Abstraction-based Reduction of Input Size for Neural Networks

Peter Backeman[0000−0001−7965−248X], Edin Jelačić[0009−0006−2745−4282], Cristina Seceleanu[0000−0003−2870−2680], Ning Xiong, and Tiberiu Seceleanu

Mälardalen University, Västerås, Sweden
{peter.backeman, edin.jelacic, cristina.seceleanu, ning.xiong, tiberiu.seceleanu}@mdu.se

Abstract. Machine learning is an increasingly popular method for modeling complex systems, to make predictions or recognize data patterns. A common machine learning model is the neural network, which can be trained to represent complicated functions to a high accuracy. While neural networks often grow large and complex, recent work is looking in how to abstract networks to yield simpler representations, while retaining some property of the original network. For instance, for every input, the abstracted network's output should be at least as large as the original. In this work, we build on previous ideas and extend them to allow for removing inputs, obtaining an under/over-approximating network instead. Further, we show how to combine these approximating networks to identify inputs which have a low impact on the final output.

Keywords: Neural network · Abstraction · Dimensionality reduction · Feature Selection.

1 Introduction

The advent of machine learning algorithms has led to their application for classification, decision making, and data analytics of complex systems, in various fields. By presenting a machine learning (ML) algorithm with a set of training patterns (relating inputs to outputs), one can obtain a model that can predict the output for an arbitrary, unseen input (in the input domain). *Neural networks* (NN), as a subset of ML approaches, can be trained to represent complicated functions to a high accuracy, however they often grow large and complex. Recent work has been studying ways of abstracting neural networks to yield simpler representations, without breaching desirable properties of the original network, e.g., for every input the abstracted network's output is at least as large as the original.

In some cases the input vector has a large size, yet only a few of the elements are significant in the computation of the output. To reduce the size of the input vector, one can for example apply principal component analysis (PCA) to obtain a smaller representation while retaining the most important information [1]. In this paper, we show how a combination of an over-approximating and an

under-approximating network can be used to identify insignificant input elements (i.e., not affecting the output by more than a relatively small *delta*). While the PCA method generates a reduced dimensional data representation, our approach focuses on identifying a subset of elements to represent the data. This approach imposes tighter constraints but facilitates a more direct connection between the new input domain and the original one, as it remains a subset, enabling a form of feature selection [2]. Furthermore, it allows us to obtain information regarding individual variables. Understanding the insignificance of certain inputs can be valuable in itself, by its potential to reveal properties about the original problem.

In this work, we build on previous ideas [3], and extend them to also consider the input layer of NN. We study how input nodes can be eliminated from a network in such a manner that an over/under-approximating network is obtained. Such an abstracted network would not be as accurate, but can be used for analysis, as well as for gaining an understanding of the network model. It should be noted that this method works on the final trained model, which means it is applicable in scenarios where the original training data may not be available.

The contributions of this paper are as follows:

- A novel method for abstracting away selected inputs from a neural network, obtaining an over/under-approximation of the latter,
- A method for obtaining a worst-case measure of the impact of a particular input to a neural network,
- A small experimental evaluation demonstrating how the proposed methods can be applied.

We begin by presenting background in Sec.2, that is, a brief presentation of neural networks, including verification and abstraction techniques. In Sec. 3, we show our method of how inputs can be removed to obtain an over/under-approximating network with smaller dimension, followed in Sec. 4 by our proposed method of identifying insignificant inputs. We perform a small experimental evaluation in Sec. 5. Finally, we present our conclusions and future work in Sec. 7.

2 Background

We start by introducing some notation regarding vectors. We use $\mathbf{x} = \{x_1, \ldots, x_n\}$ to denote *vectors* from domain $\bar{X}$, where $\mathbf{x}[i] = x_i$. We denote substitution of the i-th element by c as $\mathbf{x}[x_i = c] = \{x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n\}$. In this paper all vectors are over real numbers. Given a vector $\mathbf{x} = \{x_1, \ldots, x_i, \ldots, x_n\}$, let $\mathbf{x}^{+i} = \{x_1, \ldots, x_{i-1}, x_i, x_i, x_{i+1} \ \ldots, x_n\}$, that is the vector $\mathbf{x}$ with the i-th element repeated once, and $\mathbf{x}^{-i} = \{x_1, \ldots, x_{i-1}, x_{i+1} \ldots, x_n\}$, the vector $\mathbf{x}$ with x_i removed. We let X_i^{max} and X_i^{min} represent the maximum and minimum values of the domain of x_i, respectively.

2.1 Neural Networks

Neural networks (NN) are a widely used form of machine learning [4]. They consist of interconnected neurons trained on input-output pairs to learn functions

and make predictions of their output values. Each neuron - denoted by capital letters - in a layer, is connected to each neuron in the previous layer via an edge with a *weight*. When computing the output value of a node, an activation function is applied to the sum consisting of a node bias and each weight of every incoming edge multiplied by the output value of its corresponding node. In this paper, we focus on networks that contain the input layer (nodes $I_1, I_2, \dots$), a number (L) of hidden layers (nodes $H_1^1, H_2^1, \dots, H_1^2, H_2^2 \dots, \dots, H_1^L, H_2^L, \dots$) and an output layer with a single output node (O). We will use $\forall H_i^\ell \in H^\ell$ to express statements about all nodes in layer H^ℓ.

The input and output layer has no activation (i.e., the identity function), while the hidden layers use the ReLU activation function, i.e.,:

$$\begin{aligned} H_i^1 &= \text{ReLU}(\mathbf{W^1}[i]\mathbf{I} + \mathbf{B^1}[i]) \\ H_i^\ell &= \text{ReLU}(\mathbf{W}^\ell[i]\mathbf{H^{\ell-1}} + \mathbf{B}^\ell[i]) \qquad \text{for} \quad 1 < \ell \leq L \\ O &= \mathbf{W^O}[i]\mathbf{H^L} + \mathbf{B^O} \end{aligned}$$

where $\mathbf{I}$ is the output of the input layer, $\mathbf{W}^\ell[i], \mathbf{B}^\ell[i]$ are the weights and the bias of node H_i^ℓ, $\mathbf{W^O}[i]$ and $\mathbf{B^O}$ are weights and the bias for the output node, and ReLU is the nonlinear activation function. Let $e(I_i, H_j^1)$ be the weight of the edge connecting input node I_i with hidden layer node H_j^1, and $e(H_i^\ell, H_j^{\ell+1})$ the weight of the edge connecting hidden layer node H_i^ℓ and $H_j^{\ell+1}$ (and similarly for $e(H_i^\ell, O_1)$. We use $NN(\mathbf{x})$ to denote the output of the neural network NN with input $\mathbf{x}$.

Example 1. In Fig. 1 a simple neural network NN is presented. It has two input nodes I_1, I_2, two layers of hidden nodes $H^1 = \{H_1^1, H_2^1\}$ and $H^2 = \{H_1^2, H_2^2, H_3^2\}$, and one output node O. There are also weighted edges, e.g. $e(I_1, H_2^1) = 1.38$, $e(H_2^1, H_1^2) = 0.81$ and $e(H_2^2, O_1) = 0.59$. Consider feeding the input vector $\mathbf{x} = (10, 10)$ into the network, then[1]:

$$\begin{aligned} H_1^1 &= \text{ReLU}(1.88x_1 - 0.07x_2 + 0.52) = 18.62 \\ H_2^1 &= \text{ReLU}(1.38x_1 + 0.15x_2 + -0.31) = 14.99 \\ H_1^2 &= \text{ReLU}(2.06H_1^1 + 0.81H_2^1 - 0.23) = 50.27 \\ &\dots \\ O &= 1.56H_1^2 + 0.59H_2^2 - 0.23H_3^2 - 0.07 \approx 100,7 \end{aligned}$$

Thus $NN(\mathbf{x}) \approx 100.7$

2.2 Verification of NNs

Recently, significant efforts have been put into the verification of NNs [5], and tools for checking various aspects of NNs have arisen in the past several years [6].

[1] For convenience, we will overload and use the notation N_i to refer both to the node N_i, and its output value.

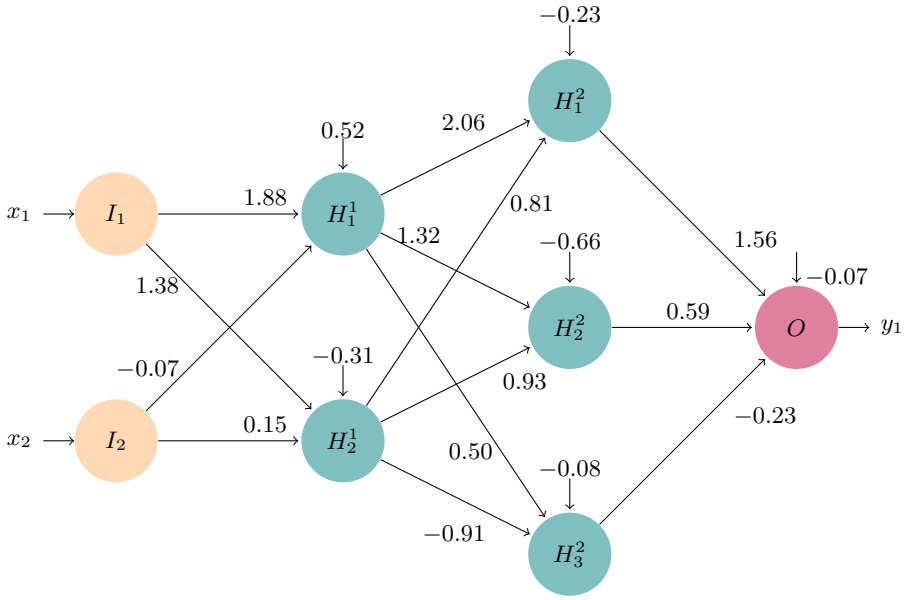

Fig. 1. Simple neural network.

For example, the verification tool `Marabou` [7] can prove a linear bound on the output under given constraint on the input in a neural network with piece-wise linear activation functions, of which ReLU is one. We use it by asking for an assignment to a network yielding an output violating the bound, and if none is found the bound is proven.

Example 2. Consider the neural network presented in Fig. 1. Let's say that when we restrict the real-valued inputs x_1 and x_2 to the range $[0, 10]$, it appears that the output is strictly less than 101. To rigorously establish this, we can employ `Marabou` by setting input constraints as follows: $x_1, x_2 \in [0, 10]$, and defining the output bound as $NN(\mathbf{x}) > 100$ (note that, in practice, we formulate a bound on the negated output: $-NN(\mathbf{x}) \leq -100$, since `Marabou` requires the constraint to be in the form of a less-than-or-equal comparison). `Marabou` will in this case return that the given constraints are unsatisfiable, verifying that 101 is indeed an upper bound for the output of the network (when $x_1, x_2 \in [0, 10]$).

2.3 Abstracting NNs

In this section, we present a short summary of the work by Elboher et al. [3], as our work is significantly based upon it. In the mentioned work, the authors present a methodology for abstracting and refining neural networks. The motivation is to create, for a given neural network NN, a simpler network (i.e., with fewer nodes) NN' such that $NN(\mathbf{x}) \leq NN'(\mathbf{x})$. This network can then be

used for verification of upper bounds; since the abstract network is an over-approximation, any upper bound for NN' is also an upper bound for NN.

Coloring. The core idea of the methodology of Elboher et al. is to classify all nodes into categories, depending on how each node is contributing to the final output. The nodes are grouped as *positive/negative*, as well as *increasing/decreasing*. The former group contains nodes that have all outgoing edges with positive/negative values, respectively, while the latter has nodes such that increasing the output of the node would increase/decrease the output of the network. In this paper, we only consider the second category, i.e., we only care if nodes are increasing or decreasing, and use the following change of terminology: we *color* a node N *green* if increasing the input value to N results in the final network output increasing. Analogously, if increasing the input value to a node N would lead to the total output of the network decreasing, we color the node *red*. It can be the case that a node cannot be colored either *green* or *red*, then we call the node *colorless*.

We can iteratively color all the nodes in a network. The process begins with the output node (we assume networks have exactly one output node in the final layer). The output node is trivially *green*. To color each node H_i^L in the second to last layer (the final hidden layer H^L), we can use the following formula:

$$\text{color}(H_i^L) = \begin{cases} \mathit{green}, & \text{if } e(H_i^L, O) \geq 0 \\ \mathit{red}, & \text{if } e(H_i^L, O) < 0 \end{cases}$$

Intuitively, if a node is connected to the output node with a positive weight, increasing its output will increase the output of the network, thus it is *green* (and v.v.). Note that if for a node H_i^ℓ, $e(H_i^\ell, O) = 0$, the node H_i^ℓ has no non-zero outgoing weight and can be ignored as it can not affect the output of the network. In the remainder of the paper, we frequently write $\mathit{green}(N)$ as a short-hand for $\mathit{color}(N) = \mathit{green}$ (and analogously for *red*).

The situation is a bit more complicated for the preceding hidden layers. Assume that layer $H^{\ell+1}$ has been colored (i.e., all nodes $H_i^{\ell+1}$ are either *green* or *red*). Now, each node that is a node in layer H^ℓ can be colored according to the following formula:

$$\text{color}(H_i^\ell) = \begin{cases} \mathit{green}, & \text{if } \forall H_k^{\ell+1} \in H^{\ell+1} \quad (e(H_i^\ell, H_k^{l+1}) \geq 0 \wedge \mathit{green}(H_k^{\ell+1})) \vee \\ & (e(H_i^\ell, H_k^{\ell+1}) \leq 0 \wedge \mathit{red}(H_k^{\ell+1})) \\ \mathit{red}, & \text{if } \forall H_k^{\ell+1} \in H^{\ell+1} \quad (e(H_i^\ell, H_k^{\ell+1}) \geq 0 \wedge \mathit{red}(H_k^{\ell+1})) \vee \\ & (e(H_i^\ell, H_k^{\ell+1}) \leq 0 \wedge \mathit{green}(H_k^{\ell+1})) \end{cases}$$

Intuitively, this means that if a node is connected to only *green* nodes with positive weights or *red* nodes with negative weights, it will increase the total output of the network, and is thus *green*. On the other hand, if a node is connected to only *green* nodes with negative weights or *red* nodes with positive weights, it

will decrease the total output of the network, and is thus *red*. In a similar way as for the last hidden layer, if a node has no non-zero outgoing weights it can be ignored.

Example 3. Consider the network in Fig. 1, if we increase the output of node H_1^2, the input of O_1 will increase, thus increasing the output of the network. Therefore, node H_1^2 is *green* (and likewise for H_2^2). On the other hand, increasing the output of node H_3^2 actually decreases the input of O (since it is multiplied by the weight -0.23) and therefore H_3^2 is *red*. In the preceding layer, increasing the output of H_2^1, will increase the input to both H_1^2 and H_2^2 (and since they are green, increasing the total output) and decrease the input to H_3^2 (which is red, thus also increasing total output). Thus, H_2^1 can be colored *green*. However, increasing the output of H_1^1 both increases the input to H_1^2 and H_3^2 which has a mixed effect on the total output of the network. Thus H_1^1 remains *colorless*. We show the (partially) colored network in Fig. 2.

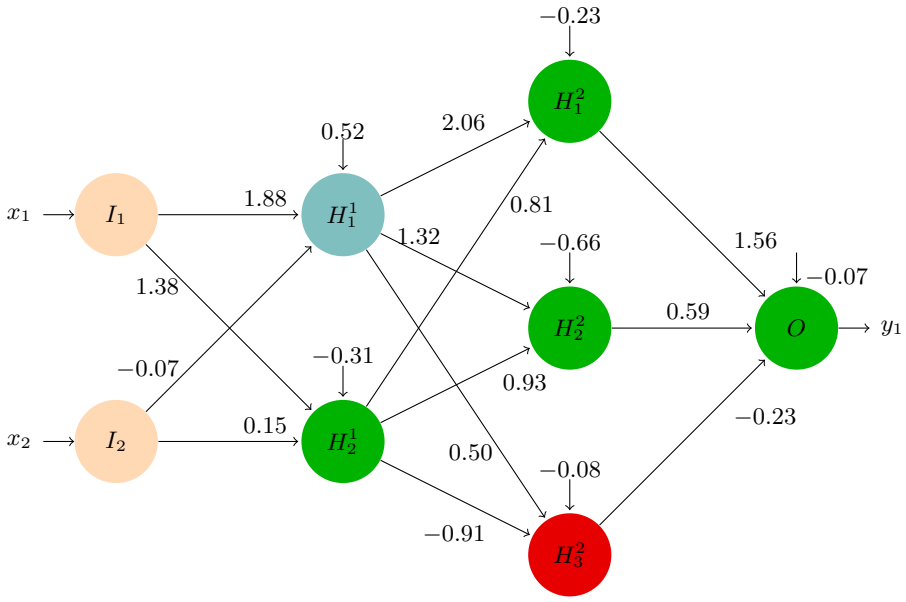

Fig. 2. Colored neural network.

Splitting node. Coloring nodes in a network might be impossible right away, as none of the two cases might apply. Then we proceed with a *split*. Splitting a node means replacing it by two nodes with the same incoming weights but each outgoing edge of the original node is only mapped to exactly one of the new nodes. This ensures that the two new nodes have the same output (as they

have identical inputs), as well as the sum of the output of the two new nodes to each node in the next layer is identical to the output of the split node in the original network. Thus the resulting network computes the exact same function as before splitting, but has one extra node.

We can see that since we assign each edge of the original node to one of the new nodes, we can choose freely which edge to assign to which node. We let all positive edges to green nodes and negative edges to red nodes be assigned to the first node, and the remaining edges to the second node. Following this, it is easy to see that one of the new nodes will be colored *green*, and the other *red*. Formally, if we split node H_i^ℓ into two new nodes H_{i+}^ℓ and H_{i-}^ℓ, we assign edges such that in the new network the following holds for all nodes $H_j^{\ell+1}$ in the next layer (we let $h_i = H_i^\ell$ and $h_j = H_j^{\ell+1}$ for clarity):

$$e(H_{i+}^\ell, h_j) =$$

$$\begin{array}{ll} e(h_i, h_j) & \textbf{if}(e(h_i, h_j) \geq 0 \wedge \mathit{green}(h_j)) \vee (e(h_i, h_j) \leq 0 \wedge \mathit{red}(h_j)) \\ 0 & \textbf{if}(e(h_i, h_j) \geq 0 \wedge \mathit{red}(h_j)) \vee (e(h_i, h_j) \leq 0 \wedge \mathit{green}(h_j)) \end{array}$$

$$e(H_{i-}^\ell, h_j) =$$

$$\begin{array}{ll} 0 & \textbf{if}(e(h_i, h_j) \geq 0 \wedge \mathit{green}(h_j)) \vee (e(h_i, h_j) \leq 0 \wedge \mathit{red}(h_j)) \\ e(h_i, h_j) & \textbf{if}(e(h_i, h_j) \geq 0 \wedge \mathit{red}(h_j)) \vee (e(h_i, h_j) \leq 0 \wedge \mathit{green}(h_j)) \end{array}$$

Moreover, the incoming edges to H_{i+}^ℓ and H_{i-}^ℓ are identical to the edges to the node H_i^ℓ before splitting:

$$\forall H_j^{\ell-1} \in H^{\ell-1} \quad e(H_j^{\ell-1}, H_{i+}^\ell) = e(H_j^{\ell-1}, H_i^\ell) = e(H_j^{\ell-1}, H_{i-}^\ell)$$

We omit the special case when the preceding layer is the input layer.

Example 4. Once again, we consider the neural network we saw earlier. It could not be completely colored as H_1^1 remained *colorless*. Therefore, we split H_1^1 into two nodes H_{1+}^1 and H_{1-}^1 accordingly. The new network can be colored and is shown in Fig. 3. It should be noted that for any input vector, the networks in Fig. 2 and Fig. 3 compute exactly the same output value.

3 Removing Inputs by Over/Under-estimation

In their work, Elboher et al. define an abstraction operator, to reduce the total number of nodes in the hidden layers of the network. However, in our work, we utilize the coloring in a different manner. In this section, we present how we can remove an input node x_i from a neural network NN creating an over-estimating network NN_i^+, such that:[2]

$$\forall \mathbf{x} : NN(\mathbf{x}) \leq NN_i^+(\mathbf{x}^{-i}). \tag{1}$$

[2] $\mathbf{x}^{-i}$ refers to the vector $\mathbf{x}$ with the i:th input removed.

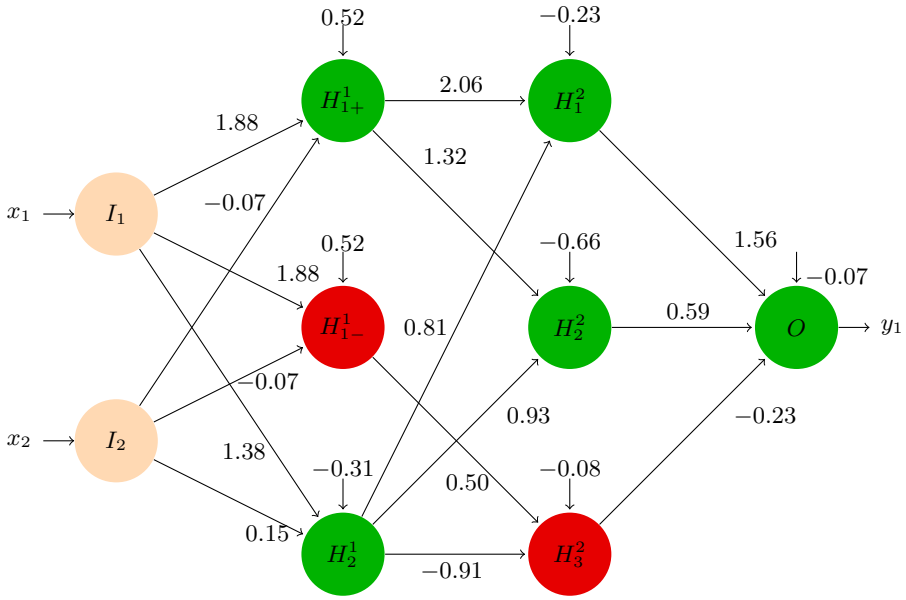

Fig. 3. Colored split neural network. Edges with weight zero are omitted.

Assume that the NN is already colored (with potentially split nodes) according to the methodology described in Sec. 2.3. To remove input x_i, we begin by splitting the node I_i (into two new nodes I_i^+ and I_i^-, which are green and red respectively). For the resulting network NN'_i, we have $NN(\mathbf{x}) = NN'_i(\mathbf{x}^{+i})$, that is, if we duplicate the i:th element of the input vector and feed it into the new network, we have the same result as the original vector for the original network.[3].

Example 5. We revisit the example neural network. If we split the second input node I_2, the resulting network is as shown in Fig. 4.

Now, for any input vector to NN'_i, we know that increasing the input value of I_i^+ will increase the output of the network (as the node is green). In particular, if we change the input to be the maximum value, with X_i^{max} the network output can only grow:

$$NN'_i(\mathbf{x}^{i+}[x_i = X_I^{max}]) \geq NN'_i(\mathbf{x}^{i+})$$

Note that the i:th input corresponds to the green node I_i^+ (and $i+1$:th to the red node I_i^-). In similar vein, if we replace the value for input node I_i^- with X_i^{min} the output also grows:

$$NN'_i(\mathbf{x}^{i+}[x_{i+1} = X_I^{min}]) \geq NN'_i(\mathbf{x}^{i+})$$

[3] $\mathbf{x}^{+i}$ refers to the vector $\mathbf{x}$ with the i:th input duplicated.

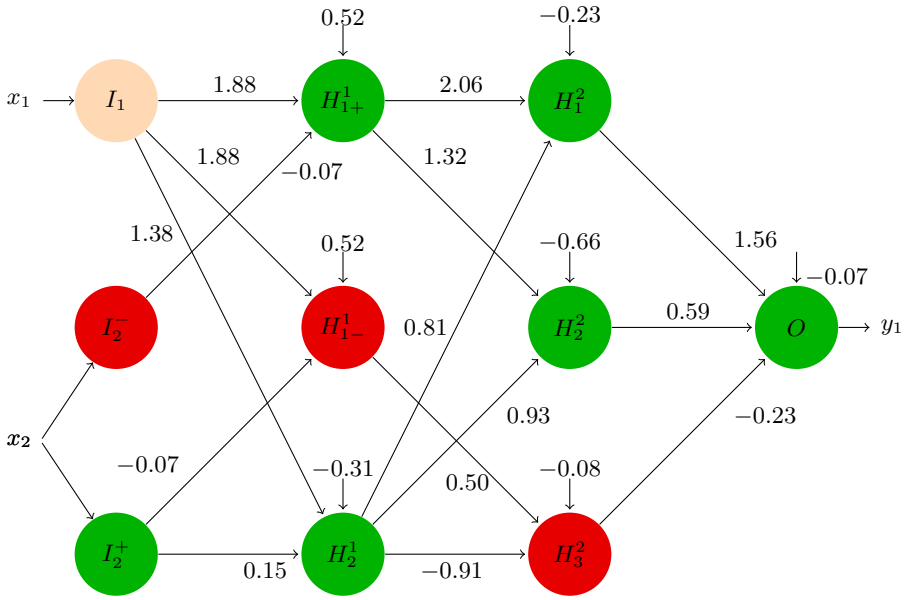

Fig. 4. Neural network after the second input node has been split. Edges with weight zero are omitted.

With this in mind, we create a new network NN_i^+ by replacing the input nodes I_i^+ and I_i^- with constant inputs X_i^{max} and X_i^{min}, respectively. To achieve this, the bias for each node in the first hidden layer is increased with it's incoming weight from I_i^+ (I_i^-) multiplied with X_i^{max} (X_i^{min}). The procedure for doing this is outlined in Alg. 1. The new network can be fed input vectors, with the i:th input removed, and will compute an over-approximation of the result.

Algorithm 1: Algorithm for creating NN_i^+.

Input: Neural network NN and input node I_i
Output: Output Neural network NN_i^+
Color network NN;
Split I_i into I_i^+ and I_i^- s.t. I_i^+ can be colored green and I_i^- red;
for *each* $e(I_i^+, H_j^1) \neq 0$ **do**
  $\mathbf{B_{H^1}}[j] \leftarrow \mathbf{B_{H^1}}[j] + e(I_i^+, H_j^1) * X_i^{max}$;
for *each* $e(I_i^-, H_j^1) \neq 0$ **do**
  $\mathbf{B_{H^1}}[j] \leftarrow \mathbf{B_{H^1}}[j] + e(I_i^-, H_j^1) * X_i^{min}$;

Example 6. Fig. 5 shows the resulting network of replacing I_2^+ and I_2^- with $X_2^{max} = 10$ and $X_2^{min} = 0$, and then propagating to the bias accordingly to

the procedure shown in Alg. 1. Note that for any input vector, if we remove the second element and feed it into the network of Fig. 5, the result will always be equal or greater to feeding the full vector into any of the networks before I_2 has been removed.

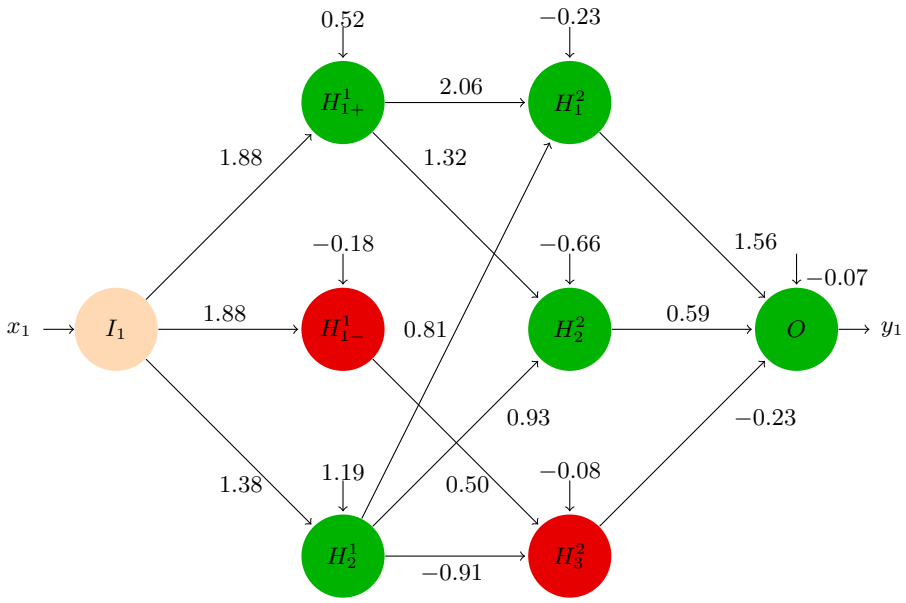

Fig. 5. Neural network after the second input has been removed. Edges with weight zero are omitted.

Theorem 1.

$$\forall \mathbf{x} : NN(\mathbf{x}) \leq NN_i^+(\mathbf{x}^{-i})$$

This theorem tells us that the neural network NN_i^+ is an over-approximating network, disregarding the ith input. The proof follows from the construction of the network (i.e., splitting and coloring).

We can in a symmetric way define a network NN_i^- underestimating the resulting value. The construction is similar to Alg. 1, except that X_i^{max} and X_i^{min} are swapped, i.e., for each green (red) node we replace with the minimum (maximum) value to minimize the output. Given a NN_i^- constructed as such, a symmetric theorem will hold.

Theorem 2.

$$\forall \mathbf{x} : NN(\mathbf{x}) \geq NN_i^-(\mathbf{x}^{-i})$$

4 Identifying Insignificant Inputs

One of the challenges in using neural networks, especially when we lack knowledge about the network's internal structure, is dealing with the potentially high dimensionality of input data. Input vectors can comprise hundreds, or even more, values. In some cases, only a subset of these inputs may significantly influence the output. In this section, we demonstrate an approach that can help to identify inputs with a low impact on the output of the network. We use the following to define insignificant inputs:

Definition 1. *For a NN with input range $\bar{X}$ and a particular input x_i, we call x_i* insignificant *(w.r.t. some $T \in \mathbb{Z}$) if*

$$\forall \mathbf{x} = \{x_0, \ldots, x_n\} \in \bar{X} : \left(\max_{c \in X_i} NN(\mathbf{x}[x_i = c]) - \min_{c \in X_i} NN(\mathbf{x}[x_i = c]) \right) \leq T$$

Intuitively, this means that for any input vector, varying x_i will not change the resulting output by more than a threshold T. For a given neural network, it can be interesting to identify the insignificant inputs, as for low enough T, these could be disregarded as their impact is negligible.

Example 7. Consider a neural network designed to estimate the total load on a computer system, where each input x_i is either zero or one, indicating whether a function f_i of the system is activated. The output should be a prediction of the CPU load of the system as a percentage. If a function f_i can be identified as insignificant w.r.t. to $L = 1$, it can be deduced to not affect the final load by more than 1%.

Computing the maximum and minimum values as required in Def. 1 by enumeration can take a prohibitively long time due to large domains. Instead, we propose to use the `Marabou` verification tool to establish if a value is insignificant or not. We begin by presenting how we can construct a *difference neural network*.

Definition 2. *We define a difference neural network for an input node x_i as NN_i^{diff}, such that,*

$$NN_i^{diff}(\mathbf{x}) = NN_i^{+}(\mathbf{x}) - NN_i^{-}(\mathbf{x})$$

Intuitively, a difference network NN_i^{diff} bounds for an input vector $\mathbf{x}^{-i}$ how much the i:th input affects the total output. It works by taking the difference between the over-approximation and under-approximation for the input vector, thus measuring a maximum change between the the highest and lowest possible values. We present a high-level algorithm for constructing a difference network in Alg. 2 for a particular input x_i. The resulting network can then be analyzed using `Marabou` to establish if it is the case that the output is less than the limit L. If this is the case, this fulfills the condition of Def. 1, hence x_i can be deemed insignificant.

Algorithm 2: Algorithm for creating a difference network.

Input: Neural network NN, Input node I_i
Output: Output result Difference neural network NN_i^{diff}
Create NN_i^+, NN_i^- as described in Alg. 1.;
Merge the two input layers;
Create an extra layer, computing the difference between the two output nodes;

Example 8. Consider the neural networks in Fig. 6. The three steps of the methodology are shown. First a neural network is obtained after training on a data set generated by the function $f(x_1, x_2) = 100x_1 + x_2$. Next we remove the first input I_1 to create an over-approximating network (the under-approximating network is not shown). Finally, we apply Alg. 2 to obtain the bottom network shown in Fig. 6. Note that the upper and lower half of the network are identical except for the biases on the hidden nodes, as well as the weight from the subtraction layer to the output layer (one for top half, negative one for bottom half). If we apply `Marabou` and verify, the results indicates that the first input is significant, as expected.

4.1 Estimating Impact of Inputs

In this section, we explore the idea of not only finding insignificant inputs, but estimating the *maximum impact of a particular input*. A difference network tells us for an input node x_i an upper bound on the change in the output for a particular assignment to the remaining inputs. Thus, if we can establish an upper bound on the output of a difference network, *we can establish an upper bound on the impact for the particular input* x_i. `Marabou` can verify if the output of a network is limited by some constant b. By applying this verification repeatedly, we can establish a bound on the difference network. We use a *binary-split* approach shown in Alg. 3 (note that the query $\forall \mathbf{x} NN_i^{diff}(\mathbf{x}) > bound$ is answered by one query to `Marabou`).

The binary approach considers the inputs one-by-one: it takes a lower bound (assumed to be zero) and an upper bound and checks if the middle point is a bound. If this is the case, the true bound lies somewhere in the lower half of the interval, otherwise in the upper half. In this way the search interval is repeatedly split by two until it is sufficiently small. Note, if the upper bound is too small, an erroneous bound will be reported (as the search will not investigate bounds above the original upper bound). This can be alleviated by initially checking if the upper bound is a true upper bound.

5 Experimental Evaluation

In this section, we perform two small studies to evaluate the approach presented in this paper. The implementation of the algorithms presented above can be found at `https://github.com/ptrbman/neuralnetworkabstraction`.

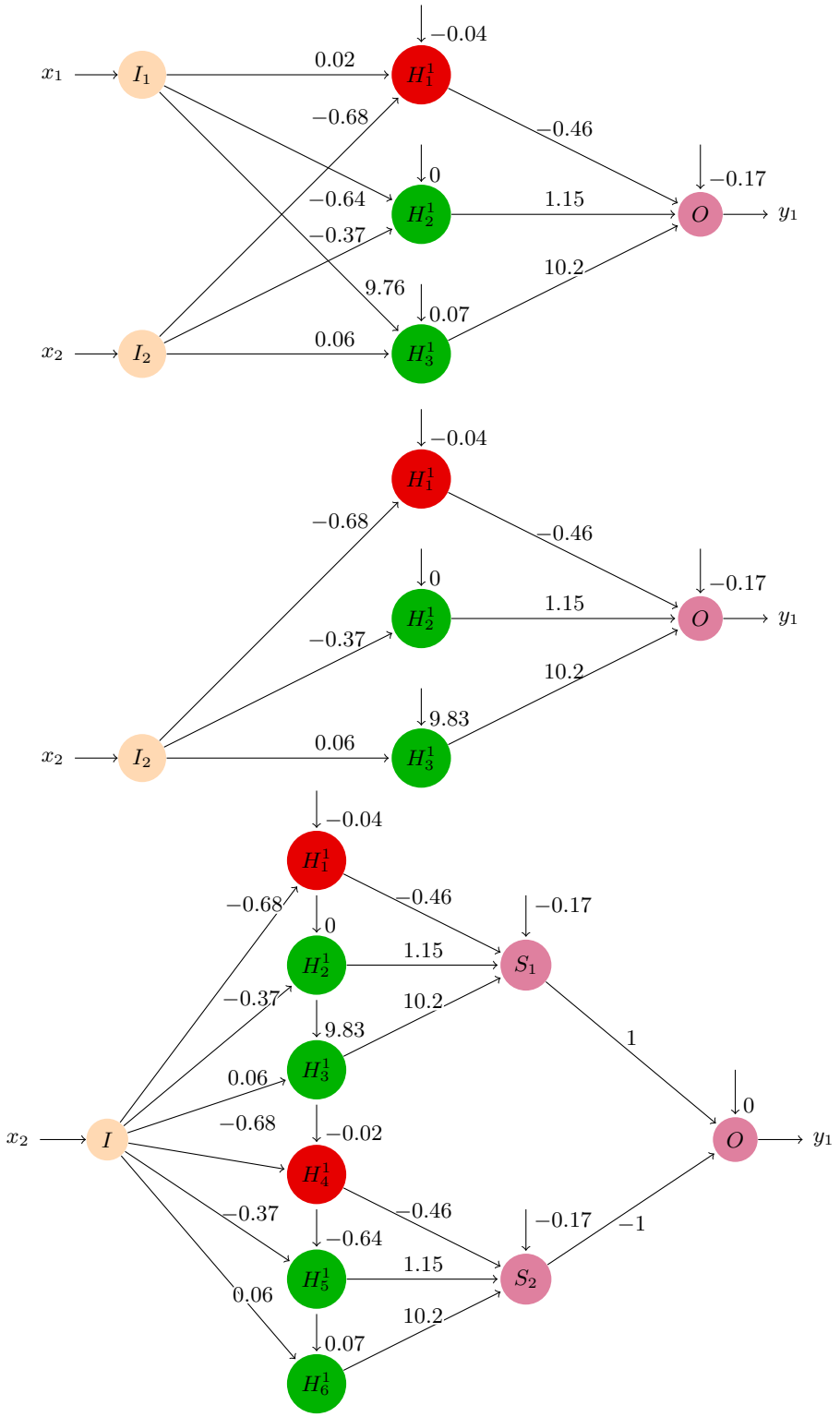

Fig. 6. (Top) A neural network trained to compute $f(x_1, x_2) = 100x_1 + x_2$. (Middle) The network after I_1 is removed. (Bottom) The difference network NN_1^{diff}. We omit all edges with weight zero.

Algorithm 3: Algorithm for the binary approach.

Input: Neural network NN and input node I_i
Input: Upper bound UB for search
Output: An upper bound of the impact of input node I_i
Create difference network NN_i^{diff};
$lower \leftarrow 0$;
$upper \leftarrow UB$;
$bound \leftarrow UB/2$;
while $upper > lower + 1$ **do**
  if $\forall \mathbf{x} NN_i^{diff}(\mathbf{x}) > bound$ **then**
    $lower \leftarrow bound$;
  else
    $upper \leftarrow bound$;
return $bound$;

5.1 Polynomial Coefficient Estimation

Consider a polynomial of the form $f(\bar{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$. If we restrict $x_i \in \{0, 1\}$ it is clear that the maximum impact of any variable x_i is equal to c_i, as when changing x_i from zero to one (or v.v.) will affect the final sum by at most c_i. We generate ten random polynomials of this form with $n = 10$ and $0 < c_i \leq 10$ and the resulting estimated impact of each input is shown in Table 1. It is noteworthy that the estimated coefficient is very often quite close to the original, except in the third case where many of the estimations are far off. This is probably due to a poorly trained network with a high validation error. The results demonstrates that the approach is capable of extracting information from a trained neural network. Each binary search requires around 60 queries from `Marabou`, where each query takes around 1 millisecond. The total time is about 16 seconds (with overhead coming from reloading the model from disk for each query).

5.2 Identifying Demanding Functions

We exemplify our approach on an industrial example: a certain embedded device has multiple interconnected software components that are operated by switching them or their sub-components "on" or "off". Every set of these values (concretely, 1 for "on" and 0 for "off"), is deemed a configuration of the device. Every configuration loaded onto the device and activated exerts a certain load on the device's CPU. We create a neural network that predicts the expected CPU load of the embedded device under a certain configuration. Afterwards, we can apply the methodology presented in this paper to estimate the impact of each input, using the maximum (1) and minimum (0) values (the respective function being enabled and disabled, respectively). The final results are shown in Table 2. These values provide the knowledge that, under the assumption that the neural network is accurate, the functions $0, 1$ and 3, never affect the CPU load by more

c_i	6	3	3	4	10	6	2	8	5	7
est.	6.640	4.296	3.515	4.296	11.328	6.640	1.953	8.984	5.078	7.421
c_i	5	4	7	8	10	6	10	1	2	4
est.	5.078	4.296	7.421	8.203	10.546	6.640	10.546	1.171	2.734	4.296
c_i	1	1	6	4	3	1	4	9	6	9
est.	4.296	6.640	8.203	6.640	5.078	3.515	5.859	8.984	5.859	8.984
c_i	4	5	4	5	7	3	6	4	2	5
est.	4.296	5.078	4.296	5.078	7.421	2.734	5.859	4.296	1.953	5.078
c_i	9	4	4	7	9	5	4	5	10	9
est.	8.984	5.859	6.640	8.984	8.984	8.203	8.984	5.078	9.765	8.984
c_i	7	5	3	3	1	6	7	4	1	1
est.	8.203	5.078	2.734	4.296	3.515	5.859	8.203	4.296	2.734	3.515
c_i	3	3	4	4	2	5	10	9	6	6
est.	3.515	3.515	4.296	4.296	1.953	5.078	10.546	9.765	6.640	6.640
c_i	10	4	10	10	3	6	10	3	4	3
est.	9.765	5.078	9.765	9.765	5.078	8.203	9.765	3.515	8.203	3.515
c_i	1	2	5	8	5	3	10	8	4	7
est.	1.171	3.515	5.859	8.984	5.859	3.515	11.328	8.984	4.296	8.203
c_i	6	7	10	8	9	3	6	10	1	6
est.	5.859	6.640	10.546	8.203	8.984	8.203	5.859	9.765	6.640	5.859

Table 1. Estimated coefficients of linear polynomial. The top value corresponds to the actual coefficient and the bottom value is the estimated.

than roughly one percent, while function 11 has by far the greatest impact on the CPU load (but turning it off does not save more than roughly ten percent). The `Marabou` solver was queried 52 times, with each query taking between 1 and 500 milliseconds. The total time for estimating the bounds was about 26 seconds (once again with overhead coming from reloading the model from disk for each query).

Function	0	1	2	3	4	5	6	7	8	9	10	11	12
Est. impact	1.17	1.17	1.95	1.17	1.95	1.95	1.95	2.73	3.51	1.95	2.73	9.76	2.73

Table 2. The maximum impact on CPU load different functions in industrial example.

5.3 Repeated Experiments

Repeated experiments on neural network training are complicated as the outcome depends on the initial (randomized) weights. Thus, the result may vary between experiments. Noteworthy is the fact that verification of a network depends on its contained weights, that is, two networks with the same structure (e.g., nodes, layers) but different weights might have significant variation in the time that it takes to verify a certain property. It has been encountered during

our experiments that certain networks take a long time to verify. Most often, this seems to be related to a high validation loss for the trained network. As our methodology does not really value any output from such a network (if the validation loss is high, the predicted accuracy of insignificant inputs is very low), we would abort such a task and restart the training with the goal of ensuring a more accurate model before continuing with the rest of the process.

6 Related Work

This work is mainly based on previous research found in the literature [3], and there have been several extensions or related work closely resembling the topic of the mentioned paper. One extension looks into how to retain information between different verification queries on similar networks [8]. This could also be possible for our case, perhaps the removal of two different insignificant input nodes requires much of the same work. Another example identifies nodes in a network that always produce an output almost zero (thus enabling the removal of those nodes) [9]. In another related work [10], the authors reformulate a Deep Neural Network (DNN) minimization problem as a DNN verification problem, and construct a provably minimal network (that is still sufficiently close to the original).

Moreover, there are different approaches towards abstracting neural networks, e.g., [11]. It would be interesting to study if these methods could also be extended to the input layer, as they give different kinds of guarantees on the abstraction, potentially enabling different use cases.

Identifying insignificant inputs can be seen as a form of *feature selection*, as studied in the literature [2]. However, since we aim for an approximated result, our method is allowed a bit more leeway when discarding input dimensions. At the moment, we have not investigated how this premise could affect other popular feature selection methods.

7 Conclusions and Future Work

In this paper, we propose a new methodology for removing inputs of neural networks, by creating an over/under-approximating network. We then show how such networks can be combined into a difference network that is capable of estimating the impact of various inputs, and we therefore combine it into a methodology by which we can identify insignificant inputs. All steps work on a trained neural network where one does not have access to the training data.

7.1 Future Work

The solution presented in this paper has been run on small networks with an unoptimized prototype implementation. We intend to improve the implementation and detect insignificant inputs in more complicated examples to investigate the

scalability of the approach. Furthermore, we have seen great variations in the speed of the underlying solver `Marabou`, and it would be beneficial to gain more understanding when the verification becomes hard. Finally, it is also interesting to see if extra goals during training (e.g., minimizing weights from input nodes) can affect the usability of the approach by tweaking the final model to try and isolate significant inputs.

Acknowledgements.

We acknowledge the support of the Swedish Knowledge Foundation via PerFlex - Performant and Flexible digital Systems through Verifiable Artificial Intelligence project, grant nr. 20220033, and ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038.

References

1. I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016.
2. V. Kumar and S. Minz, "Feature selection: a literature review," *SmartCR*, vol. 4, no. 3, pp. 211–229, 2014.
3. Y. Y. Elboher, J. Gottschlich, and G. Katz, "An abstraction-based framework for neural network verification," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pp. 43–65, Springer, 2020.
4. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. Book in preparation for MIT Press.
5. F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," 5 2018.
6. C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, M. J. Kochenderfer, *et al.*, "Algorithms for verifying deep neural networks," *Foundations and Trends® in Optimization*, vol. 4, no. 3-4, pp. 244–404, 2021.
7. G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, pp. 443–452, Springer, 2019.
8. Y. Y. Elboher, E. Cohen, and G. Katz, "Neural network verification using residual reasoning," in *Software Engineering and Formal Methods* (B.-H. Schlingloff and M. Chai, eds.), (Cham), pp. 173–189, Springer International Publishing, 2022.
9. S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz, "Simplifying neural networks using formal verification," in *NASA Formal Methods* (R. Lee, S. Jha, A. Mavridou, and D. Giannakopoulou, eds.), (Cham), pp. 85–93, Springer International Publishing, 2020.
10. B. Goldberger, G. Katz, Y. Adi, and J. Keshet, "Minimal modifications of deep neural networks using verification.," in *LPAR*, vol. 2020, p. 23rd, 2020.
11. F. Boudardara, A. Boussif, P.-J. Meyer, and M. Ghazel, "A review of abstraction methods towards verifying neural networks," *ACM Trans. Embed. Comput. Syst.*, aug 2023. Just Accepted.