Electronic Communications of the EASST
Volume 82 (2022)

11th International Symposium
on Leveraging Applications of Formal Methods, Verification
and Validation
-
Doctoral Symposium, 2022

Introduction to Symbolic Execution of Neural Networks—Towards Faithful and Explainable Surrogate Models

Maximilian Schlüter, Gerrit Nolte

27 pages

Guest Editors: Sven Jörges, Salim Saay, Steven Smyth
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Introduction to Symbolic Execution of Neural Networks— Towards Faithful and Explainable Surrogate Models

Maximilian Schlüter[1], **Gerrit Nolte**[1]

[1] {maximilian.schlueter, gerrit.nolte}@tu-dortmund.de
Department of Computer Science
TU Dortmund University, Germany

Abstract: Neural Networks are inherently opaque machine learning models and suffer from uncontrollable errors that are often hard to find during testing. Yet no other model can attain their performance in current ML tasks. Thus, methods are needed to explain, understand, or even gain trust in neural networks and their Decisions. However, many existing explainability methods are abstractions of the true model, thus not providing reliable guarantees. For safety critical tasks, both rigorous explanations and state-of-the-art predictive performance are required. For neural networks with piece-wise linear activation functions (like ReLU), it is possible to distill the network into a surrogate model that is both interpretable and faithful using *decompositional rule-extraction*. We present a simple-to-follow introduction to this topic building on a well-known technique from traditional program verification: *symbolic execution*. This is done in two steps: First, we reformulate a neural network into an intermediate imperative program that consist of only if-then-else branches, assignments, and linear arithmetic. Then, we apply symbolic execution to this program to achieve the decomposition. Finally, we reintroduce a decision-tree like data structure called *Typed Affine Decision Structure* (TADS) that is specifically designed to efficiently represent the symbolic execution of neural networks. Further, we extend TADS to cover partial symbolic execution settings, which mitigates the path explosion problem that is a common bottleneck in practice. The paper contains many examples and illustrations generated with our tool.

Keywords: Symbolic Execution, Neural Networks, Explainable AI (XAI), Decision Trees, Preimage Partition, Linear Region Decomposition, Model Distillation, Rule Extraction, White Box Model

1 Introduction and Motivation

Over the past decade, deep learning and its neural network models have not only brought machine learning to the forefront of computer science research, but also attracted increasing attention from practical applications. Today, neural networks are a staple technology in computer vision [TKT18] and have rapidly advanced the state of the art in natural language processing [VSP+17, BCE+23], being able to generate text that is almost indistinguishable from text written by humans. In addition, neural networks have enabled machine learning systems to achieve top to superhuman levels of performance in several complex, well-studied board and video games

[VBC$^+$19, SSS$^+$17]. When data and computing resources are plentiful, neural networks can solve even difficult problems with minimal human guidance.

The Issue of Opacity. At the same time, neural networks suffer from one fundamental problem: Opacity [Bur16]. Because neural networks are trained from data rather than built by hand, their behaviour is difficult to control directly. Moreover, the structure of neural networks makes them almost impossible for humans to understand, and errors introduced by the training process are difficult to detect [BH21]. This is a significant barrier to the practical use of neural networks: At best, users are faced with a technology they do not fully understand and distrust; at worst, unpredictable failures of neural networks can endanger human lives in safety-critical domains [Rud19]. Researchers from multiple disciplines join forces to tackle this problem in *explainable AI (XAI)*.

Attribution. The attribution problem is concerned with finding which elements of an input vector were "important" for the output of a neural network. Attribution methods like LIME [RSG16], SHAP [LL17] or LRP [MBL$^+$19, BBM$^+$15] are hallmark methods in the field of explainable AI, but are limited in their rigidity: The results of these methods tend to be very understandable but not particularly closely linked to the actual behavior of the neural network. For example, LIME provides a linear surrogate model of the true model that resembles it in a local neighborhood. Therefore it can neither capture non-linear behavior nor the global nature of the true model.

Adversarial Examples. Adversarial examples are perhaps one of the most infamous examples of neural network unreliability. As shown by [GSS14, SZS$^+$14, AEIK18], neural network predictions can in many cases be arbitrarily manipulated. Concretely, even slight perturbations of the neural networks' input, often imperceptible to humans, can drastically change the networks' prediction.

Symbolic execution can be used to find adversarial examples within the same linear region as the given input [GWZ$^+$18, CHH$^+$18, UNP$^+$21]. This yields adversarial examples that are much more similar to the original input than the adversarial examples found by other methods.

Symbolic Execution. Symbolic execution is a technique from the field of automated program verification that aims to explore the different execution paths that a program can take [Kin76, BEL75]. Conceptually, it is based on the symbolic representation of input variables: Instead of concrete values, each input is assigned a symbolic variable and, as the program is evaluated, each assignment to a variable is done symbolically and each branching statement is fully explored.

A key characteristic of symbolic execution is its holistic exploration. As symbolic execution covers every single program path, it can be used to find even obscure and rare bugs that a simple testing approach would miss. On the flipside, symbolic execution of neural networks can become quite expensive as the number of potential paths explodes exponentially. This is called the state-space or path-explosion problem [Val05].

Symbolic Execution of Neural Networks. As mentioned before, the chaotic behavior of neural networks poses critical hurdles for their adoption into practical use cases. Rare errors are especially problematic as neural network behavior can vary so drastically that even large-scale testing approaches might miss some obscure errors.

Symbolic execution promises a natural fix for this. By its nature, symbolic execution exhaustively finds all execution paths that a program can take and therefore gives a complete, precise and easy-to-handle view on the program. Of course, symbolic execution is not applicable to any neural network. In the general case, repeated applications of complex non-linear functions makes symbolic execution impossible.

Special Properties of ReLU Neural Networks. However, there exists a class of neural networks that lends itself well to symbolic execution: *piece-wise linear neural networks*. In piece-wise linear neural networks, the non-linear activation functions are *piece-wise linear*, i.e., their input space can be partitioned such that they behave linearly on each part of that partition. The most important instance of this class are neural networks using the ReLU activation function [JKRL09, NH10, GBB11], which is regarded as *the* default choice for activation functions in most tasks [GBC16].

In fact, piece-wise linear neural networks are an extremely benign use case for symbolic execution: Symbolic execution of ReLU neural networks is loop-free, can be efficiently represented and yields a precise, polyhedral partition of the input set into a set of regions, each of which corresponds to precisely one program path.

Linear Regions. By performing symbolic execution on piece-wise linear neural networks, one essentially decomposes the network into its linear regions. To our knowledge, the first papers that extensively considered the piece-wise linear nature of ReLU neural networks were the works of [PMB13, MPCB14]. These are concerned with bounding the total number of linear regions of a ReLU neural network. After that, more publications bounded and counted the number of linear regions [RPK$^+$17, ABMM18, STR18], and examined their properties [HR19a, ZW20]. In [HR19b, Hin21] the authors studied how linear regions are characterized by the state of all ReLUs in a neural network (also know as *activation patterns* or *ReLU configurations*), essentially providing a global decomposition.

Contribution and Outline. In this paper, we give an overview of symbolic execution in the context of piece-wise linear neural networks, summarizing the works of [GWZ$^+$18, UNP$^+$21, SNMS23]. We will recall the essential results of these papers with the goal of accessibility, i.e., providing more detailed descriptions accompanied by examples and illustrations. In Section 3 we give a brief summary of the relevant properties of neural networks. Then, we state a translation of neural networks into WHILE programs (Section 4). After these foundational works, we give an introduction to symbolic execution and discuss the domain-specific properties of symbolic execution when applied to neural networks (Section 5). Moreover, we discuss TADS, as introduced in [SNMS23], as a data structure that is specifically designed to exploit these unique properties of neural networks to achieve scalable symbolic execution. We also give an overview of existing work on (partial) symbolic execution of neural networks (Section 6) and discuss measures that are commonly taken to tackle the path explosion problem. Lastly, we introduce the notion of partial TADS, an extension of TADS that allows for partial symbolic executions (Section 7).

2 Related Work

In this section, we give an overview of related work, with a focus on applications and use cases of symbolic execution of neural networks. Besides the foundational works that are mentioned in Section 1 the following works are directly related to this paper.

Symbolic Execution. Symbolic execution of piece-wise linear neural networks was already performed in [GWZ$^+$18, SWR$^+$18, UNP$^+$21, SNMS23]. Gopinath et. al. [GWZ$^+$18] present a translation of a neural network into an imperative program. Then they analyse the resulting program of a CNN using a variant of symbolic execution, called concolic execution, to identify adversarial attacks and important pixels. Usman et. al. [UNP$^+$21] follow a similar approach. They translate a neural network into Java code and then apply symbolic execution to perform coverage based testing and finding adversarial examples. By emitting Java code, they can apply existing verification and analysis tools. The translation processes of both works [GWZ$^+$18, UNP$^+$21] are conceptually identical to the one presented here. Similar translations are also used in practice to train neural networks (e.g. PyTorch [PGM$^+$19], Tensorflow [AAB$^+$16]).

Linear Regions. We will recall and extend a domain specific data structure, called *Typed Affine Decision Structures (TADS)*, to organize the results of the symbolic execution of neural networks. Central to the this structure is a decomposition of a neural network into its linear region [SNMS23]. Similar decompositional approaches proved useful in robustness verification [GMD$^+$18, KHI$^+$19, TMM$^+$19, Bak21, ZWX$^+$22] (cf. Section 6), and post-hoc model-distillation for explainable AI [CHH$^+$18, SKS$^+$20]. A specific subfield of explainable AI is interested in distilling decision trees out of neural networks, termed decompositional rule-extraction. The resulting trees of [BB20, NKA20, LJ20, Ayt22] are syntactically similar to TADS. However, TADS also leverage the algebraic properties of neural networks [SNMS23].

3 Background: Piece-wise Linear Neural Networks

This section provides a brief introduction into (piece-wise linear) neural networks. For additional information the books [GBC16, DFO20] are recommended. Neural networks are perhaps one of todays most important machine learning models. Neural networks consist of a set of nodes called *neurons* which are organized into multiple layers. Neurons are connected by weighted edges to neurons in the previous layer and store an activation value. If every neuron is connected to every neuron in the previous layer, the network is called *fully connected.*

Defining Neural Networks. To evaluate a neural network, an input vector is assigned to the input layer of the neural network. Iteratively, the neurons from the next layer obtain a weighted sum of the activation values from their respective previous layer. Then, a non-linear *activation*

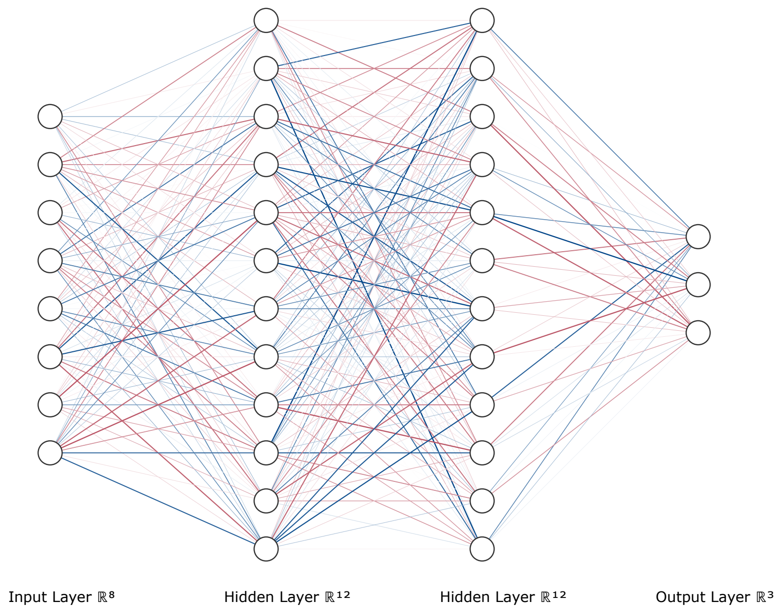

Input Layer $\mathbb{R}^8$
Hidden Layer $\mathbb{R}^{12}$
Hidden Layer $\mathbb{R}^{12}$
Output Layer $\mathbb{R}^3$

Figure 1: Example of a simple *feed-forward fully-connected* neural network. One can see the typical layer structure. In this case, the network has one input, and one output layer, and two hidden layers. Edges with larger associated weights are drawn thicker. Edges with positives weights are marked blue, those with negative weights red. Bias values are excluded in this depiction. Made with NN-SVG [LeN19].

function is applied and the activations are propagated forward once more until the output layer is reached (*feed forward*). The activation values of the output neurons are then designated as the output of the neural network (c.f. Figure 1).

Mathematically, a neural network can be concisely described as an alternating sequence of linear functions (denoted as α) and non-linear activation functions (denoted as σ) [GBC16]:

$$\mathcal{N} = \alpha_l \circ \sigma \circ \cdots \circ \sigma \circ \alpha_1 \tag{1}$$

Weights and Edges. In traditional machine learning applications, the edge weights and therefore the concrete linear functions α_i $(1 \leq i \leq l)$ would result from an optimization process, called *training*, where the neural network is iteratively adjusted to (accurately) predict desired outputs on a given dataset (*supervised learning*). Training is usually based on gradient descent-like optimization techniques [GBC16]. In this paper, we consider only pre-trained neural networks and assume that the functions α_i are fixed and known.

The Activation Function. The activation function σ is an architectural design choice made a-priori by the user. The primary purpose of the activation function is to introduce "non-linearity" into the neural network, thereby drastically increasing the amount of functions that can be approximated. In principle, almost any non-linear, monotonous function can be chosen as an activation function and a wide variety of established choices exist [GBC16, ADIP21].

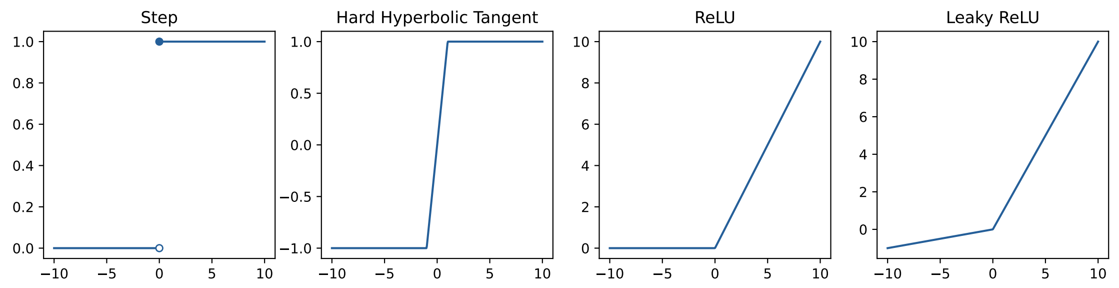
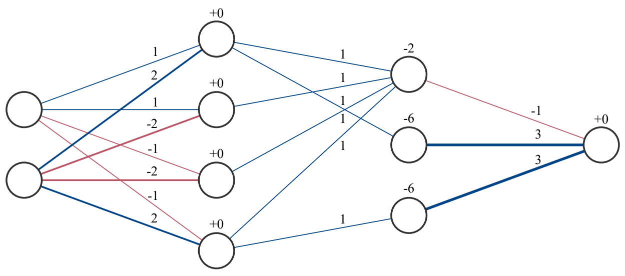


Figure 2: Common piece-wise linear activation functions [ADIP21].

ReLU. The most commonly used activation function at present is the ReLU function defined by $\phi = \max(0, x)$ (c.f., Figure 2). As an activation function, the ReLU function has proven successful in practical applications, combining convenient properties of linear functions with a sufficient degree of non-linearity, and is prominently recommended as the default choice of activation function for fully connected neural networks [GBC16]. The ReLU activation function comes with nice mathematical properties: It induces sparsity in the hidden units, is easy to compute and has a simple derivative.

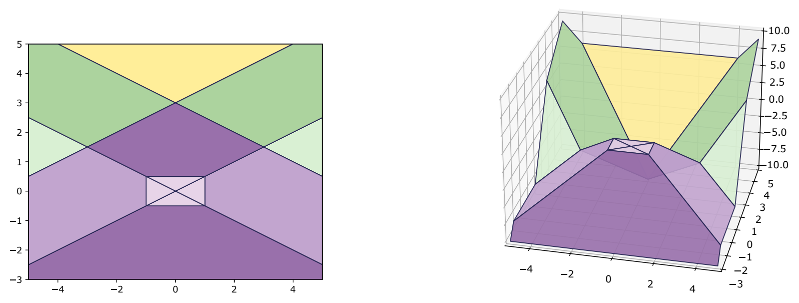
Piece-Wise Linearity. Moreover, the ReLU function is a *piece-wise linear* function. That is, it behaves linearly on the intervals $(-\infty, 0)$ and $(0, +\infty)$ respectively. As piece-wise linear functions are closed under composition, neural networks that use exclusively the ReLU activation function (or any other piece-wise linear function for that matter) also represent piece-wise linear functions. Due to this piece-wise linear structure, neural networks with ReLU activations lend themselves well to formal analysis and are typically considered in verification tasks [KBD+17, BLJ21]. For the remainder of this work, we use the ReLU function as a running example but note that all constructions immediately extend to other piece-wise linear activation functions.

Example 1 *Consider the following neural network:*



This ReLU neural network represents a piece-wise linear function $\mathbb{R}^2 \to \mathbb{R}$ that is visualized in Figure 3b*. Using mathematical notation, we can write its layer structure as:*

$$\mathcal{N} = \alpha_3 \circ \phi \circ \alpha_2 \circ \phi \circ \alpha_1 \ .$$



(a) Preimage partition. **(b)** Image.

Figure 3: Example of a piece-wise linear function. Within each polygon the function is linear. Colors are based on the gradient of the linear function.

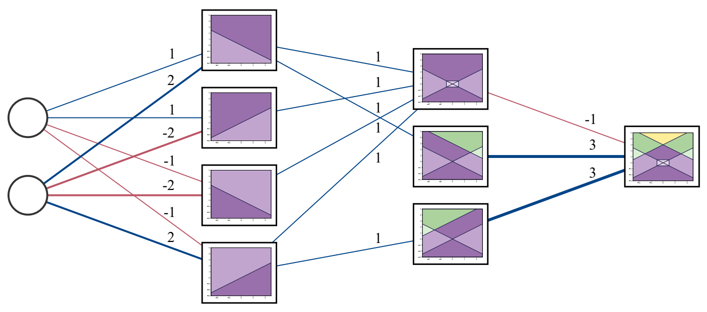


Figure 4: Contribution of each neuron to the linear regions. Illustration inspired by [HR19a].

Its weights and biases can be expressed as linear functions:

$$\alpha_1(\vec{x}) = \begin{pmatrix} 1 & 2 \\ 1 & -2 \\ -1 & -2 \\ -1 & 2 \end{pmatrix} \vec{x} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \alpha_2(\vec{x}) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \vec{x} + \begin{pmatrix} -2 \\ -6 \\ -6 \end{pmatrix}$$

$$\alpha_3(\vec{x}) = \begin{pmatrix} -1 & 3 & 3 \end{pmatrix} \vec{x} + \begin{pmatrix} 0 \end{pmatrix}$$

Like any piece-wise linear function, it separates the input space into multiple linear regions on which it each behaves linearly. The regions are shown in Figure 3a. Figure 4 illustrates how each layer contributes to the linear regions.

4 Neural Networks to Imperative Programs

Motivation. Symbolic execution can be applied to piece-wise linear neural networks resulting in an acyclic control flow graph (CFG). At first neural networks seem incompatible with symbolic execution since they do not exhibit a typical program structure. However, by just following the mathematical (operational) semantics of networks, one can quickly derive a simple set of rules for symbolic execution. To illustrate this process, we first translate a neural network into a

(WHILE) program. The CFG of the network can then be derived by applying (standard) symbolic execution to this program (cf. Section 5).

Composition. Decomposing a neural network along its layer structure results in a set of *well-understood* piece-wise linear functions. A piece-wise linear neural network is a sequence of linear functions (denoted as α) and piece-wise linear activation functions (denoted as σ):

$$\mathcal{N}(\vec{x}) = \big(\alpha_l \circ \sigma \circ \cdots \circ \sigma \circ \alpha_1\big)(\vec{x})$$

were each block $\sigma \circ \alpha_i$ constitutes a *layer*. Each component of the output of a layer is imagined as the activation of a neuron. Layers may have different widths, i.e., the input and output dimension of $\alpha_i \colon \mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}}$ can vary between layers. This structure can easily be traced on the program level, as function composition translates well to sequential juxtaposition of code blocks.

γ-rule (Composition Rule)

```
1: function $\mathcal{N}(\vec{x})$
2:     $\vec{x} \leftarrow \alpha_1(\vec{x})$
3:     $\vec{x} \leftarrow \sigma(\vec{x})$
4:     $\vdots$
5:     $\vec{x} \leftarrow \sigma(\vec{x})$
6:     $\vec{x} \leftarrow \alpha_l(\vec{x})$
7:     return $\vec{x}$
```

Note that we make use of the concept of *variable shadowing* to improve readability, that is, the type (i.e., the dimension) of $\vec{x}$ may change with each assignment. Based on this decomposition, it is now possible to translate α and σ locally, i.e., independently of previous layers.

Linear Functions. Linear functions lend themselves well for symbolic execution as they are compatible with many operations. For an input vector $\vec{x} = (x_1, \ldots x_n)^\top$, the effect of α on $\vec{x}$ can be described as a weighted sum (*linear combination*):

$$\alpha(\vec{x}) = (w_{i,1}x_1 + \cdots + w_{i,n}x_n + b_i)_{1 \le i \le n}$$

For brevity, we express these sums compactly using matrix multiplication:

$$(w_{i,1}x_1 + \cdots + w_{i,n}x_n + b_i)_{1 \le i \le n} = (\langle W_{i,\bullet}, \vec{x}\rangle + b_i)_{1 \le i \le n} = \boldsymbol{W} \cdot \vec{x} + \vec{b} \qquad (2)$$

Thus, for each x_i we get an update rule based on the previous value of $\vec{x}$.

α-rule (Linear Rule)

```
1: function $\alpha(\vec{x}) : \mathbb{R}^n \to \mathbb{R}^m$
2:     return $\boldsymbol{W} \cdot \vec{x} + \vec{b}$
```

Piece-wise Linear Functions. For piece-wise linear activation functions symbolic execution proceeds by a case discrimination of its linear regions. Let $\sigma\colon \mathbb{R}^n \to \mathbb{R}^n$ be a piece-wise linear activation function with k linear regions $Q_1,\dots,Q_k \subseteq \mathbb{R}^n$. Then one can decompose σ into its linear components using case discrimination [GZB94]:

$$\sigma(\vec{x}) = \begin{cases} \ell_1(\vec{x}) & \text{if } \vec{x} \in Q_1 \\ \vdots & \\ \ell_k(\vec{x}) & \text{if } \vec{x} \in Q_k \end{cases}$$

where $\ell_i\colon \mathbb{R}^n \to \mathbb{R}^n$ is the linear function of region Q_i ($1 \le i \le k$). The linear functions ℓ_i are handled as in the first rule. Case discrimination can be encoded using if-then-else constructs:

```
1: if $\vec{x} \in Q_1$ then
2:     $\vec{x} \leftarrow \ell_1(\vec{x})$
3:     $\vdots$
4: else if $\vec{x} \in Q_n$ then
5:     $\vec{x} \leftarrow \ell_k(\vec{x})$
```

In order to ensure that the definition is well-defined, the linear regions $Q_1,\dots,Q_k \subseteq \mathbb{R}^n$ must form a *partition* of $\mathbb{R}^n$. Additionally, the regions of the partition Q_i must be *convex polytopes* (which is the case for all considered activation functions). Therefore, the linear regions form a so-called *polyhedral partition* of $\mathbb{R}^n$ [GZB94]. That is, they obey the following laws (for all $1 \le i, j \le k$):

1. the regions Q_i are convex polytopes
2. the regions are pairwise disjoint, i.e., $Q_i \cap Q_j = \emptyset \iff i \neq j$
3. the whole input space is covered, i.e., $\bigcup_{1\le i\le k} Q_i = \mathbb{R}^n$

Note that by the second property, each $\vec{x} \in \mathbb{R}^n$ satisfies exactly one branch in the above program.

Improving on that, conditions over $\vec{X}$ can be simplified since linear regions are always *convex*. There are many definitions of convexity, but the following is most useful in this context: A polytope $Q \subseteq \mathbb{R}^n$ is convex iff there exists a matrix $\boldsymbol{A}$ and a vector $\vec{b}$ such that all points $\vec{v} \in Q$ satisfy the inequality:

$$q(\vec{v}) \le 0 \quad \text{where} \quad q(\vec{v}) := \boldsymbol{A}\vec{v} + \vec{b}. \tag{3}$$

Thus, we can rewrite the conditions using linear functions, a fact that will allow significant simplifications later. Predicates of the form Equation (3) will be called *linear predicate* from now on. Let $q_1,\dots,q_k$ be the linear functions that are derived from the convex regions $Q_1,\dots,Q_k$ according to Equation (3), then we can improve our program code as follows:

σ-rule (Piece-wise Linear Update Rule)

```
function $\sigma(\vec{x}) : \mathbb{R}^n \to \mathbb{R}^n$
    if $q_1(\vec{x}) \leq \vec{0}$ then
        $\vec{x} \leftarrow \ell_1(\vec{x})$
    else if $q_2(\vec{x}) \leq \vec{0}$ then
        $\vec{x} \leftarrow \ell_2(\vec{x})$
        $\vdots$
    else if $q_n(\vec{x}) \leq \vec{0}$ then
        $\vec{x} \leftarrow \ell_k(\vec{x})$
    return $\vec{x}$
```

Partial Activation Functions. Activation functions σ are generally applied component-wise to the output.

$$\sigma(\vec{x}) = (\varsigma(x_1), \dots, \varsigma(x_n))^\top$$

Here $\varsigma\colon \mathbb{R} \to \mathbb{R}$ is a real valued function. For example, the multi-valued ReLU function is a component-wise application of the maximum

$$\big(\phi(\vec{x})\big)_i = \max\{0, x_i\}$$

To emphasize this behavior, one can also decompose the activation function into a set of partial activation functions that act only on one specified neuron and leave the others unchanged:

$$\sigma = \sigma_n \circ \dots \circ \sigma_1$$
$$\big(\sigma_j(\vec{x})\big)_i = \begin{cases} \varsigma(x_i) & \text{if } i = j \\ x_i & \text{if } i \neq j \end{cases}$$

Resulting in the final decomposition

$$\mathcal{N}(\vec{x}) = \big(\alpha_l \circ (\sigma_{n_l} \circ \dots \circ \sigma_1) \circ \dots \circ (\sigma_{n_2} \circ \dots \circ \sigma_1) \circ \alpha_1\big)(\vec{x})$$

This has the benefit that then each neuron can be treated independently from others. For example, defining the multi-valued ReLU function $\phi\colon \mathbb{R}^n \to \mathbb{R}^n$ as previously presented using case discrimination requires 2^n mutually exclusive branches:

$$\phi(\vec{x}) = \begin{cases} \vdots & \\ (x_1, 0, 0, \dots, x_n) & \text{if } x_1 \geq 0 \land x_2 < 0 \land x_3 < 0 \land \dots \land x_n \geq 0 \\ \vdots & \end{cases}$$

However, an equivalent definition using partial ReLUs only requires the composition of n partial ReLUs, each with 2 cases:

$$\phi_i(\vec{x}) = \begin{cases} (x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n)^\top & \text{if } x_i \geq 0 \\ (x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)^\top & \text{otherwise} \end{cases}$$

Thus, the corresponding program also consists of exponentially less branches, while the number of program paths through those branches is unaffected.

Final Program. Combing the γ, α and σ-rule results in a complete translation of a piece-wise linear neural network into a WHILE program. To summarize, every program that is generated in this fashion from a neural network consists of just two types of program statements:

Linear Branches are if statements over linear predicates $q(\vec{v}) \leq \vec{0}$. These can test whether the input $\vec{x}$ lies in a specific (convex) linear region of the activation function.

Linear Updates are assignments based on linear functions α_i. These encode the network's manipulation of the input $\vec{x}$ introduced by the trained weights and biases of each layer.

Example 2 *Consider the neural network given in Example 1. Its WHILE program can be derived by a systematic application of the above rules. By using the partial ReLU function, the program code is reduced in size. Notice that each if corresponds to one partial ReLU. Technically, the partial ReLU function has two regions. However, the second region corresponds to the identity function and has therefore no effect. For readability, these branches are omitted in the following code.*

```
function $\mathcal{N}(x_1, x_2)$
    $\vec{x} \leftarrow (x_1 + 2x_2,\ x_1 - 2x_2,\ -x_1 - 2x_2,\ -x_1 + 2x_2)^\top$
    if $x_1 \leq 0$ then
        $x_1 \leftarrow 0$
    if $x_2 \leq 0$ then
        $x_2 \leftarrow 0$
    if $x_3 \leq 0$ then
        $x_3 \leftarrow 0$
    if $x_4 \leq 0$ then
        $x_4 \leftarrow 0$
    $\vec{x} \leftarrow (x_1 + x_2 + x_3 + x_4 - 2,\ x_1 - 6,\ x_4 - 6)^\top$
    if $x_1 \leq 0$ then
        $x_1 \leftarrow 0$
    if $x_2 \leq 0$ then
        $x_2 \leftarrow 0$
    if $x_3 \leq 0$ then
        $x_3 \leftarrow 0$
    return $-x_1 + 3x_2 + 3x_3$
```

5 Symbolic Execution of Neural Networks

The structure of a neural network is simple: each layer receives an input vector, manipulates it based on piece-wise linear functions, and then passes on the result to the next layer. The simplicity of this structure was highlighted by the translation into a WHILE program in Section 4.

The syntactic simplicity stands however in hard conflict with the complex semantics of neural networks.

Model Complexity. Actually, this is—in part—a driving force for the success of modern deep learning. Scalable training techniques require fast evaluation and adjustments of machine learning models. Yet, at the same time, complex use cases require correspondingly potent semantics of ML models to express suitable solutions. Neural networks overcome this gap as their semantic complexity grows exponentially with their training parameters θ with respect to a variety of complexity measures [BS14, FRH+19, MPCB14]. Additionally, as neural networks exhibit almost no control flow and many independent calculations, they can be executed utilizing the capacity of modern GPUs for parallelization, vectorization, and throughput.

On the other hand, this comes at the price of comprehensibility. Neural networks are considered opaque "black-box" models. Their data flow is chaotic (i.e., sensitive to small changes), non-linear, and parallelized (therefore intractable for humans).

Through symbolic execution, we decompose ("isolate") the data flow into a set of *program paths* such that each path is neither chaotic nor parallelized. In fact, each path will behave linearly on the inputs of the considered neural network. Linear correlation can intuitively be handled by humans. To achieve this decomposition, the control flow is completely unrolled and unmerged.

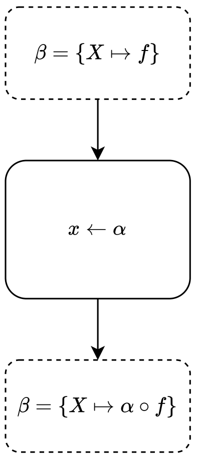
5.1 Applying Symbolic Execution

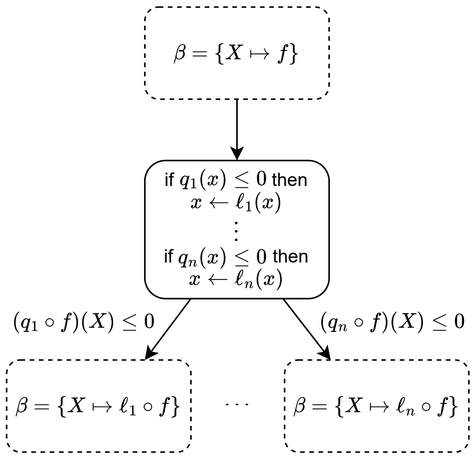
Symbolic execution is a holistic and abstract (i.e., symbolic) interpretation of a program. Control flow is branched explicitly resulting in a (theoretically) complete unrolling of the program. This results in a decomposition of the program into a set of independent executions, called *program paths*. Thereby each such path has neither incoming nor outgoing control flow and thus a fixed variable valuation. Thus it can serve as a model for analyses and optimizations.

On the other hand, the unrolling results in an exponential blow up (state-space-explosion). Therefore one reduces the number of branches through multiple techniques: (i) removing infeasible paths, (ii) merging paths where variable valuations are identical from that point onwards, (iii) using complex mathematical terms to express the values of variables.

The result of symbolic execution is represented as a tree, called *control flow graph* (CFG). Each path of the tree corresponds to exactly one program path. If-statements are converted to labeled branches, each label describing the condition that has to hold to take this path. As the variable valuations can be tracked for each path, conditions can be simplified by substituting in the variables.

Symbolic Rules. The concrete rules for symbolic execution are straightforward given the simple structure of the WHILE programs derived in Section 4. These programs have two types of statements: linear updates and linear branches. The following two rules present how the effect of each statement can be handled. Let β denote a variable valuation, X the symbolic input variable, and f an arbitrary function. Then program assignment and branching can be expressed as:


$\beta = \{X \mapsto f\}$
$x \leftarrow \alpha$
$\beta = \{X \mapsto \alpha \circ f\}$


$\beta = \{X \mapsto f\}$
if $q_1(x) \leq 0$ then $x \leftarrow \ell_1(x)$ ⋮ if $q_n(x) \leq 0$ then $x \leftarrow \ell_n(x)$
$(q_1 \circ f)(X) \leq 0$
$(q_n \circ f)(X) \leq 0$
$\beta = \{X \mapsto \ell_1 \circ f\}$
$\cdots$
$\beta = \{X \mapsto \ell_n \circ f\}$

Notice that all rules produce acyclic graphs.

Example 3 *Consider the following short WHILE program of some neural network $\alpha_2 \circ \sigma \circ \alpha_1$.*

```
$\vec{x} \leftarrow \alpha_1(\vec{x})$
**if** $q_1(\vec{x}) \leq \vec{0}$ **then**
    $\vec{x} \leftarrow \ell_1(\vec{x})$
**else if** $q_2(\vec{x}) \leq \vec{0}$ **then**
    $\vec{x} \leftarrow \ell_2(\vec{x})$
$\vec{x} \leftarrow \alpha_2(\vec{x})$
```

Applying the above rules for symbolic execution results in the tree structure presented in Figure 5. This symbolic CFG encodes the following two program paths with corresponding variable valuation.

$$\begin{aligned}(q_1 \circ \alpha_1)(\vec{X})) \leq \vec{0} &\quad\mapsto\quad (\alpha_2 \circ \ell_1 \circ \alpha_1)(\vec{X}))) \\ (q_2 \circ \alpha_1)(\vec{X})) \leq \vec{0} &\quad\mapsto\quad (\alpha_2 \circ \ell_2 \circ \alpha_1)(\vec{X})))\end{aligned}$$

Branching. Whenever an if statement is encountered, all possible executions are explored. Each branch of the if statement is associated with a condition that has to hold. In our case these conditions have a simple form ($1 \leq i \leq k$):

$$q_i(\vec{x}) \leq \vec{0}\,.$$

Substituting the current value of $\vec{X}$ into the condition gives the actual constraint imposed by the if statement. The value of $\vec{X}$ is given by the current variable valuation. In its most general form it can be stated as $\beta = \{\vec{X} \mapsto f\}$ for some linear function f. Then the path conditions are

$$\psi_i = (q_i \circ f)(\vec{X}) \leq \vec{0}$$

for each q_i. Each constraint ψ_i is associated with one branch. The body of this branch is added as a new node. The edge leading to this new node is labeled with the *path constraint* ψ_i.

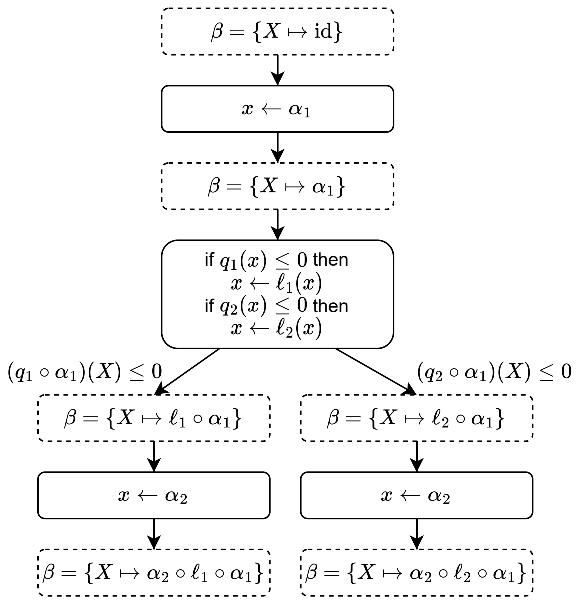

$\beta = \{X \mapsto \mathrm{id}\}$
$x \leftarrow \alpha_1$
$\beta = \{X \mapsto \alpha_1\}$
if $q_1(x) \le 0$ then
$x \leftarrow \ell_1(x)$
if $q_2(x) \le 0$ then
$x \leftarrow \ell_2(x)$
$(q_1 \circ \alpha_1)(X) \le 0$
$(q_2 \circ \alpha_1)(X) \le 0$
$\beta = \{X \mapsto \ell_1 \circ \alpha_1\}$
$\beta = \{X \mapsto \ell_2 \circ \alpha_1\}$
$x \leftarrow \alpha_2$
$x \leftarrow \alpha_2$
$\beta = \{X \mapsto \alpha_2 \circ \ell_1 \circ \alpha_1\}$
$\beta = \{X \mapsto \alpha_2 \circ \ell_2 \circ \alpha_1\}$

Figure 5: Full CFG for the WHILE program of Example 3.

Variable Domain. For each path in the program, symbolic execution proceeds by collecting updates that are applied to the program variables. By the structure of the two symbolic rules variable valuations β have the form:

$$\beta = \{\vec{X} \mapsto f_l \circ \cdots \circ f_1\} .$$

In either case, the update functions f_i are *linear*. This is a key difference when applying symbolic execution to neural networks in contrast to applying it to regular programs. It allows substantial simplifications using well-known identities from linear algebra.

A sequence of updates to $\vec{X}$ can be simplified by using associativity and completeness of linear functions with respect to composition:

$$f_2(f_1(\vec{X})) = \underbrace{(f_2 \circ f_1)}_{\text{linear}}(\vec{X})$$

Calculating the composition of two linear functions just involves matrix multiplication which can be performed efficiently:

$$(f_2 \circ f_1)(\vec{X}) = \underbrace{(\boldsymbol{W}_2 \boldsymbol{W}_1)}_{=W'} \vec{X} + \underbrace{(\boldsymbol{W}_2 \vec{b}_1 + \vec{b}_2)}_{=b'} \tag{4}$$

Notice that the concrete value of $\vec{X}$ is not needed to simplify according to this equation. In practice, one would always simplify the term of $\vec{X}$ whenever a new linear function is collected. Thus, in all steps of the symbolic execution $\vec{X}$ can be stored using only a matrix and a bias vector:

$$\beta = \{\vec{X} \mapsto (\boldsymbol{W}, \vec{b})\} .$$

Graph Rewriting. Finally, we may omit nodes in the graph that add no useful information. In most cases, the intermediate values of a neural network are not of interest. Therefore, we omit any variable valuation β but the last. Additionally, we no longer need the details of the WHILE program. The program is only used to aid in the symbolic execution. Thus the resulting tree is significantly reduced: It consists of labeled branches (ψ_i) and variable valuations (β) in its leaves.

TADS

TADS are a symbolic representation of a pice-wise linear neural network based on its acyclic CFG. They collect the path constraints from symbolic execution and arrange them in a decision tree. Each path of that decision tree corresponds to one linear region of the network. The terminals of paths store the respective linear function of the region as given by symbolic execution.

The following definition (cf. [SNMS23]) is the direct consequence of the given rules when considering the ReLU activation function. For piece-wise linear activation functions with more than two regions additional rewriting steps are needed to encode these regions using only binary splits. Although this process is well-understood, it is beyond the scope of this paper.

Definition 1 (Typed Affine Decision Structures) A TADS $T = (N, \rightarrow, \zeta)$ is a labeled binary tree with root ζ. Its nodes N and transitions $\rightarrow \subseteq N \times \{0,1\} \times N$ induce a decision structure consisting of the following two node types:

Decisions are linear inequalities $q(\vec{X}) = \langle \vec{w}, \vec{X} \rangle + b \leq 0$. Decision nodes have two successors, one if the conditions is satisfied and one if not.

Terminals are linear functions $\alpha(\vec{X}) = \boldsymbol{W}\vec{X} + \vec{b}$. Eponymously, they have no outgoing transitions.

All linear functions α and q in a TADS have the same input space $\mathbb{R}^n$. All terminals α also have the same output space $\mathbb{R}^m$. Both spaces are determined by the considered neural network. For given input dimension n and output dimension m we define the set of all TADS as $\Theta^{n \rightarrow m}$.

TADS are sequentially evaluated like a decision tree [SNMS23].

Definition 2 (TADS Evaluation) The semantic function of TADS

$$[\![\cdot]\!]_\Theta \colon \Theta^{n \rightarrow m} \rightarrow (\mathbb{R}^n \rightarrow \mathbb{R}^m)$$

is inductively defined as

$$\begin{aligned}
[\![q]\!]_\Theta(\vec{x}) &:= [\![p]\!]_\Theta(\vec{x}) && \text{if } q(\vec{x}) < 0 \text{ where } p \text{ is uniquely defined by } q \xrightarrow{0} p \\
[\![q]\!]_\Theta(\vec{x}) &:= [\![p]\!]_\Theta(\vec{x}) && \text{if } q(\vec{x}) \geq 0 \text{ where } p \text{ is uniquely defined by } q \xrightarrow{1} p \\
[\![\alpha]\!]_\Theta(\vec{x}) &:= \alpha(\vec{x}) && \text{if } \alpha \not\rightarrow
\end{aligned}$$

for a TADS $T = (N, \rightarrow, \zeta)$, with $q, p, \alpha \in N$. For convenience we introduce the shorthand $T(\vec{x}) := [\![\zeta]\!]_\Theta(\vec{x})$.

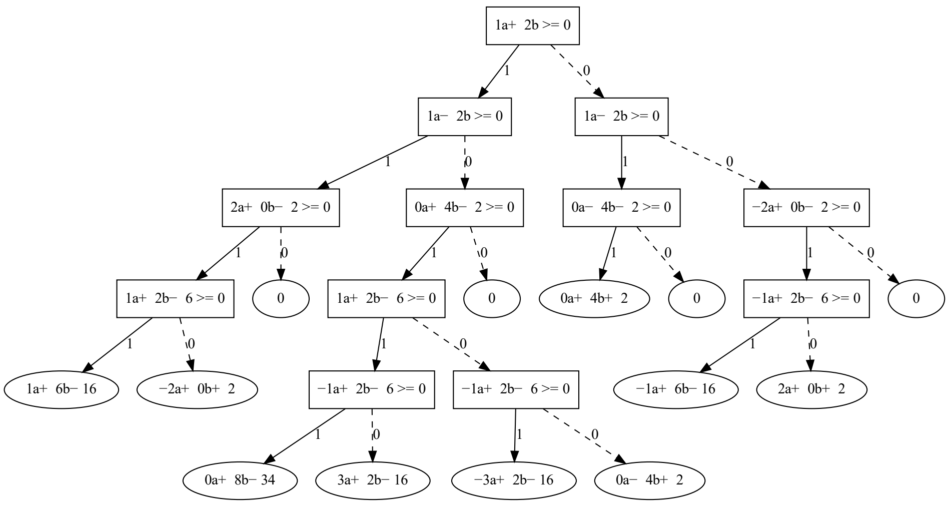


Figure 6: TADS of the neural network given in Example 1. For readability, this TADS was optimized using infeasible path elimination.

Example 4 *The TADS for the running example (c.f. Example 1) is shown in Figure 6. To gain further intuition for the structure, it is helpful to visualize the regions and function values encoded abstractly in a TADS using the matrix representation. For that, Figure 7 shows instead the linear region with the corresponding function values in the terminals.*

A direct consequence of the symbolic execution is the soundness and completeness of TADS. Both are summarized in the following theorem [SNMS23]:

Theorem 1 *Let $T \in \Theta^{n \to m}$ be the TADS derived from the piece-wise linear neural network $N \colon \mathbb{R}^n \to \mathbb{R}^m$ by symbolic execution. Then for all inputs $\vec{x} \in \mathbb{R}^n$ the following holds:*

$$N(\vec{x}) = T(\vec{x})$$

5.2 Interpreting the Resulting Structure

Paths and Regions A path in the CFG of symbolic execution always corresponds to a concrete run of the program. For neural networks, a run is the evaluation of a concrete input point. However, multiple inputs can take exactly the same path. When grouping possible inputs by this property, a partition of the preimage space is formed. In this function a TADS is similar to *binary space partition trees* (BSP trees) [TN87].

Neural Networks and TADS A TADS is a faithful, complete (global), and systematic surrogate model for the semantics of a piece-wise linear neural network. TADS are centered around the simplicity of linear functions and make use of their efficient matrix representation. Further, TADS are the result of a straightforward decomposition of a neural network along its linear functions. This decomposition can easily be achieved by applying symbolic execution. One can

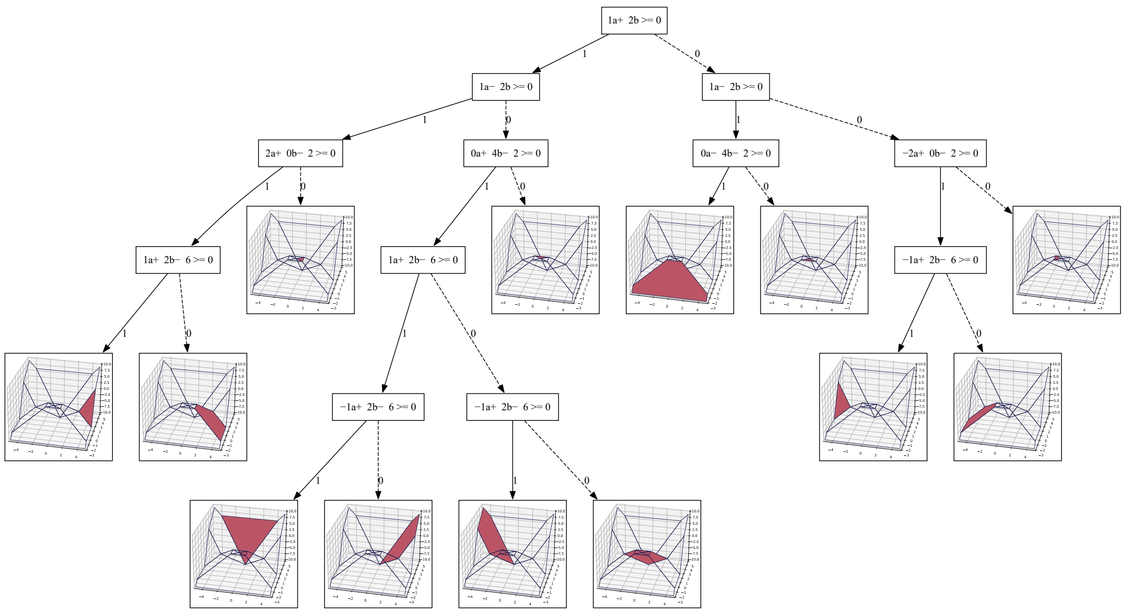


Figure 7: TADS of Figure 6 where the terminals show the linear region with the associated function value.

therefore suggest that TADS are a natural representation of piece-wise linear neural network's semantics.

Although the systematic and complete aspect of TADS was only recently considered [Ayt22, SNMS23], the decomposition of a neural network into its linear regions was studied in many publications (see Section 2).

The completeness of TADS also comes with disadvantages: As each neuron induces a binary split of each linear region, the number of linear regions (and therefore TADS leaves) can reach an upper bound of 2^n. Although optimization techniques exists (infeasible path elimination [MBN$^+$23], linear preprocessing [NSMS23]), scalability of full TADS will probably always be a problem.

For most neural networks, especially when one considers a restricted input region, this worst-case bound is not reached. Nevertheless, theoretical and practical results still indicate an exponential growth of linear regions [PMB13, MPCB14, ZW20, HR19a, STR18]. Furthermore, it is folklore in deep learning that more neurons in a neural network (usually) lead to an improvement in performance [RT18, GBC16]. Therefore, most models of interest have at least $n \geq 100$ neurons, with much higher values being more common. Any representation of a neural networks semantics that precisely describes each linear region is therefore (currently) infeasibly large. This problem is known in program verification as state-space-explosion.

6 Related Work: Use Cases obey (Partial) Symbolic Execution

A special challenge that is implicitly tackled in all related approaches is the path explosion problem: For most neural networks that are of practical interest, the aforementioned path explosion leads to prohibitively complex problems. As a result, *an exhaustive symbolic execution is almost always infeasible*. As a consequence, most approaches using symbolic execution for neural networks in practice drastically limit the parts of the symbolic execution tree that they consider. They perform only a *partial* symbolic execution of the neural network, i.e., they either consider only a subset of possible paths or they do not symbolically execute paths to their terminal state.

In the following we present a practical use case where partial symbolic execution is applied to neural networks: neural network verification. After this motivation, we will present an extension of TADS to partial symbolic execution in Section 7. The resulting structure can be applied to the following use cases.

Neural Network Verification. Neural network verification entails the task of proving that a given neural network satisfies some given property, usually of the form [LAL+21]:

$$\forall \vec{x} \big(\operatorname{pre}(\vec{x}) \implies \operatorname{post}(N(\vec{x})) \big) \ .$$

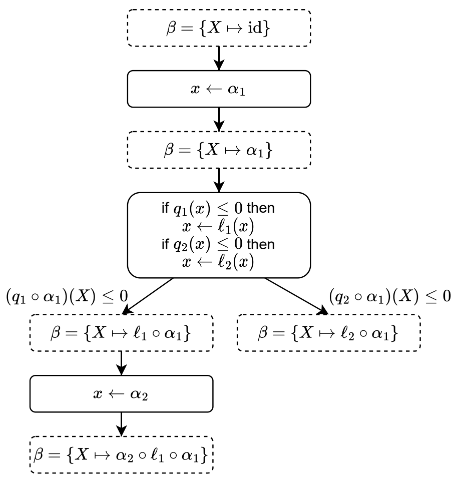
The problem of neural network verification subsumes the problem of adversarial examples as the absence of adversarial examples can be posed as a property of a neural network.

Symbolic execution lays an ideal groundwork for verification which has, in the context of neural networks, been utilized by [GWZ+18, CHH+18, UNP+21] for specific *local* tasks like attribution and finding adversarial examples. In these cases, the path explosion problem can be avoided by considering only local properties. Concretely, the authors consider only preconditions $\operatorname{pre}(\vec{x})$ that fix many of the input variables to concrete values, limiting the number of paths to explore.

In more general verification settings, the number of paths may be much higher. In these cases, other methods are required to limit path explosion. One such method is abstract interpretation. Abstract interpretation can be leveraged to decide whether certain subtrees of a symbolic execution graph can be deemed safe and therefore pruned from the search for a counterexample. This technique proved successful in recent years as a bounding heuristic in neural network verification [WZX+21, Bak21].

7 Partial Symbolic Execution of Neural Networks using TADS

As demonstrated in the previous section, partial symbolic execution has been successfully applied to numerous problems in the context of neural networks. However, the concrete implementations and methodologies used in these works are often task specific and do not generalize well to new problems. In this section, we introduce *partial TADS* to that end. A partial TADS can be seen as an intermediate version of a full TADS which only partially reflects the behavior of the considered neural network. It can be generated by a straightforward adaptation of the previously introduced rules for symbolic execution. Partial TADS allow users to iteratively explore any program path of interest of a neural network without ever actually computing the full TADS. This enables task-driven construction of partial TADS such that only the required subset of paths is actually explored.



$\beta = \{X \mapsto \mathrm{id}\}$
$x \leftarrow \alpha_1$
$\beta = \{X \mapsto \alpha_1\}$
if $q_1(x) \leq 0$ then $x \leftarrow \ell_1(x)$ if $q_2(x) \leq 0$ then $x \leftarrow \ell_2(x)$
$(q_1 \circ \alpha_1)(X) \leq 0$
$(q_2 \circ \alpha_1)(X) \leq 0$
$\beta = \{X \mapsto \ell_1 \circ \alpha_1\}$
$\beta = \{X \mapsto \ell_2 \circ \alpha_1\}$
$x \leftarrow \alpha_2$
$\beta = \{X \mapsto \alpha_2 \circ \ell_1 \circ \alpha_1\}$

Figure 8: Example for a partial symbolic Execution. In contrast to Figure 5 the right branch is not explored to its end.

Partial Symbolic Execution. Whereas symbolic execution is characterized by its holistic exploration of programs paths, partial symbolic execution may choose to not explore all branches. This has two direct consequences:

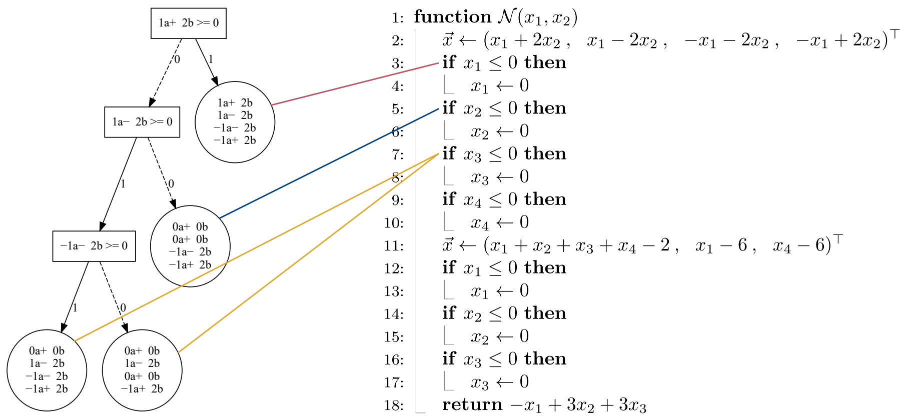
1. Not every program path is explored.
2. Some program paths are not explored to their end.

Example 5 *Considering again the short WHILE program (c.f. Example 3):*

```
1: $\vec{x} \leftarrow \alpha_1(\vec{x})$
2: if $q_1(\vec{x}) \leq \vec{0}$ then
3:     $\vec{x} \leftarrow \ell_1(\vec{x})$
4: else if $q_2(\vec{x}) \leq \vec{0}$ then
5:     $\vec{x} \leftarrow \ell_2(\vec{x})$
6: $\vec{x} \leftarrow \alpha_2(\vec{x})$
```

The full symbolic execution of this program was already presented in Figure 5. As described, partial symbolic execution may choose to not explore all branches. Since this program has only two branches, there are four possible explorations: (i) explore all branches, (ii) explore only the left or right branch, (iii) explore no branch. The first case corresponds to the full symbolic execution, the latter three to partial symbolic execution. An example where only the left branch is explored is given in Figure 8.

Partial TADS. We introduce partial TADS to match this behavior. Partial TADS are similar to full TADS. They are also derived from a neural network using symbolic execution. Therefore they also decompose the network into regions. However, in their case the symbolic rules given on page 12 are not applied holistically. Therefore, partial TADS are a subgraph of full TADS before the graph rewriting is applied. After omitting all but the last variable valuation β partial and full TADS differ as they explore program paths to different depths. As a result, partial TADS



```
function $\mathcal{N}(x_1, x_2)$
    $\vec{x} \leftarrow (x_1 + 2x_2\,,\ \ x_1 - 2x_2\,,\ \ -x_1 - 2x_2\,,\ \ -x_1 + 2x_2)^\top$
    if $x_1 \leq 0$ then
        $x_1 \leftarrow 0$
    if $x_2 \leq 0$ then
        $x_2 \leftarrow 0$
    if $x_3 \leq 0$ then
        $x_3 \leftarrow 0$
    if $x_4 \leq 0$ then
        $x_4 \leftarrow 0$
    $\vec{x} \leftarrow (x_1 + x_2 + x_3 + x_4 - 2\,,\ \ x_1 - 6\,,\ \ x_4 - 6)^\top$
    if $x_1 \leq 0$ then
        $x_1 \leftarrow 0$
    if $x_2 \leq 0$ then
        $x_2 \leftarrow 0$
    if $x_3 \leq 0$ then
        $x_3 \leftarrow 0$
    return $-x_1 + 3x_2 + 3x_3$
```

Figure 9: A partial TADS for the two layer neural network of Example 1 aligned with its corresponding program code (cf. Example 2). Colored lines indicate the statement up to which the respective leaf was progressed.

can have leaves that only reflect the neural network to a certain depth. Further, this depth can vary between leaves.

Definition 3 (Partial TADS) A partial TADS $T = (N, \rightarrow, \zeta)$ is a TADS with a relaxed constraint: Terminals α must not all have the same output space.

This relaxation allows that the leaves of partial TADS can represent the network to different depths, that is, they describe the output of different (hidden) layers. This is essential to store the variable valuations derived from partial symbolic execution as previously explained.

Evaluating a partial TADS. The evaluation of partial TADS coincides with full TADS. However, its results have a different interpretation. Standard TADS evaluation always corresponds to the output of the underlying neural network

$$\forall \vec{x}.\, T(\vec{x}) = N(\vec{x})$$

The evaluation of partial TADS P yields

$$\forall \vec{x} \exists r.\, P(\vec{x}) = N_r(\vec{x}).$$

where N_r is the neural network that consists only of the first r statements of N. In practice, one would want to include the last linear function, i.e., only consider odd values of r:

$$N = \alpha_l \circ \cdots \circ \sigma \circ \underbrace{\alpha_{0.5(r+1)} \circ \sigma \circ \cdots \circ \sigma \circ \alpha_1}_{N_r}$$

Thus, each path induces a division of the network into two parts: One part of the network (the statements up to r) are evaluated by the partial TADS in a white-box fashion, whereas the remaining part of the network (all statements after r) remains opaque.

Iterative Exploration. In practice, partial TADS are used to *iteratively* explore the linear regions of a considered neural network. Examples for this were given in Section 6. To enable an iterative exploration, one must be able to associate each terminal node of a partial TADS with a queue of pending operations. Fortunately, the simple program structure allows to do this efficiently. The next operation of a terminal corresponds to one of the two rules on page 12. Which rule is applicable is uniquely defined by the next program statement (either linear update or linear branch). Thus, for each terminal one only needs to store the next statements. And, as the program is fixed for all terminals, one can just provide pointers to the respective statement in one centrally stored version of the program.

To implement these pointers, we enumerate all statements of the program. This index can then be used to uniquely identify each statement. We call this index *statement numbers*. As each statement corresponds to one step of symbolic execution, the statement number of a terminal corresponds exactly to the number of symbolic execution steps applied. An example is given in Figure 9.

The full TADS can be seen as an upper bound of all partial TADS.

Exploring a path further. Any path in a partial TADS can be explored further by incrementally evaluating the next remaining statement of the network. We call this *progressing* a path. Progressing a path with terminal node α is done as follows:

1. Look up the the statement number $n = r(t)$ of α.
2. Find the n-th statement f of the underlying neural network N.
3. Perform the transformation corresponding to f on α.
4. Associate each new leaf of α with statement number $n+1$.

In essence, this involves applying the transformation (equivalent to the transformations used in full TADS) corresponding to the next statement of the neural network to α.

We say that a partial TADS is *complete* if all paths have been progressed to the end of the neural network. A complete partial TADS is by construction fully equivalent to the standard TADS and, by extension, also fully equivalent to the neural network N.

By incrementally progressing partial TADS, paths in the full TADS can be explored without being fully evaluated, saving time and memory. Of course, this comes at the cost of completeness: The core challenge of partial symbolic execution entails dealing with its inherent incompleteness. In the following section, we will give an overview over some use cases in related work that have used partial symbolic execution and discuss how they mitigate this issue of completeness and how their approach would be reflected in a partial TADS.

8 Conclusion and Outlook

In this paper, we gave an introduction to the concept of symbolic execution in the context of (piece-wise linear) neural networks, presenting a range of existing results from a traditional symbolic execution perspective.

As we showed, neural networks possess a variety of desirable properties with respect to symbolic execution, specifically: They are loop-free, side-effect-free and contain only linear conditions and assignments. As a consequence, symbolic execution of neural networks can be done completely, precisely and be represented efficiently. We also presented a new introduction to Typed Affine Decision Structures (TADS) as a domain specific data structure specifically designed to exploit these properties and act as a minimal, efficient representation of a neural networks symbolic execution graph.

Despite the beneficial properties of TADS, exhaustive symbolic execution of neural networks is still almost always infeasible. As a survey of related work showed, most prominent approaches leverage partial symbolic execution instead.

Partial symbolic execution of neural networks has shown promising results in related work, spanning from verification problems to adversarial attacks and explainability tasks. To this end, we introduced partial TADS, which serve as a novel, task specific extension of TADS to the problem of partial symbolic execution. We believe that (partial) TADS can serve as a unifying framework for future symbolic execution approaches and that they can serve as a basis for an efficient, general implementation.

In the future, we are keen to build upon (partial) TADS and explore a variety of use cases for symbolic execution in the context of neural networks.

Bibliography

[AAB$^+$16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[ABMM18] R. Arora, A. Basu, P. Mianjy, A. Mukherjee. Understanding Deep Neural Networks with Rectified Linear Units. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.
https://openreview.net/forum?id=B1J_rgWRW

[ADIP21] A. Apicella, F. Donnarumma, F. Isgrò, R. Prevete. A survey on modern trainable activation functions. *Neural Networks* 138:14–32, 2021.
doi:10.1016/j.neunet.2021.01.026

[AEIK18] A. Athalye, L. Engstrom, A. Ilyas, K. Kwok. Synthesizing robust adversarial examples. In *International conference on machine learning*. Pp. 284–293. 2018.

[Ayt22] C. Aytekin. Neural Networks are Decision Trees. *arXiv preprint arXiv:2210.05189*, 2022.

[Bak21] S. Bak. nnenum: Verification of relu neural networks with optimized abstraction refinement. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*. Pp. 19–36. 2021.

[BB20] R. Balestriero, R. G. Baraniuk. Mad max: Affine spline insights into deep learning. *Proceedings of the IEEE* 109(5):704–727, 2020.

[BBM$^+$15] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one* 10(7):e0130140, 2015.

[BCE$^+$23] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[BEL75] R. S. Boyer, B. Elspas, K. N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10(6):234–245, 1975.

[BH21] N. Burkart, M. F. Huber. A survey on the explainability of supervised machine learning. *Journal of Artificial Intelligence Research* 70:245–317, 2021.

[BLJ21] S. Bak, C. Liu, T. Johnson. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*, 2021.

[BS14] M. Bianchini, F. Scarselli. On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE transactions on neural networks and learning systems* 25(8):1553–1565, 2014.

[Bur16] J. Burrell. How the machine 'thinks': Understanding opacity in machine learning algorithms. *Big data & society* 3(1):2053951715622512, 2016.

[CHH$^+$18] L. Chu, X. Hu, J. Hu, L. Wang, J. Pei. Exact and consistent interpretation for piecewise linear neural networks: A closed form solution. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Pp. 1244–1253. 2018.

[DFO20] M. P. Deisenroth, A. A. Faisal, C. S. Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[FRH$^+$19] M. Fazlyab, A. Robey, H. Hassani, M. Morari, G. Pappas. Efficient and accurate estimation of lipschitz constants for deep neural networks. *Advances in Neural Information Processing Systems* 32, 2019.

[GBB11] X. Glorot, A. Bordes, Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. Pp. 315–323. 2011.

[GBC16] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GMD$^+$18] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, M. Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*. Pp. 3–18. 2018.

[GSS14] I. J. Goodfellow, J. Shlens, C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[GWZ+18] D. Gopinath, K. Wang, M. Zhang, C. S. Pasareanu, S. Khurshid. Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439*, 2018.

[GZB94] V. V. Gorokhovik, O. I. Zorko, G. Birkhoff. Piecewise affine functions and polyhedral sets. *Optimization* 31(3):209–221, 1994.

[Hin21] P. Hinz. Using activation histograms to bound the number of affine regions in ReLU feed-forward neural networks. *ArXiv* abs/2103.17174, 2021.

[HR19a] B. Hanin, D. Rolnick. Complexity of Linear Regions in Deep Networks. In Chaudhuri and Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*. Proceedings of Machine Learning Research 97, pp. 2596–2604. PMLR, 09–15 Jun 2019.
https://proceedings.mlr.press/v97/hanin19a.html

[HR19b] B. Hanin, D. Rolnick. Deep relu networks have surprisingly few activation patterns. *Advances in neural information processing systems* 32, 2019.

[JKRL09] K. Jarrett, K. Kavukcuoglu, M. Ranzato, Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th international conference on computer vision*. Pp. 2146–2153. 2009.

[KBD+17] G. Katz, C. Barrett, D. L. Dill, K. Julian, M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International conference on computer aided verification*. Pp. 97–117. 2017.

[KHI+19] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, C. W. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Dillig and Tasiran (eds.), *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Lecture Notes in Computer Science 11561, pp. 443–452. Springer, 2019.
doi:10.1007/978-3-030-25540-4_26

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM* 19(7):385–394, 1976.

[LAL+21] 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* 4(3-4):244–404, 2021.

[LeN19] A. LeNail. NN-SVG: Publication-Ready Neural Network Architecture Schematics. *J. Open Source Softw.* 4(33):747, 2019.

[LJ20] G.-H. Lee, T. S. Jaakkola. Oblique Decision Trees from Derivatives of ReLU Networks. In *International Conference on Learning Representations*. 2020.
https://openreview.net/forum?id=Bke8UR4FPB

[LL17] S. M. Lundberg, S.-I. Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30, 2017.

[MBL⁺19] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, K.-R. Müller. Layer-wise relevance propagation: an overview. *Explainable AI: interpreting, explaining and visualizing deep learning*, pp. 193–209, 2019.

[MBN⁺23] A. Murtovi, A. Bainczyk, G. Nolte, M. Schlüter, B. Steffen. Forest GUMP: a tool for verification and explanation. *International Journal on Software Tools for Technology Transfer*, 2023.
doi:10.1007/s10009-023-00702-5

[MPCB14] G. F. Montufar, R. Pascanu, K. Cho, Y. Bengio. On the number of linear regions of deep neural networks. *Advances in neural information processing systems* 27, 2014.

[NH10] V. Nair, G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. Pp. 807–814. 2010.

[NKA20] T. D. Nguyen, K. E. Kasmarik, H. A. Abbass. An exact transformation from deep neural networks to multi-class multivariate decision trees. *arXiv preprint arXiv:2003.04675*, 2020.

[NSMS23] G. Nolte, M. Schlüter, A. Murtovi, B. Steffen. The Power of Typed Affine Decision Structures: A Case Study. *International Journal on Software Tools for Technology Transfer*, 2023.
doi:10.1007/s10009-023-00701-6

[PGM⁺19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Pp. 8024–8035. Curran Associates, Inc., 2019.
http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-libra
pdf

[PMB13] R. Pascanu, G. Montufar, Y. Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

[RPK⁺17] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, J. Sohl-Dickstein. On the expressive power of deep neural networks. In *international conference on machine learning*. Pp. 2847–2854. 2017.

[RSG16] M. T. Ribeiro, S. Singh, C. Guestrin. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. Pp. 1135–1144. 2016.

[RT18] D. Rolnick, M. Tegmark. The power of deeper networks for expressing natural functions. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.
https://openreview.net/forum?id=SyProzZAW

[Rud19] C. Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence* 1(5):206–215, 2019.

[SKS+20] A. Sudjianto, W. Knauth, R. Singh, Z. Yang, A. Zhang. Unwrapping The Black Box of Deep ReLU Networks: Interpretability, Diagnostics, and Simplification. *ArXiv* abs/2011.04041, 2020.

[SNMS23] M. Schlüter, G. Nolte, A. Murtovi, B. Steffen. Towards rigorous understanding of neural networks via semantics-preserving transformations. *International Journal on Software Tools for Technology Transfer*, 2023.
doi:10.1007/s10009-023-00700-7

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton et al. Mastering the game of go without human knowledge. *nature* 550(7676):354–359, 2017.

[STR18] T. Serra, C. Tjandraatmadja, S. Ramalingam. Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*. Pp. 4558–4566. 2018.

[SWR+18] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, D. Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Pp. 109–119. 2018.

[SZS+14] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, R. Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014*. 2014.

[TKT18] B. B. Traore, B. Kamsu-Foguem, F. Tangara. Deep convolution neural network for image recognition. *Ecological informatics* 48:257–268, 2018.

[TMM+19] H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, T. T. Johnson. Star-based reachability analysis of deep neural networks. In *International symposium on formal methods*. Pp. 670–686. 2019.

[TN87] W. C. Thibault, B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. Pp. 153–162. 1987.

[UNP+21] M. Usman, Y. Noller, C. S. Păsăreanu, Y. Sun, D. Gopinath. NEUROSPF: A tool for the Symbolic Analysis of Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Pp. 25–28. 2021.

[Val05] A. Valmari. The state explosion problem. *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pp. 429–528, 2005.

[VBC+19] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782):350–354, 2019.

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin. Attention is all you need. *Advances in neural information processing systems* 30, 2017.

[WZX+21] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, J. Z. Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems* 34:29909–29921, 2021.

[ZW20] X. Zhang, D. Wu. Empirical Studies on the Properties of Linear Regions in Deep Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. https://openreview.net/forum?id=SkeFl1HKwr

[ZWX+22] H. Zhang, S. Wang, K. Xu, Y. Wang, S. Jana, C.-J. Hsieh, Z. Kolter. A Branch and Bound Framework for Stronger Adversarial Attacks of ReLU Networks. In *Proceedings of the 39th International Conference on Machine Learning*. Volume 162, pp. 26591–26604. 2022.