G., Gehr, T., Püschel, M., Vechev, M.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (2018)

[24] Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 41 (2019)

[25] Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. arXiv preprint arXiv:1711.07356 (2017)

[26] Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. arXiv preprint arXiv:2004.05511 (2020)

[27] Tran, H.D., Lopez, D.M., Musau, P., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Star-based reachability analysis of deep neural networks. In: International Symposium on Formal Methods. pp. 670–686. Springer (2019)

[28] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems. pp. 6367–6377 (2018)

[29] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1599–1614 (2018)

[30] Weng, T.W., Zhang, H., Chen, H., Song, Z., Hsieh, C.J., Boning, D., Dhillon, I.S., Daniel, L.: Towards fast computation of certified robustness for relu networks. arXiv preprint arXiv:1804.09699 (2018)

[31] Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. arXiv preprint arXiv:1711.00851 (2017)

[32] Zhang, P., Wang, J., Sun, J., Dong, G., Wang, X., Wang, X., Dong, J.S., Ting, D.: White-box fairness testing through adversarial sampling. In: Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (2020)