Abstraction-based Reduction of Input Size for Neural Networks

Peter Backeman$^{[0000-0001-7965-248X]}$, Edin Jelačić$^{[0009-0006-2745-4282]}$, Cristina Seceleanu, Ning Xiong, and Tiberiu Seceleanu

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

Abstract. Machine learning is an increasingly popular method for modeling complex system. A common machine learning model is the neural network. They can be trained to represent complicated functions to a high accuracy. However they often grow large and complex. Recent work is looking in how to abstract networks to yield simpler representation, while retaining some property of the original network, e.g., for every input the abstracted networks output is at least as large as the original. In this work, we build on previous ideas and extend it to also consider the input layer.
Sometimes, the input vector has a large size, while only a few of the elements are significant in the computation of the output. In this paper, we propose to use a trained neural network model to identify insignificant input elements, i.e, elements which do not contain important information. We show how the presented abstraction method for the input layer can be utilized to achieve this.

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

1 Introduction

Machine learning is an increasingly popular method for modeling complex system. By presenting a machine learning algorithm with a set of training patterns (relating inputs to outputs), one can obtain a model which can predict the output for an arbitrary input (in the input domain). They can be trained to represent complicated functions to a high accuracy. However they often grow large and complex. Recent work is looking in how to abstract networks to yield simpler representation, while retaining some property of the original network, e.g., for every input the abstracted networks output is at least as large as the original.

In this work, we build on previous ideas and extend it to also consider the input layer [1]. We consider how input nodes can be eliminated from a network in such a manner that an over/under-approximating network is obtained. Such a network would not be as accurate, but if one is tasked with ensuring that a certain bound is not breached, it can still be useful. For example, if we are using a neural network to predict processor load w.r.t. a set of functions being executed on the processor, with this approach we could eliminate inputs, effectively adding their worst case load (or an over-approximation of it) statically to the output. If the abstracted network would then state that the bound is respected w.r.t. a set of activated functions, we know that the actual load would also be below the threshold. Additionally, the resulting network would have a smaller input dimensionality, which could make it more efficient, especially if we wish to verify some property of it (which can be resource-consuming for larger networks).

In some cases the input vector has a large size, while 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 [2]. In this paper, we show how a combination of an over-approximating and an under-approximation network can be combined to identify insignificant input elements (i.e., not affecting the output by more than a 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 greater 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 [3]. Moreover, it allows us to obtain information regarding individual variables. Understanding the insignificance of certain inputs can be valuable by itself, revealing properties about the original problem.

We begin by presenting background in Sec.2, including brief presentation of neural networks, including verification and abstraction techniques. In Sec. 3 we show how inputs can be removed to obtain an over/under-estimating network with smaller dimension, followed in Sec. 4 by a method of identifying insignificant inputs. Finally we present our conclusions and future work in Sec. 6.

2 Background

We begin by introducing some notation regarding vectors. We use $\mathbf{x} = \{x_1, \ldots, x_n\}$ to denote *vectors*, 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. Let $\mathbf{x} \setminus x_i = \{x_1, \ldots, x_{i-1}, x_{i+1} \ldots, x_n\}$, i.e., 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 (NNs) are a widely used form of machine learning [4]. They consist of interconnected neurons trained on input-output pairs to learn and make predictions. Each neuron in a layer is connected to each neuron in the previous layer via an edge with a *weights*. 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, for simplicity we focus on networks that contain the input layer (nodes $I_1, \ldots$[1]), one hidden layer (nodes $H_1, \ldots$) and a single output node (node O_1). The input and output layer has no activation (i.e., the identity function), while the hidden layer uses the ReLU activation function, i.e.,:

$$H_i = \mathrm{ReLU}(\mathbf{W_H}[i]\mathbf{I} + \mathbf{B_H}[i]),$$

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

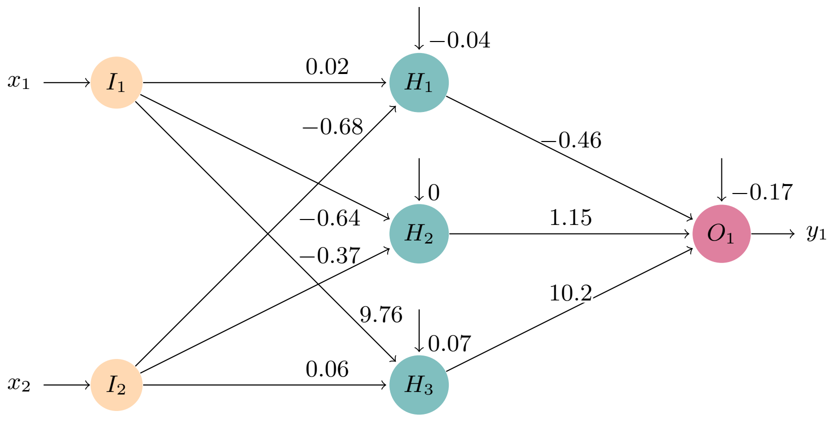

Fig. 1. Simple neural network.

Example 1. In Fig. 1 a simple neural network NN is presented. It has two input nodes I_1, I_2, three hidden nodes H_1, H_2, H_3 and one output node O_1, as well as (for example) weights $e(I_2, H_2) = -0.37$ and $e(H_2, O_1) = 1.15$. Consider feeding the input vector $\mathbf{x} = (0, 1)$ into the network, then:

$$\begin{aligned}
H_1 &= \mathrm{ReLU}(0.02x_1 + (-0.68)x_2 - 0.04) = 0\\
H_2 &= \mathrm{ReLU}(-0.64x_1 + (-0.37)x_2 + 0) = 0\\
H_3 &= \mathrm{ReLU}(9.76x_1 + 0.06x_2 + 0.07) = 0.13\\
O_1 &= -0.46H_1 + 1.15H_2 + 10.2H_3 - 0.17 = 1.326
\end{aligned}$$

Thus $NN(\mathbf{x}) = 1.326$

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

2.2 Verification of NNs

Recently, significant efforts have been put into the verification of NNs. [5] There exist many aspects of networks that can be checked, and many tools for checking NNs have arisen in the past several years. [6] 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.

Example 2. Consider the neural network presented in Fig. 1. Without loss of generality, and for simplicity, we omit the bias terms in the neural network. When we restrict the real-valued inputs x_1 and x_2 to the range between zero and one, it appears that the output can never be less than 0.5. To rigorously establish this, we can employ Marabou by setting input constraints as follows: $P(\bar{\mathbf{x}}) = 0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1$, and defining the desired output property as $Q(NN(\bar{\mathbf{x}})) > 0.5$ (note that, in practice, we use the property $-Q(NN(\bar{\mathbf{x}})) \leq -0.5$ since Marabou requires the constraint to be in the form of a less-than-or-equal comparison, though the mathematical meaning remains equivalent). It's important to highlight that we need to negate the property we are seeking—essentially, we are querying Marabou for the existence of an input $\bar{\mathbf{x}}$ such that the output is greater than -0.5.

2.3 Abstracting NNs

In this section we present a short summary of the work in [1] by Elboher et. al., as our work is significantly based upon it. In [1], a methodology for abstracting and refining neural networks is presented. The motivation is to create, for a given neural network NN, a simpler network (i.e., with fewer nodes) NN' such that $NN(\mathbf{x}) \leq NN'(\mathbf{x})$.

The core idea of their methodology is to classify all nodes into categories, how they are contributing to the final output: we call a node n *green* if increasing the input value to n results in the network output increasing, otherwise we call it *red*. In this paper we present only a simplified approach with one hidden layer. With multiple hidden layers sometimes nodes need to be split to allow for single color per node (in [1] they classify on two orthogonal categories, while here we simplify to one color).[2] Formally, for the hidden layer we color each node as follows:

$$\text{color}(H_i) = \begin{cases} \textit{green}, & \text{if } e(H_i, O_1) \geq 0 \\ \textit{red}, & \text{if } e(H_i, O_1) < 0 \end{cases}$$

Example 3. Consider the network in Fig. 1, if we increase the output of node H_3, the input of O_1 will increase, thus increasing the output of the network. Therefore, node H_3 is green. On the other hand, increasing the output of node H_1 actually decreases the value of O_1 (since it is multiplied by the weight -0.46. Thus, H_1 is considered red. We show the colored network in Fig. 2.

[2] We intend for a full paper to expand this and integrate the full algorithm with multiple layers.

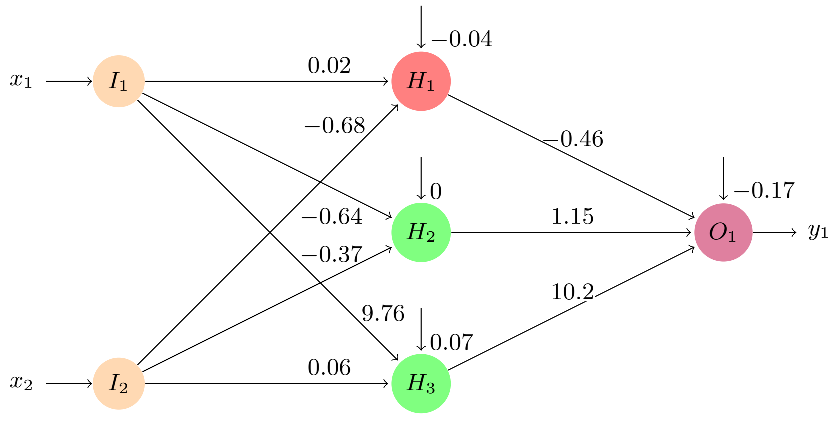

Fig. 2. Colored neural network.

3 Removing Inputs by Over/Under-estimation

In their work, Elboher et. al. also define an abstraction operator, which we base our idea in this section.[3] In this section we present how we can remove an input node x_i from a neural network NN creating an overestimating network NN_i^+, such that

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

Assume that NN is already extended with nodes and classified according to the methodology described in Sec. 2.3. To remove input x_i. Let H^+ be the set of all the nodes in the hidden layer which satisfy color(H) = *green*, and conversely H^- the set of all the nodes in the hidden layer which satisfy color(H) = *red*. Intuitively, we wish to split the I_i into two pseudo-nodes I_i^+ and I_i^-, such that the former would be green (i.e., contributing to the increase of the network output) and the latter red (i.e., contributing to the decrease of the network output). Formally, we define a new network NN_i' with input node I_i replaced by I_i^+ and I_i^- such that:

$$\forall H_j \in \{H_1, \ldots H_n\} \qquad e(I_i^+, H_j) =$$

[3] We do not implement the abstraction and refinement loop of [1], but it could also be helpful for our work.

$$\begin{array}{ll} e(I_i, H_j), & \begin{array}{c}\textbf{if}(e(I_i, H_j) \geq 0 \wedge \mathit{color}(H_j) = \mathit{green}) \\ \vee \\ (e(I_i, H_j) \leq 0 \wedge \mathit{color}(H_j) = \mathit{red})\end{array} \\ \\ 0, & \begin{array}{c}\textbf{if}(e(I_i, H_j) \geq 0 \wedge \mathit{color}(H_j) = \mathit{red}) \\ \vee \\ (e(I_i, H_j) \leq 0 \wedge \mathit{color}(H_j) = \mathit{green})\end{array} \end{array}$$

and

$$\forall H_j \in \{H_1, \dots H_n\} \qquad e(I_i^-, H_j) =$$

$$\begin{array}{ll} e(I_i, H_j), & \begin{array}{c}\textbf{if}(e(I_i, H_j) \geq 0 \wedge \mathit{color}(H_j) = \mathit{green}) \\ \vee \\ 0 \leq 0 \wedge \mathit{color}(H_j) = \mathit{red})\end{array} \\ \\ (e(I_i, H_j), & \begin{array}{c}\textbf{if}(e(I_i, H_j) \geq 0 \wedge \mathit{color}(H_j) = \mathit{red}) \\ \vee \\ (e(I_i, H_j) \leq 0 \wedge \mathit{color}(H_j) = \mathit{green})\end{array} \end{array}$$

For the resulting network NN'_i, we have $NN(\mathbf{x}) = NN'_i(\mathbf{x}^i)$. Now, for any input vector $\mathbf{x}^i$, if we modify the input value to I_i^+ with X_i^{max} the network output can only grow, i.e.:

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

In similar vein, if we replace the value for input node I_i^- with X_i^{min} the output also grows:

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

Finally, we modify NN'_i such that we obtain a network which always treats the input of I_i^+ as X_I^{max}, and of I_i^- as X_I^{min}. Since in such a network, the inputs for I_i^+, I_i^- are ignored, we can remove the input nodes by propagating the constant X_I^{max} through I_i^+ by multiplying each outgoing edges weight with X_I^{max} and add to the bias of the corresponding node in the hidden layer (and analogously for I_i^-). Note that the network obtained with these two input nodes fulfills Eq. (1) and we denote this network NN_i^+. The procedure for doing this is outlined in Alg. 1.

Theorem 1.

$$\forall \mathbf{x} : NN(\mathbf{x}) \leq NN_i^+(\mathbf{x} \setminus x_i)$$

Example 4. Consider the network in Fig. 1. We can apply the over-estimation strategy to the first input node to obtain the network shown in Fig. 3. Note that since we are doing an over-approximation and the edges from x_1 to H_1 and H_2 are a positive weight to a red node, and a negative weight to a green node, they are not added to the bias. However, the final weight from x_1 to H_3 is multiplied with one and added to the bias of H_3 as it is the maximum effect the x_1 input can have on the output of H_3.

Algorithm 1: Algorithm for creating NN_i^+.

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

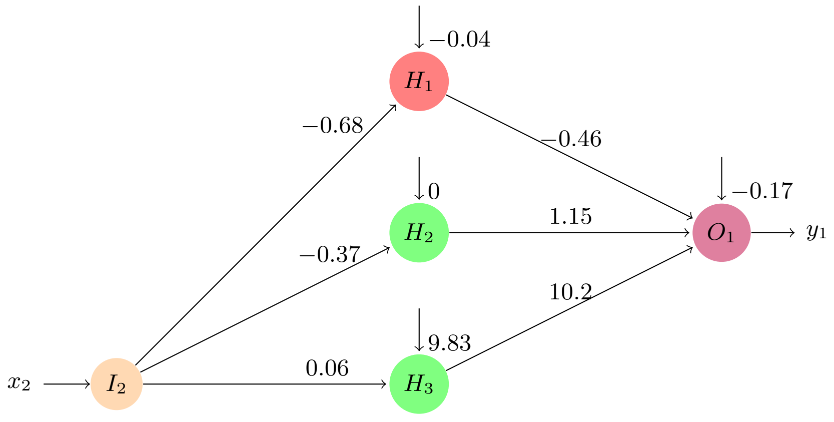

Fig. 3. Over-approximating neural network with input x_1 removed.

We can in a symmetric way define a network NN_i^- underestimating the resulting value.

Theorem 2.

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

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 many cases, only a subset of these inputs may significantly influence the output, while the values assigned to the remaining inputs have a negligible impact on the output, often approximated as a small constant.

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

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

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

Example 5. 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 $L = 0.1$, it might be possible to ignore it in an analysis as its impact on the final load is very small.

As the domains can be infinite, computing the maximum and minimum values as required in definition 1 is not possible by enumeration. 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}, s.t.,*

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

We present a high-level algorithm for constructing a difference network in Alg. 2. 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.

Example 6. Consider the neural network in Fig. 1. It is obtained after training on a data set generated by the function $f(x_1, x_2) = 100x_1 + x_2$. If we apply Alg. 2 to the network and the first input node, we obtain the network shown in Fig. 4. 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

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;
Create a final output layer with one node;
```

apply Marabou and verify, the results indicates that the first input is significant, as expected.

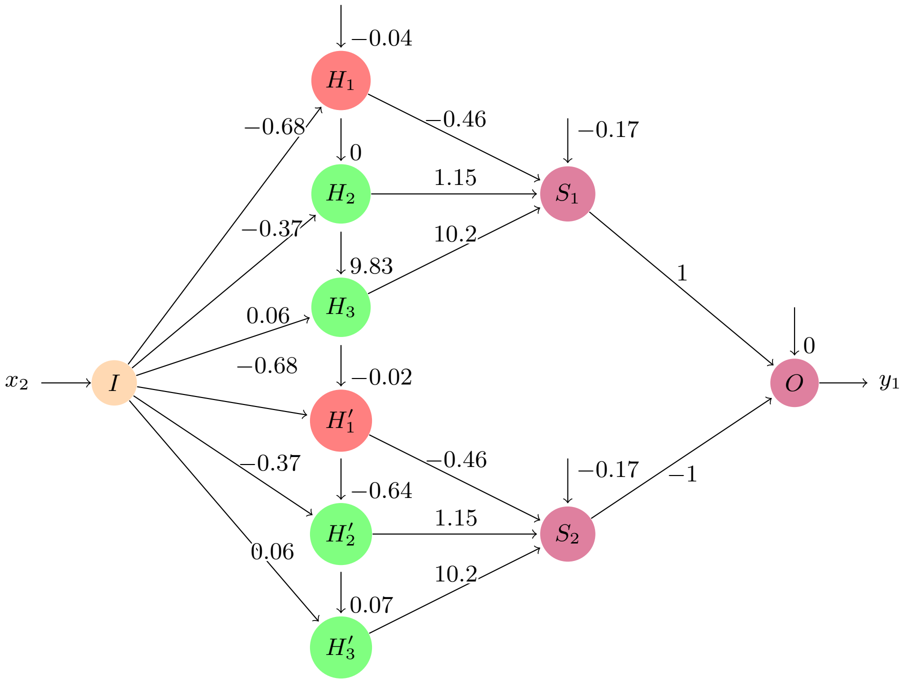

Fig. 4. Difference network NN_1^{diff}, i.e., input x_1 removed. We omit all edges with weight zero.

5 Related Work

This work is mainly based on [1], and there have been several extensions or work closely related to the paper. One extension looks in 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 a lot of the same work. Another example identifying nodes in a network which always produces an output almost zero (thus enabling the removal of them) [9]. In [10] the authors reformulates a DNN minimization problem as a DNN verification problem, and constructs a provably minimal network (which is still sufficiently close to the original).

Moreover, there are different approaches towards abstracting neural networks, see, 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*, see, e.g., [3]. However, since we aim for an approximated result, our method is allowed a bit more leeway when discarding input dimensions. We have not had time to look into how this premise could affect other popular feature selection methods.

6 Conclusions

In this paper we propose a new methodology for removing inputs by creating an over/under-approximating network. We present a method for using this to detect insignificant inputs.

6.1 Future Work

Our plan is to extend this work to handle multiple hidden layers. We will try to estimate the magnitude of the over-estimations, and how the structure and training of the network can affect it. Furthermore, we wish to investigate the capability of detecting insignificant inputs in more complicated examples, and in particular applying it to an industrial use case.

Acknowledgements.

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

References

1. 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.
2. 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.

3. V. Kumar and S. Minz, “Feature selection: a literature review,” *SmartCR*, vol. 4, no. 3, pp. 211–229, 2014.
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.