Towards Formal Explainability: Faithful Distillation of Deep Neural Networks into Interpretable Surrogate Models

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Maximilian Schlüter

Dortmund

2025

Tag der mündlichen Prüfung:
04. April 2025

Dekan:
Prof. Dr. Jens Teubner

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Nils Jansen

Acknowledgements

First and foremost, I would like to thank *Bernhard Steffen* for supervising my dissertation and supporting me throughout all the years I worked at his chair. Early on in my academic path he invited me to join his chair and to work with him and his working group on interesting topics in formal methods, such as the RERS challenge or the formal explanation of deep neural networks, for which I am very grateful.

I would also like to express my gratitude to *Nils Jansen* for his role as my co-supervisor.

I want to thank *Gerrit Nolte* for the wonderful time together at the chair. We wrote several papers together based on our joint research, which make up the first attached papers of this thesis.

I would like to thank *Alnis Murtovi* for the meaningful cooperation especially in the last year. Further, his emotional support in the last few month, as we both finished up our theses, helped me through this challenging time.

I am grateful to all proofreaders of this thesis. Especially to *David Schmidt* for his thoughtful and detailed feedback that was very helpful to me.

The time at the research chair was special to me, and the support, encouragement, and solidarity of my colleagues means very much to me. From chats in the coffee kitchen to the (A)ISoLA conferences, we formed many memories that I will happily look back to. Thanks to *Alexander Bainczyk*, *Steve Boßelmann*, *Daniel Busch*, *Johannes Düsing*, *Markus Frohme*, *Frederik Gossen*, *Marc Jasper*, *Anemone Kampkötter*, *Michael Lybecait*, *Oliver Rüthing*, *Steven Smyth*, *Jonas Schürmann*, and *Tim Tegeler* for this time.

Finally, I would like to thank my parents, *Sylvia* and *Ralf*, for their continuous support throughout my life. You believed in me when others did not.

Abstract

The goal of this thesis is to make the semantic function of trained deep neural networks accessible in a compact and efficient data structure based on formal principles.

Deep neural networks are today the predominant machine learning model based on their remarkable performance over the last two decades. From the first breakthroughs in computer vision with AlexNet to today's sophisticated large language models for natural language processing, deep neural networks have become the state-of-the-art in machine learning. One key factor for this success is their ability to autonomously learn from data without human guidance, enabling end-to-end optimization. On the other hand, the lack of human involvement makes the internal structure of neural networks hard to understand, as it is missing structure and design. Besides their large number of learnable parameters, three key factors are identified that make these learned intermediate representations so difficult to understand: they are *distributed*, *non-linear*, and *sub-symbolic*.

This thesis proposes a new post-hoc approach for explaining deep neural networks based on faithful surrogate models. Through a systematic and property-preserving decomposition of piecewise linear neural networks into their linear regions, the internal structure of neural networks is compiled into a new surrogate model. In this way, the typical representation of DNNs based on their dataflow, which is optimized for execution speed on graphics cards, is converted into a representation focusing on controlflow, which is more suitable for formal analysis. Consequently, the new representation is free of the distributed and non-linear internal representations. The surrogate model provides explanatory information from which different types of explanations can be derived, such as outcome explanations, class characterizations, and model explanations.

At the core of this approach stands an optimized data structure, a binary decision tree, that combines ideas from Algebraic Decision Trees, Binary Space Partitioning Trees, and classic program optimization. By placing function composition at the center, these trees enable a modular approach to faithful distillation that is easily extensible and simplifies reasoning. Through optimizations, such as infeasible path elimination, redundancies in the tree are identified and pruned.

As the distilled tree mirrors the network's behavior, it can be used to analyze its semantic properties, such as fairness or robustness. As a result of their formal grounding, these trees integrate well with mathematical notions. Furthermore, based on two-dimensional slices, it is possible to visualize the actual decision boundaries of a neural network, setting an ideal ground for exploring its behavior using intuition.

Attached Publications

The following list provides an overview over the attached publications of this dissertation. Throughout this work, they are cited in a distinct and recognizable way (e.g. [AP I: [Sch+23]]) in order to distinguish them from other literature. They are a collaborative effort of all authors and have been published in journals or conferences. Notes about my contribution have been added.

I Maximilian Schlüter, Gerrit Nolte, Alnis Murtovi, and Bernhard Steffen. „Towards rigorous understanding of neural networks via semantics-preserving transformations“. In: *International Journal on Software Tools for Technology Transfer* (May 2023). ISSN: 1433-2787. DOI: 10.1007/s10009-023-00700-7: Below cited as [AP I: [Sch+23]]
The presented ideas were discussed among all authors. Gerrit Nolte and I share equal authorship for all sections.

II Gerrit Nolte, Maximilian Schlüter, Alnis Murtovi, and Bernhard Steffen. „The Power of Typed Affine Decision Structures: A Case Study“. In: *International Journal on Software Tools for Technology Transfer* (Apr. 2023). ISSN: 1433-2787. DOI: 10.1007/s10009-023-00701-6: Below cited as [AP II: [Nol+23]]
The presented ideas were discussed among all authors. Gerrit Nolte and I share equal authorship for all sections.

III Alnis Murtovi, Alexander Bainczyk, Gerrit Nolte, Maximilian Schlüter, and Bernhard Steffen. „Forest GUMP: a tool for verification and explanation“. In: *International Journal on Software Tools for Technology Transfer* (May 2023). ISSN: 1433-2787. DOI: 10.1007/s10009-023-00702-5: Below cited as [AP III: [Mur+23]]
Alnis Murtovi was the main author. I co-authored section 4 (infeasible path elimination).

IV Maximilian Schlüter and Gerrit Nolte. „Introduction to Symbolic Execution of Neural Networks-Towards Faithful and Explainable Surrogate Models“. In: *Electronic Communications of the EASST* 82 (Oct. 2023). DOI: 10.14279/tuj.eceasst.82.1225: Below cited as [AP IV: [SN23]]
The presented ideas were discussed among all authors. I was the main author of all sections except section 6 and 7, which I co-authored. I carried out implementation and experiments.

V Maximilian Schlüter and Bernhard Steffen. „Affinitree: A Compositional Framework for Formal Analysis and Explanation of Deep Neural Networks“. In: *Tests and Proofs*. Ed. by Marieke Huisman and Falk Howar. Cham: Springer Nature Switzerland, Sept. 2024, pp. 148–167. ISBN: 978-3-031-72044-4. DOI: 10.1007/978-3-031-72044-4_8: Below cited as [AP V: [SS24]]
I was the main author of this paper and carried out implementation and experiments.

Other Peer Reviewed Publications

During my time as a Ph.D. student, I also worked on the following publications. Although they are not directly related to the main topic of this thesis, they still influenced this work indirectly.

- Barnaby Crook, Maximilian Schlüter, and Timo Speith. „Revisiting the Performance-Explainability Trade-Off in Explainable Artificial Intelligence (XAI)“. in: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. Sept. 2023, pp. 316–324. DOI: 10.1109/REW57809.2023.00060:

- Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. „Computing Inflated Explanations for Boosted Trees: A Compilation-Based Approach“. In: *The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*. Ed. by Mike Hinchey and Bernhard Steffen. Cham: Springer Nature Switzerland, Oct. 2024, pp. 183–201. ISBN: 978-3-031-73887-6. DOI: 10.1007/978-3-031-73887-6_14:

- Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. „Voting-Based Shortcuts through Random Forests for Obtaining Explainable Models“. In: *Real Time and Such: Essays Dedicated to Wang Yi to Celebrate His Scientific Career*. Ed. by Susanne Graf, Paul Pettersson, and Bernhard Steffen. Cham: Springer Nature Switzerland, Oct. 2024, pp. 135–153. ISBN: 978-3-031-73751-0. DOI: 10.1007/978-3-031-73751-0_11:

- Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. „An Efficient Compilation-based Approach to Explaining Random Forests Through Decision Trees“. In: *Proceedings of the 17th International Conference on Agents and Artificial Intelligence, ICAART 2025*. Vol. 2. SCITEPRESS, 2025, pp. 484–495. DOI: 10.5220/0013188600003890

Contents

CHAPTER 1

Introduction

1.1 Motivation

1.1.1 Introduction to Artificial Intelligence

The concept of artificial intelligence (AI) captured the interest of humans for a long time, even before computers existed. Ideas of creating artificial creatures from inorganic materials, such as bronze or clay, can already be found in ancient mythical and fictional stories, such as Galatea, Talos, and Pandora in Greek mythology [GBC16; Buc05]. The famous computer science pioneers *Ada Lovelace* and *Alan Turing* thought about mechanical [Lov42] or electrical machines [Tur50] that resemble human intelligence long before their practical implementation was feasible. One of Turing's lasting contributions to this field was the Turing Test, which asks whether an AI can be distinguished from a human by a third person in a conversation. Only with the recent emergence of large language models (LLMs) has this test been sufficiently satisfied [Hul+23; NSM23; Mei+24]. AI also became a recurring motif in pop culture, including books such as Asimov's "I, Robot" [Asi50] or Marc-Uwe Kling's "QualityLand 2.0" [Kli20], movies like "The Terminator" [FWC85], and computer games like "Detroit, become human" [Dre18]. Many of these explore deep philosophical questions regarding the human nature and identity, as well as the *opportunities* and *risks* associated with AI.

Defining AI. While the term AI is today ubiquitous, it is associated with many different concepts and ideas, such as automation or robotics. Definitions of AI vary even under experts working in that area [SVS24]. This linguistic conflation is in part based on the long history of AI and the many different approaches and application areas that were explored. For example, an early milestone in the history of AI was when IBM's Deep Blue defeated Garry Kasparov in a chess match in 1997. To evaluate game positions, Deep Blue relied on hardcoded rules that express human knowledge of the game [CHH02]. In contrast, modern AI-based chess engines no longer count on human game knowledge, but instead use *machine learning* algorithms to learn their own evaluation function, such as AlphaZero [Sil+18]. Similarly, the system

ELIZA [Wei76] from 1967 that (superficially) emulated conversational skills through *pattern matching* and *substitution* rules, differs significantly from its modern AI counterparts like ChatGPT, whose conversational skills are a result of training on a very large text corpus [Bro+20; Ouy+22]. These examples show how the field of AI has changed over the years, and consequently how broad the meaning of the term AI has become. In the Oxford English Dictionary the term AI is defined as:

> The capacity of computers or other machines to exhibit or simulate intelligent behavior; the field of study concerned with this. In later use also: software used to perform tasks or produce output previously thought to require human intelligence, esp. by using machine learning to extrapolate from large collections of data. ([Oxf])

This definition already reflects the shift over time to machine learning approaches. In the following, we will focus on this modern branch of AI.

Deep Learning. Contemporary AI is almost exclusively based on machine learning (ML), a field where *general purpose* stochastic models are adapted through a training process to solve a given task based solely on *data*. More specifically, a subfield of ML called deep learning (DL) is the current state-of-the-art based on its impressive performance on various historically challenging tasks [LBH15], such as computer vision [Far+12; KSH12; He+16], speech recognition [Hin+12], and natural language processing [Sut14; Vas+17; Bro+20]. DL systems are characterized by their neural architectures (deep neural networks (DNNs)), which originate from biological nervous systems, starting with Rosenblatt's perceptron model in 1958 [Ros58]. In contrast to other ML models, training DNNs scales well with the number of training samples and learnable parameters of the model, and can be efficiently computed on modern hardware [LBH15; GBC16]. Additionally, they offer *end-to-end* optimization, meaning that the training algorithm can—given enough training samples—tweak all aspects of the processing pipeline. Empirically, this form of optimization has been observed to be more successful when given enough training data and computational resources than AI systems build arround human knowledge [Sut19]. For example, in an end-to-end optimized self-driving car, the neural network would be directly connected to input devices, such as sensors, and output devices, such as the engine. In contrast, in classic systems responsibilities would be split into different subsystems, like speed control or environment detection. Through end-to-end optimization, DNNs can learn their own intermediate representations that are specifically optimized for the task at hand, without requiring any guidance by humans.

Not only does the lack of human involvement improve performance,[1] it also enables the application of DL in areas where human knowledge is either limited or hard to encode in a computer-readable format, like image recognition and natural language processing. At the same time, this autonomy also makes the systems harder to understand, since it is not a trivial task to match learned representations of DNNs to human concepts. Three key properties have been identified in the

[1] Under certain assumptions about data availability, quality, and representativeness.

literature that make these intermediate representations so complex: they are *non-linear*, *distributed* [Hin86], and *sub-symbolic*. Consequently, DNNs are characterized as being *opaque* and are considered *black-box models* [Bur16; Zed21; FT21].

1.1.2 Ubiquity of Deep Learning.

The development of DL in the last two decades has made significant advances, pushing the limits of what was imagined as science fiction into being reality. These innovations were not only celebrated in academia, but also in the industry and general public.[2] The following short timeline of recent and popular AI technologies shows how AI increasingly affects our lives. In 2006, *Google Translate* launches, a widely known translation service that is now based on a neural architecture (transfomer). In 2011, Apple introduced the virtual assistant *Siri* for iPhones, which was later followed by Amazon's *Alexa*. In 2014, *Tesla Autopilot* was introduced, a level 2 automation of self-driving cars. In 2015, *AlphaGo* became the first computer program to beat a professional human player, Fan Hui, at Go. Subsequent improvements of this technology resulted in *AlphaZero* (2017), which achieves superhuman performance in Go, chess, and shogi just through the act of playing against itself. In 2017, *DeepL Translator* launches, another widely known translation service, which is based on convolutional neural networks. In 2021, *GitHub Copilot* was released, which supports software engineers by providing code completion based on comments or partially written code. In 2022, generative image models like *DALL-E*, *Midjourney*, and *Stable Diffusion* were publicly launched, enabling users to generate images of any motive with styles from different epochs and artists through a natural language description thereof. In 2022, the first chatbot *ChatGPT* was introduced, the first system capable of handling large-scale natural language conversations with non-experts on a wide range of topics.

In addition to these well-known technologies, there are also many applications of AI inside other products. For example, recommender systems are used for a long time in online shopping to predict items that might also interest the customer: "users who liked x also like y". Similar recommender systems can be found in music or film streaming services for recommendations or as part of an auto-play feature. On many social media platforms, the content that is shown to users on their *feed* is curated by algorithms to increase user engagement.

While many of these technologies are proprietary, it is reported that most of them are (now) based on DL. From these examples one can see the ubiquity of DL technology.

1.1.3 Safety of Deep Learning.

Artificial intelligence (AI)—and in particular DL—already impacts our lives and will continue to do so [Flo+18]. Even people who do not use AI directly are still affected by it through its use in commercial products and public services. As with any new technology, AI introduces opportunities and risks at the time. The ethics of technology, i.e., the question of whether technology has an inherent normative value (good

[2] With research teams in companies like Google, Meta, and OpenAI, the line between academia and industry is blurring.

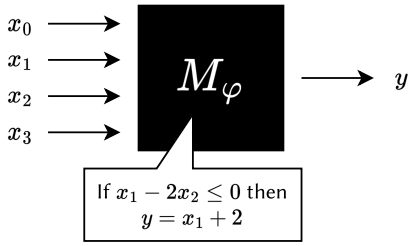


Figure 1.1: Illustration of an opaque neural network as a black box where only inputs $x_0, \ldots, x_3$ and outputs y can be observed. The explanation provides insights into the internal process of the black box. Illustration based on [Arr+20].

or bad) or is neutral, has received great attention after the Manhattan Project and recently again with the advances in AI. A pragmatic stance to that question called *technological instrumentalism* states that the normative value of technology depends on the context it is used in and therefore calls for a (consequentialist) evaluation on a case-by-case basis. For example, on the one hand, the introduction of self-driving cars involves many risks and ethical questions regarding their decision making (see, e.g., [NS16] for a comparison with the trolley problem). On the other hand, car accidents are one of the leading causes of death among young people, and are often caused by avoidable reasons, such as speeding, driving under influence, and distracted driving [Par+22]. Here, even imperfect self-driving cars might improve the status quo, and not using them could result in overall more severe accidents (*opportunity cost*).

That being said, the rapidly growing usage of AI in societal areas increases the potential harm it can cause. Indeed, there are already cases where deployed AI systems produced unfavorable results, including systemic discrimination [Meh+21] and deadly accidents [ZS19]. Perhaps one of the most famous cases of machine bias is the COMPAS system for judging recidivism risk of prisoners. An article in ProPublica revealed that this actively-used system exhibits clear bias against the black demographic [Ang+16]. In 2018, the first recorded pedestrian fatality involving a self-driving car was reported [KC19]. Although the system involved in the accident could not be analyzed as it is proprietary, the study [KC19] showed that state-of-the-art object recognition systems such as YOLO were able to identify the involved human from the original camera images. In [Meh+21], it was shown through metamorphic testing that a popular software for self-driving cars at the time was faulty. This brings the question to mind if the accident was avoidable.

Therefore, when using AI in societal or *safety-critical* use cases, we should apply the same regulations and diligence as we do in similar cases: in engineering products like cars, critical infrastructure like nuclear reactors, and software systems like airplane autopilots regulations are in place to ensure safety.

1.1.4 Explainable Artificial Intelligence

To address issues like these, the field of *explainable AI (XAI)* has emerged, which can be summarized by the metaphor "opening the black box" [Gui+18], where black-box refers to opaque ML systems (Figure 1.1). Its techniques can be classified into different groups. Here the taxonomy of [Spe22] is used: *ante-hoc* techniques strive

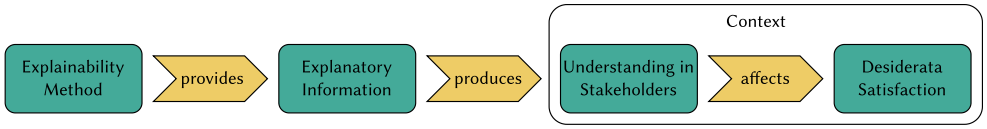


Figure 1.2: Overview of the intended function of an explanation in XAI. Simplified version of [Lan+21].

to make ML models themselves more interpretable (see [Rud19]), while the more prominent *post-hoc* methods extract explanatory information from opaque models after training. Some methods are *model-agnostic*, meaning that they can be applied to a wide range of ML models such as Support Vector Machines (SVMs), Random Forests (RFs), or DNNs, while others are *model-specific*. Finally, some explanations are *local* to specific predictions of a model, while others are *global* taking the entire model into account. From there, many fine-grained distinctions can be made, such as *explanation by feature relevance*, *visual explanation*, *explanation by simplification*, and others.

Defining Explanations. The central objects of XAI research are *explanations* for AI models. For defining explanations, multiple models exist in *philosophy of science* that are typically used in this context. One of the most common models there is called *deductive-nomological explanation*, which links empirical observations with law-like statements (rules) to arrive at a description of the phenomenon with the help of deduction. Other models are more concerned with the causal processes involved or with reasons. An essential characteristic of all explanations is that they give answer to the question "Why?". Overall, the goal of explanations is to foster *understanding* in an individual, which depends on many factors, including the individual's prior knowledge and mental state. Here, psychological effects must also be taken into account, for example, providing a user with more information about how a system works might actually decrease his trust in the system than if he has no information at all. Therefore, the recipients of explanations are typically grouped into homogeneous parties, called *stakeholders* in XAI. Another important part of an explanation is to decide what aspect should be understood by the explanation. In XAI research these are typically referred to as *desiderata*.

Stakeholder and Desiderata. Concretely for the XAI setting, explanations can be seen as the process depicted in Figure 1.2 by [Lan+21]. An XAI technique produces explanatory information, which when considered by some stakeholder produce some insight into the system. In the context of XAI, involved stakeholders are usually *users*, *developers*, and *regulators*. There are multiple goals (desiderata) for the complete process, i.e., what specific aspect of the system should be explained. The following list is an excerpt from the desiderata identified in [Lan+21]:

- Accountability
- Debugability
- Fairness
- Informed Consent
- Legal Compliance

- Morality/Ethics
- Performance
- Privacy
- Responsibility
- Robustness
- Trustworthiness

Depending on the desideratum, the stakeholder, and the context, the requirements on the explanation change. For example, for a non-critical application of AI tasked to predict the next song as part of an auto-play feature, a simple explanation like "The song was recommended to you, because you often listen to songs from this genre" might be completely sufficient. On the other hand, when AI is used to detect and identify road signs as part of a self-driving vehicle, the explanation "The sign was identified as a stop sign, because it looks similar to other stop signs" is probably insufficient. From this explanation it is not clear if the same stop sign would be detected under bad weather conditions, as it lacks information to generalize the systems behavior. Further, for a user of the car a pictogram or a zoomed in image of the road sign might be sufficient for fostering trust, while a regulator needs a more sophisticated explanation to judge legal compliance. The need for detail also increases when an AI system fails at its task.

Both stakeholder understanding and desiderata satisfaction are beyond the scope of this thesis, but should be kept in mind for the big picture. What is relevant from all of the discussed points are the so-called *epistemic needs* that they imply. For example, in order to check the satisfaction of the desideratum fairness, one needs certain facts or information about the system regarding its behavior with respect to, e.g., different genders. We have seen that there are stakeholders, contexts, and desiderata that require a lot of accurate and detailed information about an AI system for reasonable judgments, and these are exactly the cases for which the presented method of this thesis is designed for. In the following we will refer to them simply as *high-stakes scenarios*.

Recently, it was shown that post-hoc methods often fall short in capturing the intricate structure of DNNs, leading users to form false beliefs about the system [BKS17; Rud19; MI22; INM19; LB20; Dim+20]. E.g., in [Ade+18] it was shown that certain proposed saliency maps are indifferent to parameter changes of the DNN. To overcome these problems, *correct* and *complete* explanations are required, which are known to be NP complete [Kat+17]. This is where this thesis takes place.

1.2 Research Problems Addressed in the Thesis

1.2.1 Conceptual and Technical Overview

This thesis presents a post-hoc, model-specific XAI technique based on surrogate models that is well suited for high-stakes scenarios. To create the surrogate model, the original opaque DNN is compiled—often refered to in this context as *distilled*—into another representation that is both more comprehensible to humans and more accessible to algorithms for analysis. In the respective XAI literature, only three models are generally considered being comprehensible: decision trees, rule-based

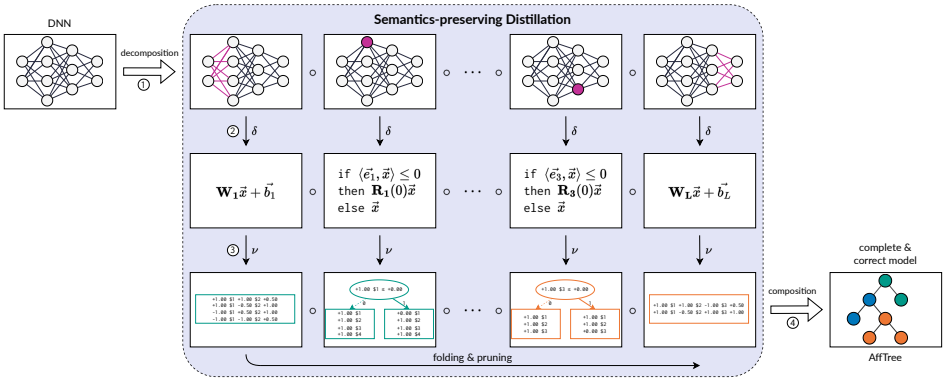


Figure 1.3: Faithful distillation process. The PWL neural network is decomposed into layers, which are then mapped to functionally-equivalent `AffTrees` (only the hidden layers contain activation functions). The resulting sequence of `AffTrees` is aggregated into a single equivalent `AffTree` through step-wise composition with optimizations.

models, and linear systems [Gui+18; Lip18; Mol25]. Out of these, decision trees are an optimal model for representing the semantics of DNNs when they are piecewise linear (PWL) functions. The tree structure is also computationally efficient.

A key difference between the original opaque DNN and the distilled tree is that the former is a dataflow representation optimized for inference speed on graphics cards, while the latter is a controlflow representation that is suitable for comprehension and analysis. The process is akin to symbolic execution, and can indeed be defined in that way. Furthermore, the internal representations (latent space) of the hidden layers of the DNN are expanded in this process, thus eliminating two sources of opacity in the distilled tree: *non-linearity* and *distributed representations*.

The distillation approach introduced here is modular and flexible based on algebraic properties. This enables it to be applicable for different desiderata and different kinds of explanations, such as outcome explanations (local), or class characterizations and model explanations (global), where local explanations are less resource-intensive.

Technically, a DNN $h\colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ is first represented as a sequence of L PWL layers:

$$h : \vec{x} \mapsto \big(h_L \circ \sigma_{L-1} \circ \cdots \circ \sigma_1 \circ h_1\big)(\vec{x})$$

where $h_\ell(\vec{x})$ are affine functions, and σ_ℓ are non-affine activation functions ($1 \leq \ell \leq L$). Activation functions σ_ℓ are further decomposed into a sequence of partial activation functions, each acting only on one neuron:

$$\sigma_\ell = \sigma_\ell^{(n_\ell)} \circ \cdots \circ \sigma_\ell^{(1)}$$

This is possible for most activation functions, as they are typically an element-wise application of a basis function, like $\max\{x, 0\}$ in the case of ReLU. Given this decomposition, each of these 'layers' is mapped to an `AffTree` instance with the same semantics (process $\nu \circ \delta$ in Figure 1.3). Then, this sequence of `AffTrees` is aggregated step-by-step into a single `AffTree` by repeatedly applying function composition (Figure 1.3). In each composition step, optimization routines, such as infeasible path elimination, are applied. In this way, redundancies are pruned early on in the

process, thus avoiding compound redundancies down-the-road. For example, when a node in an `AffTree` is infeasible, subsequent compositions would only append additional nodes under it. Nevertheless, the whole subtree would still be infeasible.

Every step of the pipeline is *semantics-preserving*. By decomposing the network into small components (here the layers), the mapping of these components to `AffTrees` is simplified. For example, the partial ReLU activation function (which acts on only one neuron) can be mapped to an `AffTree` with three nodes. The 'heavy lifting' is performed by the implementation of the composition operator and the optimizations. Therefore, it is straightforward to extend the process, e.g., by adding new activation functions. Instead of specifying `AffTrees` directly, it is also possible to generate them from an imperative language called `Afflang`.

The distilled `AffTree` is a faithful yet interpretable surrogate model (FISM) of the network, providing explanatory information. Particularly interesting is a visualization mode where the tree is restricted to two-dimensional *slices* of the input space. These can be plotted for visual inspection of the model and its decision boundaries.

An accompanying tool, called `Affinitree`, implements the concepts introduced in this thesis. It is written in the system language Rust for cache and memory-efficient layouting of `AffTrees`. To appeal to a wider audience, a thin Python wrapper is provided that forwards the underlying Rust API. Both versions are available under the laguage-specific package indicies, crates.io and PyPI. While the main body of this thesis is focussed on theoretical aspects of the distillation process, the attached publication [AP V: [SS24]] covers implementation details.

While this thesis presents the case for explaining PWL DNNs through faithful distillation, the discussed approach is, at its core, universal and therefore applicable to all PWL functions. This approach can be best applied to domains that also have an algebraic structure, specifically compositionality, to make sure that the modularity and flexibility of this approach can be fully utilized. Interesting other ML domains may be PWL ensemble models, such as Random Forests or Boosted Trees [GS21; MSS24a].

1.2.2 Scientific Contribution

This thesis combines and extends the state-of-the-art in formal XAI concerning the extraction of accurate and relevant information from DNNs. An improved distillation method is presented that focuses on the algebraic properties of DNNs. This approach guarantees formal correctness and provides a compositional framework for analyzing and interpreting DNNs. The main contributions of this approach are as follows:

Efficient Representation of the Semantic Function. After the ReLU function became the most prominent activation function in DL, it was demonstrated in [PMB13; Mon+14] that ReLU DNNs are PWL. The PWL nature of DNNs allows to decompose trained DNNs into their linear regions [Chu+18; Ayt22], which simplifies analysis tasks as linear functions are one of the simplest functions that exist. For example, in a ReLU neural network when all ReLU's are fixed to one of their two linear regions (called *active* for $(0, \infty)$ and *inactive* for $(-\infty, 0]$), the network acts linearly. These are called ReLU configurations

or more generally activation patterns. However, this decomposition is computationally expensive, as the number of activation patterns that have to be checked grows exponentially with the number of learnable parameters in DNNs [PMB13].

Recently, it was shown that the number of linear regions that DNNs actually attain in practice is much lower than what is theoretically possible [HR19b; HR19a]. Depending on the learned weights of the network or its architecture, some activation patterns cannot be realized as they are contradictory. In [HR19b], a statistical argument for the distribution of the weights and biases was given as to why so many activation patterns are infeasible in trained DNNs. Furthermore, from a geometric and combinatorial perspective, the input dimension of the network also limits the number of attainable regions [Zas81]. The geometry of the latent space is influenced by previous layers of the network. For example, when a network contains a bottleneck, like in auto encoders, then the number of attainable linear regions is reduced [Aro+18].

This thesis improves the decomposition of DNNs by identifying and pruning infeasible activation patterns *early* and *efficiently*, resulting in reduced time and space complexity of the distillation process as well as reducing the size of the final surrogate model. Experiments will show that applying these optimizations results in surrogates that are orders of magnitude smaller than in a distillation without them.

Furthermore, the representation is improved by applying ideas from algebraic decision diagrams (ADDs) and binary space partitioning (BSP) Trees allowing an efficient storing of PWL functions. This representation only keeps decisions that are necessary to maintain faithfulness and otherwise removes redundancies. As a result of this thesis, a DNN with r linear regions can be distilled into a surrogate model with a size bounded by $\mathcal{O}(r)$.

Modular and Holistic Framework. The landscape of existing explainability methods is large and diverse. Popular tools like LIME [RSG16], SHAP [LL17], saliency maps [SVZ13], and counterfactual explanations [WMR17] are typically fast and understandable, but only capture local relationships of DNNs and are often not faithful. In contrast, tools like neural network verifiers (e.g., [Bak21; Tra+20]) provide provable guarantees, but are computationally expensive and limited to certain properties, such as checking ϵ-robustness. Unfortunately, their methodologies cannot be combined in a way that joins their strengths.

In contrast, the distillation approach in this thesis leverages the algebraic properties of DNNs to provide a novel, *modular* approach to distillation. At the core of this method stands a binary decision tree structure called `AffTree` that adopts useful aspects of ADDs, such as their compositionality and algebraic properties, while avoiding properties that are not applicable to this setting, like their fixed variable order or their directed acyclic graph (DAG) structure. The result is a best-of-both-worlds approach: The decision structure maintains its efficiency while structured operations can still be computed efficiently. In this

manner, the surrogate model natively supports operations, such as function composition, addition, multiplication, and equality. This has two main benefits

1. It simplifies reasoning over the operations by providing levels of abstraction. For example, when t_m corresponds to the behavior of the DNN for males and t_f for females, then equality between both genders can be expressed as $t_m = t_f$. While the efficient computation of the resulting tree may involve many details (which are laid out in this thesis), the concept of equality is intuitively clear.
2. It provides great flexibility and enables recombination of analysis results. Thus, users can mix existing building blocks in a way that matches their needs. For example, one could compute the average over multiple different random initalizations. Through the modular approach it is also easy to implement missing functionality, like custom activation functions.

Configurable Levels of Detail. Adding to the previous point, existing methods are also not flexible. When the level of detail of an explanation of LIME is insufficient, it is not possible to spend more computation time to get a better explanation in return. Neural network verifiers are more flexible in this regard, but are limited in the information that can be concluded from their analysis (for most, linear properties).

Through the modular approach the distillation is more flexible in both regards. For one, intermediate results can be reused in all supported analyses. And distillation can be computed up to a certain time budget, and when it is found that this budget was insufficient to conclusively answer relevant questions, it can be increased reusing the former results. When exhaustive analysis is unnecessary, the method supports constructing semantic representations at varying levels of granularity, balancing precision or completeness with computational efficiency.

Visualization Tool. The accompanying tool of the thesis, which applies the theoretical concepts in practice, allows for visualizing the behavior of DNNs, in particular on two-dimensional slices of the input space formally supported by a concolic execution. These visualizations facilitate intuitive interpretation of network behavior, supporting human-understandable insights. Furthermore, they can educate users on the true nature of DNNs by studying illustrations of their semantic function.

By combining these contributions, the thesis advances explainability through surrogate models for deep neural networks, particularly in safety-critical application areas where formal guarantees and human interpretability are desirable.

1.3 Outline

This thesis is organized into an introductory passage, a main passage consisting itself of two parts, and a closing passage. In the introduction (Chapter 1), a brief summary over the history, significance, and central terms of AI and DL is given.

The chapter is followed by preliminaries in Chapter 2 concerning linear algebra, piecewise linear functions, deep learning, and decision diagrams.

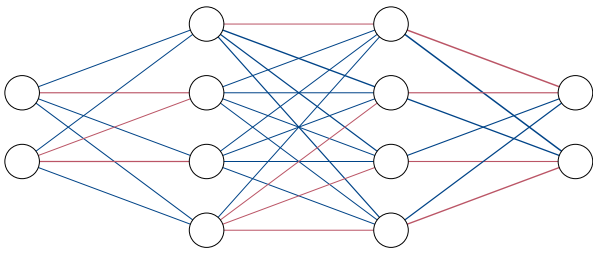
The first part of the main section is titled "Theory" and is concerned with the mathematics of the faithful distillation approach that is presented in this thesis. A short introduction to this part is given in Chapter 3. To prepare the distillation approach, an imperative language called `Afflang` is introduced in Chapter 4 that is well-suited for the representation and analysis of PWL functions. The chapter also reflects on the formal properties of this language. Building on this language, Chapter 5 presents formal encodings of many DL architectures into `Afflang`, including feed forward neural networks (FNNs) and convolutional neural networks (CNNs). A short outlook is given how recurrent neural networks (RNNs) might fit into this framework, before the chapter concludes with a summary of what Open Neural Network Exchange (ONNX) operators match the discussed components. From their, Chapter 6 states how symbolic and concolic execution can be applied to `Afflang` programs, producing faithful control flow graphs (CFGs). When put together with Chapters 5 and 6, CFGs can be generated for PWL DNNs. Through rewriting, these graphs are converted into an optimized representation in form of binary decision trees, called `AffTrees`. These are FISMs that exactly represent the semantic function of the DNN and thus enable their formal and rigorous analysis.

The second part is focused on "Practice". In Chapter 7, semantics preserving optimizations are presented for improving running time and space complexity of the distillation. Applications of `AffTrees` for explaining DNNs are demonstrated in Chapter 8. In this chapter, two use cases concerning fairness analysis and adversarial examples are presented.

Finally, the thesis concludes with related work (Chapter 9) and a summary and future work (Chapter 10).

Reading Hints

FNNs are visualized in this thesis as graphs of neurons. Edges between these neurons correspond to their weights and are colored blue when the weight is positive and red when it is negative. The thickness of an edge is proportional to the square root of its weight. A neural network $\mathbb{R}^2 \to \mathbb{R}^2$ with two hidden layers, each consisting of four neurons, can be visualized as:



This thesis includes two types of complementary material, which can be treated as parenthesis. They provide useful foundational information (*preliminaries*) or give specific details (*excursions*) that are useful, but can also divert from the main story.

Therefore, they are clearly marked with a pictogram, a title, and an end symbol. They look as follows:

Preliminaries (Title). Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Excursion (Title). Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

CHAPTER 2

Preliminaries

This chapter provides a brief introduction to the fundamental topics of this thesis, mainly linear algebra, piecewise linear functions, polytopes, deep learning, and decision diagrams.

2.1 Linear Algebra

In this thesis, the theory of linear algebra plays a central role based on (i) its importance within deep learning itself and (ii) its algebraic properties such as compositionality. The following introduction to linear algebra states important concepts for this thesis and establishes a consistent notation based on the book [Axl97]. Basic knowledge about algebraic structures, such as fields, is assumed in the following. A comprehensive introduction to this topic can be found in the book [Jud20].

Definition 2.1.1 (Vector Space [Axl97]). A vector space $(V, +, \cdot)$ over a field $\mathbb{F}$ is an algebraic structure consisting of a non-empty set V equipped with two operations

$$\begin{aligned} + &: V \times V \to V && \text{vector addition} \\ \cdot &: \mathbb{F} \times V \to V && \text{scalar multiplication} \end{aligned}$$

such that the following properties hold:

1. $\vec{u} + \vec{v} = \vec{v} + \vec{u}$ for all $\vec{u}, \vec{v} \in V$;
2. $(\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$ for all $\vec{u}, \vec{v}, \vec{w} \in V$;
3. $(\lambda\mu)v = \lambda(\mu\vec{v})$ for all $\vec{v} \in V$ and all $\lambda, \mu \in \mathbb{F}$;
4. there exists an element $\vec{0} \in V$ such that $\vec{v} + \vec{0} = \vec{v}$ for all $\vec{v} \in V$;
5. for every $\vec{v} \in V$, there exists a $\vec{w} \in V$ such that $\vec{v} + \vec{w} = \vec{0}$;
6. $1\vec{v} = \vec{v}$ for all $\vec{v} \in V$;
7. $\lambda(\vec{u} + \vec{v}) = \lambda\vec{u} + \lambda\vec{v}$ for all $\vec{u}, \vec{v} \in V$ and all $\lambda \in \mathbb{F}$;
8. $(\lambda + \mu)\vec{v} = \lambda\vec{v} + \mu\vec{v}$ for all $\vec{v} \in V$ and all $\lambda, \mu \in \mathbb{F}$;

Definition 2.1.2 (Span [Axl97]). The set of all linear combinations of a list of vectors $\vec{v_1}, \ldots, \vec{v_n}$ of V is called the *span* of $\vec{v_1}, \ldots, \vec{v_n}$, denoted as

$$\operatorname{span}\{\vec{v_1}, \ldots, \vec{v_n}\} := \{\, \lambda_1\vec{v_1} + \cdots + \lambda_n\vec{v_n} \mid \lambda_1, \ldots, \lambda_n \in \mathbb{F} \,\} \subseteq V$$

Definition 2.1.3 (Linear Independent [Axl97]). A list of vectors $\vec{v_1}, \ldots, \vec{v_n}$ of V is called *linearly independent* iff the equation ($\lambda_1, \ldots, \lambda_n \in \mathbb{F}$)

$$\lambda_1 \vec{v_1} + \cdots + \lambda_n \vec{v_n} = \vec{0}$$

has only the solution $\lambda_1 = \cdots = \lambda_n = 0$.

Definition 2.1.4 (Basis [Axl97]). A list of vectors $\vec{v_1}, \ldots, \vec{v_n}$ of V is called *basis* iff it is linearly independent and spans V.

Definition 2.1.5 (Dimension [Axl97]). The *dimension* of a vector space is the length of any basis $\vec{v_1}, \ldots, \vec{v_n}$ of the vector space and notated as

$$\dim V = n$$

in case it is finite.

Theorem 2.1.6. Any n-dimensional vector space V over a field $\mathbb{F}$ is isomorphic to $\mathbb{F}^n$.

A vector $(x_1, \ldots, x_n)$ of $\mathbb{F}^n$ is abbreviated as $\vec{x}$. To refer to the i-th component of a vector $\vec{v}$ we write v_i.

2.1.1 Matrices and Affine Maps

A matrix $\boldsymbol{W}$ is a collection of values from a field $\mathbb{F}$ arranged in a rectangular array with m rows and n columns:

$$\boldsymbol{W} = \begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix} .$$

The number of rows, columns, and the underlying field can be stated shortly by writing $\boldsymbol{W} \in \mathbb{F}^{m \times n}$. An element in the i-th row and j-th column of the matrix $\boldsymbol{W}$ is denoted either by $w_{i,j}$, where w is written in lowercase, or by $\left(\boldsymbol{W}\right)_{i,j}$, where the matrix is enclosed in parentheses (where $1 \leq i \leq n$ and $1 \leq j \leq m$). A matrix $\boldsymbol{W} \in \mathbb{F}^{m \times n}$ can be reflected along the main diagonal resulting in the transpose $\boldsymbol{W}^\mathsf{T}$ of shape $n \times m$ defined by the equation

$$\left(\boldsymbol{W}^\mathsf{T}\right)_{i,j} := \boldsymbol{W}_{j,i}$$

The i-th row of $\boldsymbol{W}$ can be regarded as a $1 \times n$ matrix given by

$$\boldsymbol{W}_{i,:} := (w_{i,1}, \ldots, w_{i,n}) \ .$$

Similarly, the j-th column of $\boldsymbol{W}$ can be regarded as a $m \times 1$ matrix defined as

$$\boldsymbol{W}_{:,j} := (w_{1,j}, \ldots, w_{m,j})^\mathsf{T} \ .$$

Matrix addition is defined component-wise for matrices with the same type, i.e.,

$$\left(\boldsymbol{W} + \boldsymbol{N}\right)_{i,j} := w_{i,j} + n_{i,j}$$

and scalar multiplication as

$$\left(\lambda \cdot \boldsymbol{W}\right)_{i,j} := \lambda \cdot w_{i,j} \ .$$

The multiplication of two matrices with compatible type $\boldsymbol{W} \in \mathbb{F}^{m \times r}$ and $\boldsymbol{N} \in \mathbb{F}^{r \times n}$ is defined as

$$\left(\boldsymbol{W} \cdot \boldsymbol{N}\right)_{i,j} := \sum_{k=1}^{r} w_{i,k} \cdot n_{k,j} \ .$$

Identifying

- $n \times 1$ matrices with (column) vectors
- $1 \times m$ matrices with row vectors
- 1×1 matrices with scalars

as indicated above, makes the well-known dot product of $\vec{v}, \vec{w} \in \mathbb{F}^n$

$$\langle \vec{v}, \vec{w} \rangle := \sum_{i=1}^{n} v_i \cdot w_i$$

just a special case of matrix multiplication. The same holds for matrix-vector multiplication that is defined for a $m \times n$ matrix $\boldsymbol{W}$ and a vector $\vec{x} \in \mathbb{F}^n$ as

$$\left(\boldsymbol{W} \cdot \vec{x}\right)_i := \boldsymbol{W}_{i,:} \cdot \vec{x} \ .$$

Matrices with the same number of rows and columns, i.e., with type $n \times n$ for some $n \in \mathbb{N}$, are said to be *square matrices*. Square matrices have a neutral element for matrix multiplication, called *identity matrix*, that is zero everywhere except for the entries on the main diagonal which are one:

$$\boldsymbol{I}_n := \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix} \ .$$

A square matrix $\boldsymbol{W} \in \mathbb{F}^{n \times n}$ has a multiplicative inverse $\boldsymbol{W}^{-1}$ iff

$$\boldsymbol{W} \cdot \boldsymbol{W}^{-1} = \boldsymbol{I}_n = \boldsymbol{W}^{-1} \cdot \boldsymbol{W}$$

In that case, $\boldsymbol{W}$ is called *invertible* or *regular*. Regular matrices form a group with matrix multiplication. A square matrix $\boldsymbol{Q}$ is called *orthogonal* iff its transpose is also its inverse, i.e., iff $\boldsymbol{Q}\boldsymbol{Q}^\intercal = \boldsymbol{I}_n = \boldsymbol{Q}^\intercal\boldsymbol{Q}$. A matrix $\boldsymbol{W} \in \mathbb{F}^{m \times n}$ is in *Toeplitz form* when constants $c_{1-m}, \ldots, c_{n-1} \in \mathbb{F}$ exists such that $w_{i,j} = c_{j-i}$, i.e., when the rows are shifted versions of each other:

$$\begin{bmatrix} c_0 & c_1 & \ldots & c_{n-1} \\ c_{-1} & c_0 & \ldots & c_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ c_{1-m} & c_{2-m} & \ldots & c_{n-m} \end{bmatrix} \ .$$

Matrices have a tight link to linear and affine maps, which are the isomorphisms of linear and affine spaces.

Definition 2.1.7 (Linear Map [Axl97]). A function $f : V \to W$ is called *linear* iff for all $\vec{u}, \vec{v} \in V$ and $\lambda \in \mathbb{F}$ the following properties hold:

$$\begin{aligned} f(\vec{u} + \vec{v}) &= f(\vec{u}) + f(\vec{v}) && \text{additivity} \\ f(\lambda\vec{v}) &= \lambda f(\vec{v}) && \text{homogeneity} \end{aligned}$$

Linear maps with finite-dimensional domain and codomain are isomorphic to matrices.

Lemma 2.1.8. For every linear map $f : \mathbb{F}^n \to \mathbb{F}^m$ exists a unique matrix $\boldsymbol{M} \in \mathbb{F}^{m \times n}$ such that:

$$f(\vec{v}) = \boldsymbol{M}\vec{v}$$

for all $\vec{v} \in \mathbb{F}^n$. Conversely, for every matrix $\boldsymbol{M} \in \mathbb{F}^{m \times n}$ the function f defined by $f(\vec{v}) = \boldsymbol{M}\vec{v}$ is linear.

Many concepts defined for (linear) functions have an equivalent concept in terms of matrices. Let $f(\vec{v}) = \boldsymbol{M}\vec{v}$ be a linear function, then

(i) f is bijective iff $\boldsymbol{M}$ is invertible,
(ii) f is surjective iff the columns of $\boldsymbol{M}$ span the entire codomain (has full row rank).
(iii) f is injective iff the columns of $\boldsymbol{M}$ are linearly independent (has full column rank).

Definition 2.1.9 (Affine Map). A function $f : V \to W$ is called *affine* iff for all $\vec{u}, \vec{v} \in V$ and $\lambda \in \mathbb{F}$ the following property holds:

$$f(\lambda\vec{u} + (1 - \lambda)\vec{v}) = \lambda f(\vec{u}) + (1 - \lambda) f(\vec{v})$$

Lemma 2.1.10. Any affine map $\alpha : \mathbb{F}^n \to \mathbb{F}^m$ can be represented as

$$\alpha(\vec{v}) = \boldsymbol{W}\vec{v} + \vec{b}$$

for a fixed matrix $\boldsymbol{W} \in \mathbb{F}^{m \times n}$ and vector $b \in \mathbb{F}^m$.

Definition 2.1.11 (Operations over Affine Functions). For affine functions we define the operations *addition*, *function composition*, and *concatenation* with the following symbols and types:

$$\begin{aligned} + &: (\mathbb{F}^n \to \mathbb{F}^m) \times (\mathbb{F}^n \to \mathbb{F}^m) \to \mathbb{F}^n \to \mathbb{F}^m \\ \circ &: (\mathbb{F}^m \to \mathbb{F}^r) \times (\mathbb{F}^n \to \mathbb{F}^m) \to \mathbb{F}^n \to \mathbb{F}^r \\ \oplus &: (\mathbb{F}^n \to \mathbb{F}^m) \times (\mathbb{F}^n \to \mathbb{F}^r) \to \mathbb{F}^n \to \mathbb{F}^{m+r} \end{aligned}$$

Note that the result is always another affine function, although the type might change. They are defined as:

$$\begin{aligned} (\vec{x} \mapsto \boldsymbol{W_1}\vec{x} + \vec{b_1}) + (\vec{x} \mapsto \boldsymbol{W_2}\vec{x} + \vec{b_2}) &= \vec{x} \mapsto (\boldsymbol{W_1} + \boldsymbol{W_2})\vec{x} + (\vec{b_1} + \vec{b_2}) \\ (\vec{x} \mapsto \boldsymbol{W_1}\vec{x} + \vec{b_1}) \circ (\vec{x} \mapsto \boldsymbol{W_2}\vec{x} + \vec{b_2}) &= \vec{x} \mapsto (\boldsymbol{W_1}\boldsymbol{W_2})\vec{x} + (\boldsymbol{W_1}\vec{b_2} + \vec{b_1}) \\ (\vec{x} \mapsto \boldsymbol{W_1}\vec{x} + \vec{b_1}) \oplus (\vec{x} \mapsto \boldsymbol{W_2}\vec{x} + \vec{b_2}) &= \vec{x} \mapsto \begin{bmatrix} \boldsymbol{W_1} \\ \boldsymbol{W_2} \end{bmatrix} \vec{x} + \begin{bmatrix} \vec{b_1} \\ \vec{b_2} \end{bmatrix} \end{aligned}$$

When we restrict the arguments to certain affine functions (e.g., linear functions, constant functions), we get the following operations 'for free' ($s \in \mathbb{F}, \vec{v} \in \mathbb{F}^n, \boldsymbol{M} \in \mathbb{F}^{r \times m}$):

$$\begin{aligned} s \cdot (\vec{x} \mapsto \boldsymbol{W}\vec{x} + \vec{b}) &= \vec{x} \mapsto (s \cdot \boldsymbol{W})\vec{x} + (s \cdot \vec{b}) \\ \boldsymbol{M} \cdot (\vec{x} \mapsto \boldsymbol{W}\vec{x} + \vec{b}) &= \vec{x} \mapsto (\boldsymbol{M} \cdot \boldsymbol{W})\vec{x} + (\boldsymbol{M} \cdot \vec{b}) \\ \vec{v} + (\vec{x} \mapsto \boldsymbol{W}\vec{x} + \vec{b}) &= \vec{x} \mapsto \boldsymbol{W}\vec{x} + (\vec{v} + \vec{b}) \end{aligned}$$

Some operations are closed when restricting to compatible types:

Lemma 2.1.12 (Affine Functions as Vector Space). For any $n, m \in \mathbb{N}$ the set of all affine functions with type $\mathbb{F}^n \to \mathbb{F}^m$ is a vector space with addition + and scalar multiplication $\cdot$.

Lemma 2.1.13 (Typed Monoid [AP I: [Sch+23]]). The set of all affine functions with composition $\circ$ is a monoid when operands are restricted to compatible types.

$$\begin{gathered} \circ : (\mathbb{F}^r \to \mathbb{F}^m) \times (\mathbb{F}^n \to \mathbb{F}^r) \to (\mathbb{F}^n \to \mathbb{F}^m) \\ (f \circ g)(\vec{v}) \coloneqq f(g(\vec{v})) \end{gathered}$$

Interim: Real Numbers and Constructivism

While vector spaces and metric spaces are generally defined over any field, normed vector spaces and inner product spaces are mostly defined over the reals. Indeed, the real numbers $\mathbb{R}$ are probably the standard number model in most publications.

At the same time, the real numbers are somewhat artificial: they are uncountable and non-computable. For any practical application, the reals can only be seen as a mathematical ideal that is approximated, while other systems like the rationals can be fully implemented.

Constructive mathematics offer an alternative perspective by focussing on computability and explicit constructions. Where possible, I prefer constructive arguments over non-constructive ones. However, this is not the primary focus of this work, and I aim to avoid unnecessary complications.

Thus, I will follow convention and use the reals explicitly in the following definitions. However, I wish to introduce one concept that bridges the gap between the reals and computability:

Definition 2.1.14 (Real Closed Field). A field $\mathbb{F}$ is called *real closed* if it has the following property: Any first order sentence over fields is true in $\mathbb{F}$ iff it is true over the reals.

Prominent fields with this property are the algebraic numbers (roots of polynomials with rational coefficients) and the computable numbers (those definable by Turing Machines), both of which are countable.

2.1.2 Linear Equations and Matrix Factorizations

A *linear equation system* is a set of equations where each row corresponds to a linear combination of *unknowns* x_i with *coefficients* $a_{ij} \in \mathbb{R}$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

when the constants b_i are all equal to zero, the system is called *homogeneous*, otherwise *inhomogeneous*. In matrix notation, this system can be compactly written as:

$$\boldsymbol{A}\vec{x} = \vec{b}$$

where $\boldsymbol{A}$ is the $m \times n$ coefficient matrix, $\vec{x}$ is the vector of unknowns, and $\vec{b}$ is the vector of constants.

Matrices play a central role in solving linear equation systems. A system represented as $\boldsymbol{A}\vec{x} = \vec{b}$ can be transformed to simplify or directly solve it using matrix operations, such as Gaussian elimination or matrix factorization. If $\boldsymbol{A}$ is square and invertible, the system has a unique solution given by $\vec{x} = \boldsymbol{A}^{-1}\vec{b}$. If $\boldsymbol{A}$ is not square or not invertible, solutions (if they exist) involve generalized techniques, such as the pseudoinverse.

Matrix factorizations decompose a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ into products of matrices with certain properties, such as being an upper triangular matrix or orthogonal.

1. **LU decomposition** $\boldsymbol{A} = \boldsymbol{LU}$, where $\boldsymbol{L}$ is lower triangular and $\boldsymbol{U}$ is upper triangular. Solve $\boldsymbol{L}\vec{y} = \vec{b}$ (forward substitution), then $\boldsymbol{U}\vec{x} = \vec{y}$ (back substitution).
2. **QR decomposition** $\boldsymbol{A} = \boldsymbol{QR}$, where $\boldsymbol{Q}$ is orthogonal and $\boldsymbol{R}$ is upper triangular. Multiply both sides by $\boldsymbol{Q}^\mathsf{T}$ to simplify: $\boldsymbol{R}\vec{x} = \boldsymbol{Q}^\mathsf{T}\vec{b}$.
3. **Singular value decomposition** $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^\mathsf{T}$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal, and Σ is diagonal. Useful for underdetermined or overdetermined systems, providing least-squares solutions. This gives rise to the *Moore–Penrose pseudoinverse* defined as $\boldsymbol{A}^+ = \boldsymbol{U}^\mathsf{T}\Sigma^+\boldsymbol{V}$.

2.1.3 Norms and Distances

Vector norms generalize the idea of distances, and thus of neighborhoods, to vector spaces. These are formalized in mathematics using metric spaces and normed spaces [Mag22].

Definition 2.1.15 (Inner Product Space [Axl97]). An *inner product space* is a vector space V with a mapping $\langle \cdot, \cdot \rangle \colon V \times V \to \mathbb{R}$, called *inner product*, that satisfies the following properties:

positivity $\langle v, v \rangle \geq 0$ for all $v \in V$;

definiteness $\langle v, v \rangle = 0$ iff $v = 0$;

additivity $\langle u + v, v\rangle = \langle u, w\rangle + \langle v, w\rangle$ for all $u, v, w \in V$;

homogeneity $\langle \lambda u, v\rangle = \lambda\langle u, v\rangle$ for all $\lambda \in \mathbb{R}$ and all $u, v \in V$;

symmetry $\langle u, v\rangle = \langle v, u\rangle$ for all $u, v \in V$.

Every inner product induces a norm, called its *canonical norm*, defined as:

$$\|v\| := \sqrt{\langle v, v\rangle} \ .$$

Therefore, every inner product space is also a normed space. In a similar manner, every norm implies a metric, called *norm induced metric*:

$$d(v, w) := \|w - v\| \ .$$

And, finally, a metric induces a topology where the open balls defined as

$$B_r(v) := \{\, w \in V \mid d(v, w) < r \,\}$$

form a *base*.

A special type of normed spaces defined by so-called l-norms is of special interest:

Definition 2.1.16 (l_m-norms). For $m \in \mathbb{N}$, the l_m-norm is the function $\|\cdot\|_m \colon \mathbb{R}^n \to \mathbb{R}$ defined by:

$$\|\vec{x}\|_m := \sqrt[m]{\sum_{i=1}^{n} |x_i|^m} \ .$$

Important l_m-norms are the l_1 norm

$$\|\vec{x}\|_1 := \sum_{i=1}^{n} |x_i|$$

and the l_2 norm or Euclidean norm[1]

$$\|\vec{x}\|_2 := \sqrt{\sum_{i=1}^{n} x_i^2} \ .$$

Another important norm is the so-called l_∞ norm. While not technically an l-norm according to the definition above, it arises naturally as the limit of the l-norms as m approaches infinity and is defined by:

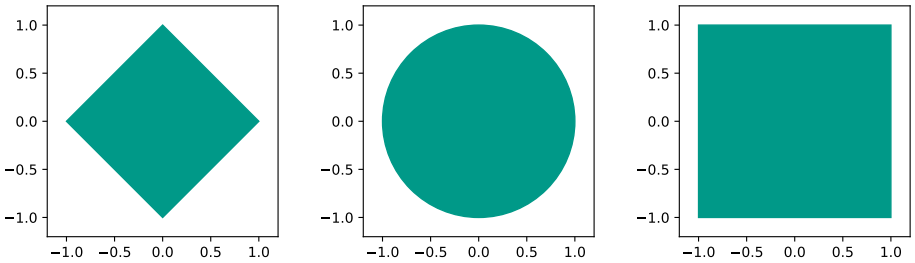
$$\|\vec{x}\|_\infty := \max_{i \in \{1,\dots,n\}} |x_i|$$

With these definitions we can now formalize the neighborhood of a point.

Definition 2.1.17 (Unit Ball). For a given l_m-norm $\|\cdot\|_m$, we define the corresponding m-unit ball as

$$B_m := \{\, \vec{x} \in \mathbb{R}^n \mid \|\vec{x}\|_m < 1 \,\} \ . \tag{2.1}$$

[1]For real vector spaces $\mathbb{R}^n$, the Euclidean norm is the canonical norm as $\|\vec{x}\|_2 = \sqrt{\langle \vec{x}, \vec{x}\rangle}$.



(a) $m = 1$ **(b)** $m = 2$ **(c)** $m = \infty$

Figure 2.1: Selection of two-dimensional m-balls for different l_m-norms.

The unit ball is a closed, convex subset of $\mathbb{R}^n$ centered at the origin. It generalizes the notion of a disk with radius 1 to both higher dimensions ($n > 2$) and non-euclidean spaces ($m \neq 2$). The relevant unit balls for this thesis are illustrated in Figure 2.1. Every generic m-ball of $\mathbb{R}^n$ can be constructed using translation

$$\vec{y} + B_m = \{ \vec{x} \in \mathbb{R}^n \mid \|\vec{x} - \vec{y}\|_m < 1 \}$$

and scaling

$$rB_m = \{ \vec{x} \in \mathbb{R}^n \mid \|\vec{x}\|_m < r \}$$

of unit m-balls. In the latter case r is called the radius. If it is clear from the context, we omit the dimensionality of the m-ball.

2.2 Convex Polytopes

This section is based on the books [Bro83; GZB94; HRZ17].

Definition 2.2.1 (Halfspace). Let $\vec{w} \in \mathbb{R}^n$ with $\vec{w} \neq 0$ and $b \in \mathbb{R}$. Then the set

$$p = \{ \vec{x} \in \mathbb{R}^n \mid \langle \vec{w}, \vec{x} \rangle + b = 0 \}$$

is called a *hyperplane* of $\mathbb{R}^n$. A hyperplane partitions $\mathbb{R}^n$ into two convex subspaces, called *halfspaces*. The positive and negative halfspaces of p are defined as

$$p^+ := \{ \vec{x} \in \mathbb{R}^n \mid \langle \vec{w}, \vec{x} \rangle + b > 0 \}$$
$$p^- := \{ \vec{x} \in \mathbb{R}^n \mid \langle \vec{w}, \vec{x} \rangle + b < 0 \}$$

Definition 2.2.2 (Polytope). A polytope $Q \subseteq \mathbb{R}^n$ is the intersection of k halfspaces for some natural number $k \in \mathbb{N}$.

$$Q = \bigcap_{i=1}^{k} \{ \vec{x} \in \mathbb{R}^n \mid \langle \vec{w}_i, \vec{x} \rangle + b_i \leq 0 \}$$

Polytopes are the models of affine predicates.

Definition 2.2.3 (Affine Predicate). An *affine predicate* is a logic formula of the form $\boldsymbol{A}\mathfrak{u} \leq \vec{b}$ for a fixed matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$ that is satisfied iff the relation $\leq$ holds for all rows. The variable $\mathfrak{u}$ is symbolic (written in Fraktur), its value from domain $\mathbb{R}^n$ is provided by the caller.

For convenience, we introduce the short hands ($z \in \mathbb{R}, \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{V} \in \mathbb{R}^{o \times n}, \vec{b} \in \mathbb{R}^m, \vec{c} \in \mathbb{R}^o$):

$$\begin{aligned} u_i \leq z &\equiv \vec{e_i}^\top \mathfrak{u} \leq z \\ u_i \leq u_j &\equiv (\vec{e_i} - \vec{e_j})^\top \mathfrak{u} \leq 0 \\ \boldsymbol{A}\mathfrak{u} \geq \vec{b} &\equiv (-\boldsymbol{A})\mathfrak{u} \leq (-\vec{b}) \\ \boldsymbol{A}\mathfrak{u} \leq \vec{b} \wedge \boldsymbol{V}\mathfrak{u} \leq \vec{c} &\equiv \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{V} \end{bmatrix} \mathfrak{u} \leq \begin{bmatrix} \vec{b} \\ \vec{c} \end{bmatrix} \end{aligned}$$

When used in a logic with negation, we also allow the short hands $\boldsymbol{A}\mathfrak{u} < \vec{b} \equiv \neg(\boldsymbol{A}\mathfrak{u} \geq \vec{b})$ and $\boldsymbol{A}\mathfrak{u} > \vec{b} \equiv \neg(\boldsymbol{A}\mathfrak{u} \leq \vec{b})$. Following common conventions in logic, we use the following notation to say that a formula is satisfied by a model ($\vec{x} \in \mathbb{R}^n, M \subseteq \mathbb{R}^n, \boldsymbol{A} \in \mathbb{R}^{m \times n}, \vec{b} \in \mathbb{R}^m$):

$$\begin{aligned} \vec{x} \models \boldsymbol{A}\mathfrak{u} \leq \vec{b} &\iff \boldsymbol{A}\vec{x} \leq \vec{b} \\ M \models \boldsymbol{A}\mathfrak{u} \leq \vec{b} &\iff \forall \vec{x} \in M : \vec{x} \models \boldsymbol{A}\mathfrak{u} \leq \vec{b} \\ \text{Models}(\boldsymbol{A}\mathfrak{u} \leq \vec{b}) &:= \{\, \vec{x} \in \mathbb{R}^n \mid \vec{x} \models \boldsymbol{A}\mathfrak{u} \leq \vec{b} \,\} \end{aligned}$$

The set of all models $\text{Models}(\boldsymbol{A}\mathfrak{u} \leq \vec{b})$ is always a convex polytope. Each row $A_{i,:}\mathfrak{u} = b_i$ of the predicate defines a hyperplane in $\mathbb{R}^n$. The boundary $\text{bd}\,\text{Models}(\boldsymbol{A}\mathfrak{u} \leq \vec{b})$ consists exclusively of points from these hyperplanes. However, not all row hyperplanes are part of the boundary. When a hyperplane is part of the boundary, we call it *supporting* (see [Bro83] for a formal definition). Removing non-supporting rows from an affine predicate does not alter its semantics.

2.3 Piecewise Linear Functions

This thesis is concerned with functions that are piecewise linear (PWL), i.e., with functions that are linear in some local regions. This class of functions is used in many applications, most notably in the context of function approximation. Their success in this area is based on two properties. First, PWL functions are universal function approximators, i.e., they can approximate any measurable function arbitrary well. Second, they inherit the simplicity of linear functions almost everywhere. The following definition is based on [Ovc02].

Definition 2.3.1 (PWL). A function $f : \Gamma \to \mathbb{R}^m$ over a convex domain $\Gamma \subseteq \mathbb{R}^d$ is said to be *continuous piecewise linear* (CPWL) iff there exists a finite family of closed regions $\mathscr{Q} = \{Q_1, \dots Q_q\}$ with $\Gamma = \bigcup \mathscr{Q}$ such that f coincides with a linear function g_i on each region Q_j. We then say that g_i is active in the region Q_j. The unique functions $\{g_1, \dots, g_k\}$ are called the component functions of f.

By this definition, PWL functions are always continuous, as the regions are closed and therefore the function values have to agree on the shared boundaries. When the regions are open, the definition captures the class of all PWL functions (then the regions must be closed explicitly for the following definitions to be correct). Without loss of generality, we assume henceforth that the family $\mathscr{Q}$ partitions the domain Γ of f in the following sense [GZB94]:

Definition 2.3.2 (Polyhedral Partition). A *polyhedral partition* over a set $\Gamma \subseteq \mathbb{R}^d$ is a finite family $\mathscr{Q}$ of sets such that for all $Q_i, Q_j \in \mathscr{Q}$ the following hold:

1. Each $Q_i \in \mathscr{Q}$ is a convex (bounded or unbounded) polytope.
2. The interiors of the sets are pairwise disjoint, i.e.,

$$\operatorname{int} Q_i \cap \operatorname{int} Q_j = \emptyset \iff i \neq j$$

3. The set Γ is covered, i.e., $\Gamma = \bigcup \mathscr{Q}$.

In the above definition of PWL functions, the regions $\mathscr{Q} = \{Q_1, \dots Q_q\}$ are not uniquely determined by f. The following unique regions are defined in the literature [TM99; Xu+16]:

Projection Regions For each *unique* component function g_i, project the area where g_i coincides with f onto the domain Γ. This area may be non-convex and non-connected. Calling the resulting region Ω_i, the collection of all such sets $\{\Omega_1, \dots \Omega_k\}$ are the projection regions of f.

Base Regions A base region $D_i \subseteq \Gamma$ of f is the maximal set where no other component function is both greater than and less than the single active component function g_j on D_i.

Unique-order Regions A unique-order region $O_i \subseteq \Gamma$ is the inclusion-maximal set where all component functions can be linearly order, i.e., $g_{i_1}(x) \leq g_{i_2}(x) \leq \dots \leq g_{i_k}(x)$ for all $x \in O_i$.

It was shown in a number of publications [BKS95; Ovc00; TM99; Wil63] that every PWL function can be written as a polynomial in a lattice (linear functions with maximum and minimum) using just its component functions. The following lemma was stated and proven in [Ovc02].

Theorem 2.3.3 (Lattice Representation of PWL). For any continuous piecewise linear function $f\colon \Gamma \to \mathbb{R}$ defined on a convex domain $\Gamma \subseteq \mathbb{R}^d$ with unique component functions $\{g_1, \dots, g_k\}$, there exists a family of incomparable (with respect to $\subseteq$) index sets $\{S_j\}_{j \in J}$ such that

$$f(\vec{x}) = \max_{j \in J} \min_{i \in S_j} g_i(\vec{x}) \tag{2.2}$$

holds for all $\vec{x} \in \Gamma$.

The family of index sets $\{S_j\}_{j \in J}$ can be derived from the following regions (details are given in the respective publications):

1. Convex projection regions $\{\Omega_1, \dots, \Omega_k\}$ [TM99]
2. Base regions $\{D_1, \dots, D_{k_3}\}$ [Xu+16]
3. Unique-order regions $\{O_1, \dots, O_{k_4}\}$ [TM99]
4. Domain Γ [Ovc00]

2.4 Deep Learning

Deep learning (DL) encompasses many proficient neural architectures for many different application areas. Details about their history and training can be found in [GBC16]. Throughout this thesis, the explanation of relevant DL components is integrated in the text where it is needed, using the preliminary blocks. Here, only a few aspects are discussed that are too detailed for such an integration.

2.4.1 Representation of Inputs and Outputs.

The input and output types of DNNs vary from application domains, including tabular data, images, sound, and text. At some point, however, all data is converted into a sequence of floating-point values, which can be represented as a vector. Typically, these are organized as the famous *tensors* from deep learning.[2] For example, images are commonly seen as tensors of shape (N, C, H, W) where N is the batch length, C the number of channels, H the height, and W the width. In the following calculus, we will consider input data as flattened vectors. When the input to the network was a higher dimensional tensor, we flatten it in row-major order. This does not restrict the expressiveness, as one can seamlessly convert between tensors and flattend vectors. Concretely, for the given example of image tensors of shape (N, C, H, W) the conversion can be done as follows:

$$a[i,j,k,l] = b[iCHW + jHW + kW + l]$$
$$b[m] = a\left[\frac{m}{CHW}, \frac{m \mod CHW}{HW}, \frac{m \mod HW}{W}, m \mod W\right]$$

where a is a multi-dimensional array (tensor) and b is a single-dimensional array view of the data.

To make the conversion explicit, we define the *vectorize* operation that flattens any multi-dimensional array (matrices, tensors) into a vector, assuming a row-major ordering. Given a tensor x of order n with indices $i_n, \ldots, i_1$ where i_1 is the index that varies the fastest and i_n the one that varies the slowest, and let d_k be the length of the k-th index, then the vectorize or flattening operation can be defined as:

$$\big(\mathrm{vec}(x)\big)_j := x^{i_n,\ldots,i_1}$$

where

$$i_k = \left(\left\lfloor \frac{j-1}{\prod_{l=1}^{k-1} d_l} \right\rfloor \mod d_k\right) + 1$$

Here we assume that the index j is in bounds, i.e., $1 \le j \le \prod_{l=1}^{n} d_l$. The row-major order is sometimes also called lexicographic access pattern, as the injective function $g\colon \mathbb{N} \to \mathbb{N}^n$ given by $g(j) := (i_n, \ldots, i_1)$, where i_k is defined as above, gives rise to the lexicographic order $\le_{\mathrm{lex}} \subseteq \mathbb{N}^n \times \mathbb{N}^n$ when defining:

$$g(j_1) \le_{\mathrm{lex}} g(j_2) \iff j_1 \le j_2$$

[2] For example, giving name to the library TensorFlow or being the central data structure in PyTorch. Note that they are different from tensors used in other branches of mathematics and physics.

2.4.2 Convolutional Neural Networks

Convolutional neural network (CNN) are a very important architecture in DL based on their historical and ongoing success in computer vision. CNNs were among the first neural architectures that surpassed other ML models on computer vision tasks such as image classification and object recognition. Over time the difference became very pronounced, to the point where CNNs became the default model for computer vision tasks.

CNNs make use of *parameter sharing* resulting in an efficient use of the network's resources: to express similarly complex semantics, a CNN requires considerably fewer parameters than an FNN. Important for utilizing these few parameters for improved training speed were advances in GPU programming (GPGPU). CNNs introduce domain knowledge into the architecture, e.g., by placing importance on spatial relationships and local features in the input image. It can be proven that under certain conditions CNNs are translation equivariant, i.e.,

$$f(\tau \circ x) = \tau \circ f(x)$$

for translations τ.

Mathematical Background. Mathematically, convolution is a commutative, linear operation applied to two functions (signals) f and g which computes the overlap between their area:

$$(f \star g)(t) \coloneqq \int_{-\infty}^{\infty} f(t-x)g(x)\,\mathrm{d}x$$

This corresponds to a sliding window shifted by t where the area of the function f is weighted by the function g. As such, convolutions can extract spatial (or time, depending on the interpretation of t) patterns from f by designing g in a specific way. For example, in statistics convolution can implement a moving average or in audio processing it is used for computing reverberation.

Convolutions have a long history in ML for feature extraction both within deep learning and other ML methods. In these cases, the discrete version is more relevant, which can be stated as

$$(f \star g)(t) = \sum_{x=-M}^{M} f(t-x)g(x)$$

where g has finite support in $\{-M, \ldots, M\}$. Since in ML both f and g are finite, it is custom to shift the indices such that g falls into the range $\{0, \ldots, 2M\}$. We call g a *kernel* or *filter* applied to f. Given their discrete form, f and g can be represented as finite vectors $f \in \mathbb{R}^m$ and $g \in \mathbb{R}^n$ (with $n \ll m$). To see that the operation is indeed linear one can define the following Toeplitz matrix $\boldsymbol{C} \in \mathbb{R}^{(m-n+1)\times m}$:

$$c_{i,j} = \begin{cases} g(j-i+1) & \text{if } 1 \leq j-i+1 \leq n \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

Then convolution can be expressed as $f \star g = \boldsymbol{C}f$. This route is often taken when analyzing CNNs, e.g., [Geh+18; BB20]. However, forming this Toplitz matrix is memory-intensive, especially when the input $f \in \mathbb{R}^m$ is large relative to the filter $g \in \mathbb{R}^n$.

Images. To fully track the spatial relations in images, one needs to consider both axes of the 2D input images. Thus, we update the convolution formula[3] for finite, two-dimensional array representing images as follows: Let $\boldsymbol{I}$ denote a 2D input matrix of shape $H \times W$ (height × width), and let $\boldsymbol{K}$ be a kernel of shape $F \times F$, then the convolution slides $\boldsymbol{K}$ over $\boldsymbol{I}$ following its two-dimensional grid [GBC16]:

$$\left(\boldsymbol{I} \star \boldsymbol{K}\right)_{i,j} := \sum_{\mu=1}^{F} \sum_{\nu=1}^{F} I_{i+\mu-1,j+\nu-1} K_{\mu,\nu}$$

In the formula, one can clearly see that the *same* $F \times F$ elements of the filter are used to weigh a *moving* window of $F \times F$ elements from $\boldsymbol{I}$ by means of a linear combination. The elements from $\boldsymbol{I}$ that are relevant for computing a single output value (pixel) (i, j) are called the receptive field. One can also clearly see that when we move one pixel in the output by incrementing i or j (assuming they are within bounds), the corresponding receptive field also moves by one pixel, and thus for $F > 1$ the fields overlap. There is a parameter called *stride*, which controls by how much the receptive field is moved between convolution steps, with the default being one. In other words, striding controls the step size at which the filter moves over the input. When s stands for the stride, we can update the formula to [GBC16]:

$$\left(\boldsymbol{I} \star_s \boldsymbol{K}\right)_{i,j} := \sum_{\mu=1}^{F} \sum_{\nu=1}^{F} I_{s(i-1)+\mu,s(j-1)+\nu} K_{\mu,\nu}$$

The output image has shape $H_{out} \times W_{out}$, where the dimensions H_{out} and W_{out} are determined by:

$$H_{out} = \left\lfloor \frac{H + 2\text{padding} - F}{\text{stride}} \right\rfloor + 1 \tag{2.4}$$

$$W_{out} = \left\lfloor \frac{W + 2\text{padding} - F}{\text{stride}} \right\rfloor + 1 \tag{2.5}$$

In order to keep operations as modular as possible we will extract padding operations into an own operation. Thus, we will assume that the dimensions of H, W, F and stride match in such a way that the above divisions have no remainder (when necessary through a preceding application of padding).

Example. In this capacity, kernels can be used to compute Gaussian blur or to detect edges in an image. For example, consider the following kernel known as *Sobel Operator*:

$$\boldsymbol{G_x} := \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

Convolving an image with $\boldsymbol{G_x}$ as kernel can detect horizontal edges in the image [KVB88].

[3] To be precise, machine learning uses a rotated kernel with respect to the mathematical definition (as can be seen by the switched signs), which is called cross-correlation in mathematics. We follow the convention in machine learning and also call it convolution.

Channels. Although kernels can detect interesting spatial relations over the whole image with relatively few parameters, the expressiveness of a single kernel is limited. It is therefore beneficial to compute many kernels in parallel over the same input. In CNNs, this concept of parallel convolutions is organized in *channels*. To express channels from a computer science view it is beneficial to define higher-dimensional arrays so that the new channels can be axes of the array. Mathematically speaking, this concept is related to *tensors*, although there are differences. Tensors generalize matrices to higher dimensions and as such have a strong connection to multilinear forms and also obey basis change rules.

Let the input now be a tensor of shape $C_{in} \times H \times W$ where C_{in} is the number of input channels. The kernel tensor then contains one independent filter of size $F \times F$ for each input channel C_{in} leading to an array of shape $C_{in} \times F \times F$ of parameters. Additionally, the kernel tensor also contains C_{out} many independent filters of the previous kind, leading to the final shape $C_{out} \times C_{in} \times F \times F$. Thus the kernel tensor is of order four, i.e., it can be addressed as a four-dimensional array $k^{\alpha,\beta,\mu,\nu}$ where α, β, μ, ν are indices. Where matrices can be decomposed into a collection of row or column vectors, tensors can be decomposed into *fibers* written with a colon : as index. When multiple indices are not fixed it is called a *slice*. Thus, when we write $k^{\alpha,\beta,:,:}$ we refer to the $F \times F$ kernel of the α-th output channel and the β-th input channel. Then the output tensor Z of multichannel strided convolution is [GBC16]:

$$z^{\alpha,i,j} = \left(\sum_{\beta=1}^{C_{in}} (i^{\beta,:,:} \star_s k^{\alpha,\beta,:,:}) \right)_{i,j} \tag{2.6}$$

$$= \sum_{\beta=1}^{C_{in}} \sum_{\mu=1}^{F} \sum_{\nu=1}^{F} i^{\beta,s(i-1)+\mu,s(j-1)+\nu} k^{\alpha,\beta,\mu,\nu} \tag{2.7}$$

Here the sum over all input channels is a popular aggregation pattern that allows to combine the information from the different input channels. For example, see the input channels as the color channels red, green, and blue of a colored image. First, the filters are convoluted over each channel separately, extracting useful features like edges for each color channel independently. But then, through the addition of all previously computed feature maps, it is possible in the next layer to react to the interaction of the colors with each other.

Toeplitz Matrices. For the α-th output channel and β-th input channel, $k^{\alpha,\beta,:,:}$ corresponds to an $F \times F$ filter. Each such filter can be written as a Toeplitz matrix

$$T^{\alpha,\beta} := \begin{bmatrix} k^{\alpha,\beta,1,1} & k^{\alpha,\beta,1,2} & \cdots & k^{\alpha,\beta,1,F} & 0 & \cdots & 0 & k^{\alpha,\beta,2,1} & \cdots \\ 0 & k^{\alpha,\beta,1,1} & k^{\alpha,\beta,1,2} & \cdots & k^{\alpha,\beta,1,F} & \ddots & \vdots & & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 & & \\ 0 & \cdots & 0 & k^{\alpha,\beta,1,1} & k^{\alpha,\beta,1,2} & \cdots & k^{\alpha,\beta,1,F} & & \end{bmatrix}$$

which is of shape $H_{out} \cdot W_{out} \times H_{in} \cdot W_{in}$. The input is just a flattend version of the input image. For each input channel $\beta \in [C_{in}]$ a new filter exists with its own

coefficients, which are aggregated through summation into a single output image of shape $H_{out} \times W_{out}$. The summation can be naturally expressed in as a block matrix

$$\mathrm{vec}\Big(\big(x \star k\big)^{\alpha,:,:}\Big) = \begin{bmatrix} T^{\alpha,1} & T^{\alpha,2} & \cdots & T^{\alpha,C_{in}} \end{bmatrix} \cdot \mathrm{vec}(x)$$

As this process is repeated for each output channel $\alpha \in [C_{out}]$ the resulting matrix is block matrix:

$$\mathrm{vec}\big(x \star k\big) = \begin{bmatrix} T^{1,1} & T^{1,2} & \cdots & T^{1,C_{in}} \\ T^{2,1} & T^{2,2} & \cdots & T^{2,C_{in}} \\ \vdots & \vdots & \ddots & \vdots \\ T^{C_{out},1} & T^{C_{out},2} & \cdots & T^{C_{out},C_{in}} \end{bmatrix} \cdot \mathrm{vec}(x)$$

Each block $T^{\alpha,\beta}$ is a Toeplitz matrix corresponding to the convolution filter applied between input channel β and output channel α. The overall block matrix represents the entire multichannel convolution operation.

2.4.3 Recurrent Neural Networks

Recurrent neural network (RNN) are a class of neural architectures designed to handle sequential data. Most architectures are able to accept sequence of arbitrary length. This class also makes use of parameter sharing similar as CNNs. However, while in CNNs the parameters are shared per kernel for *one* application over the image, RNNs share parameters over the depth of their computation tree [GBC16]. They are characterized by the following formula

$$h^{(t)} := f(x^{(t)}, h^{(t-1)}; \theta)$$

for a sequence $x^{(t)}, \ldots, x^{(1)}$ of input data, a hidden state $h^{(t)}$, and parameters θ. All input vectors $x^{(i)}$ of the sequence have the same length and the size of the hidden state is also fixed but must be chosen as a hyperparameter.

While the size of the input sequence may vary, any specific input has a finite and fixed length n. For this sequence, the RNN can be *unfolded* such that

$$\begin{aligned} h^{(n)} &= f(x^{(n)}, h^{(n-1)}; \theta) \\ &= f(x^{(n)}, f(x^{(n-1)}, h^{(n-2)}; \theta); \theta) \\ &= f(x^{(n)}, f(x^{(n-1)}, \cdots f(x^{(1)}, \vec{0}; \theta) \cdots ; \theta); \theta) \end{aligned}$$

Here the parameter sharing can be clearly seen: each f is parameterized by the same θ. As such, the network can better generalize to sequences of different length, as no position in the sequence has any special meaning or treatment. The final hidden state $h^{(n)}$ may be processed further by subsequent parts of the network, such as fully connected layers.

The function f must aggregate new information given by the current element of the sequence $x^{(t)}$ with the old information stored in $h^{(t-1)}$. Since the size of the hidden state is fixed while the input sequence can be arbitrary long, this aggregation can be assumed to be lossy. The training process optimizes the parameters θ (ideally)

in a way such that the information aggregated in the hidden state are relevant for solving the specific task at hand.

A typical realization of an RNN hat produces for each input $x^{(t)}$ an output $\hat{y}^{(t)}$ is given by

$$h^{(t)} := \tanh\left(\boldsymbol{U}x^{(t)} + \boldsymbol{W}h^{(t-1)} + \vec{b}\right)$$
$$\hat{y}^{(t)} := \operatorname{softmax}\left(\boldsymbol{V}h^{(t)} + \vec{c}\right)$$

In this form, for example, RNNs could be used to translate an input sentence $\vec{x}$ in French into an output sentence $\vec{y}$ in English.

2.5 Decision Diagrams

Algebraic decision diagrams (ADDs) are DAGs that lift an abstract algebra (A, O) to functions $\Sigma \to A$ [GS21]. An algebra (A, O) consists of a carrier set A and a set O of operations over A. Operations $op \in O$ have a fixed arity and are closed under A. The lifting respects the operations of the algebra (definition based on [MSS24b]):

Definition 2.5.1 (Lifting). Given an r-ary operation $h \in O \subseteq A^r \to A$ of the algebra (A, O), it is lifted to ADDs such that for r ADDs $f_1, \ldots, f_r \colon \Sigma \to A$ and all inputs $x \in \Sigma$ the following holds

$$h_A\big(f_1, \ldots, f_r\big)(x) := h\big(f_1(x), \ldots, f_r(x)\big)$$

where $h_A\big(f_1, \ldots, f_r\big)$ is, in particular, an ADD itself.

Part I

Theory

CHAPTER 3

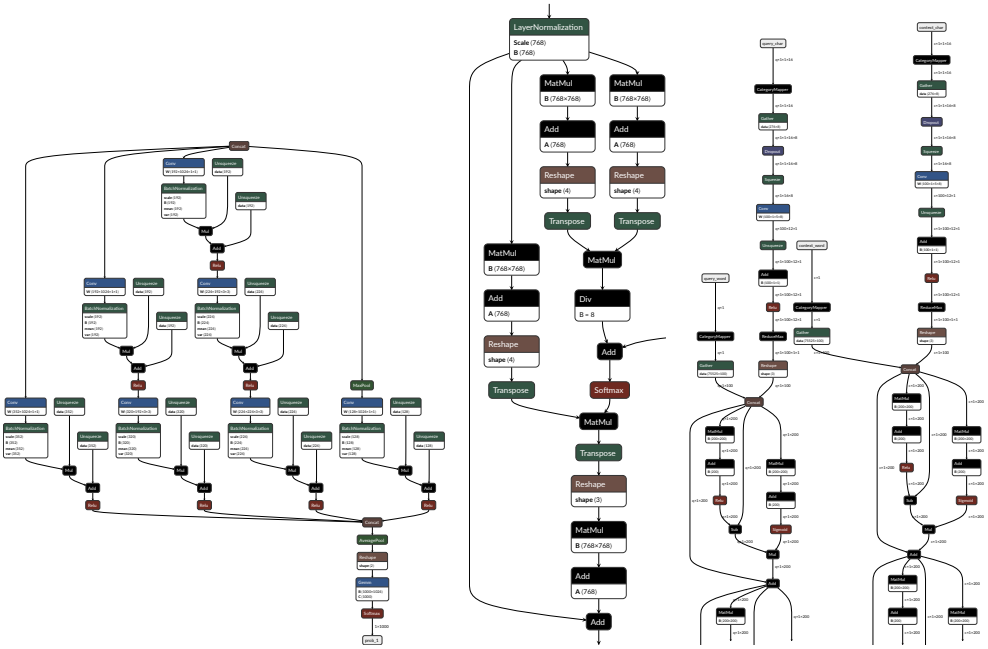
Introduction

This chapter presents the formal foundations for the distillation process of deep learning models into interpretable surrogate models through a series of semantics-preserving transformations. While neural networks show remarkable performance for many complex machine learning tasks (Section 1.1), they lack the transparency that is needed for their responsible use in safety-critical applications (Section 1.1.4). To address this issue, the chapter introduces a series of semantics-preserving transformations, each improving on specific aspects of the syntactic representation of the models. Ultimately, the goal is to transform these initial representations—which include deep learning models such as feed-forward, convolutional, and recurrent neural architectures—into representations that are more amenable to rigorous analysis.

The core idea is to reduce the complexity of neural network architectures by encoding them into `Afflang`, a minimal yet expressive language designed for piecewise linear (PWL) functions. This transformation enables the application of established techniques from program analysis and formal methods to the resulting `Afflang` programs. These techniques can then be utilized to verify a broad set of requirements, referred to as *desiderata* (Section 1.1.4). The verification of these requirements can be boiled down to the *epistemic needs* of the involved stakeholders (Section 1.1.4). As each individual case has a different epistemic profile, there is no one-size-fits-all solution. The here introduced *framework* starts at the network and presents a modular, systematic approach by which true propositions (*facts*) about the inner workings of the neural network can be derived, referred to as *explanatory information*. These propositions can then be abstracted and contextualized to bridge formal mathematical claims with human-understandable concepts, facilitating the verification of the desiderata (which is briefly explored in Chapter 8).

To achieve this, the following first part of this thesis presents a constructive approach, introducing transformations, symbolic methods, and analytical constructs in the following chapters:

(i) First, the imperative language `Afflang` is introduced, which consists of an essential core of instructions designed for expressing PWL functions. These core instructions of `Afflang` will allow us to encode all PWL deep learning architectures,



(a) Inception v2 (2015) **(b)** BERT (2018) **(c)** BIDAF (2016)

Figure 3.1: Excerpts from the computation graphs of various deep learning models. These examples show diverging and merging dataflow, as well as many different Open Neural Network Exchange (ONNX) operators that make up proficient neural networks. Specifications taken from ONNX Zoo and visualized with Netron.

as, e.g., specified in the ONNX format, into small programs of this language. Although `Afflang` is designed to be simple, it can encode deep neural networks (DNNs) with complex dataflow graphs (Figure 3.1). Based on its minimal structure, the language is well-suited for formal reasoning, such as proving theoretical properties by structural induction or for defining central concepts over them in an inductive manner.

(ii) Next, popular deep learning architectures—including feed forward neural networks (FNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs)—are systematically reduced to `Afflang` programs. At the end of this section, a table will be provided that lists which ONNX operators can be translated to `Afflang` programs while preserving semantics. Based on the encodings that were performed, any semantic result proven for `Afflang` automatically holds true for every deep learning model that can be expressed in it.

(iii) We will define *symbolic execution* for `Afflang`, which is a straightforward extension of its standard semantics, since the language was designed with this goal in mind. For this task, we will update the semantic definition of `Afflang` from step (i) to handle the general case of symbolic inputs. It will also trace branches in the control flow instead of evaluating them. Symbolic execution will provide a control flow graph (CFG) that captures the semantics of neural networks in a concise and universal manner based on their `Afflang` representation.

(iv) Building on the previous results, we will apply established techniques from formal (program) analysis such as *concolic execution*, *preconditions*, and *postcondi-*

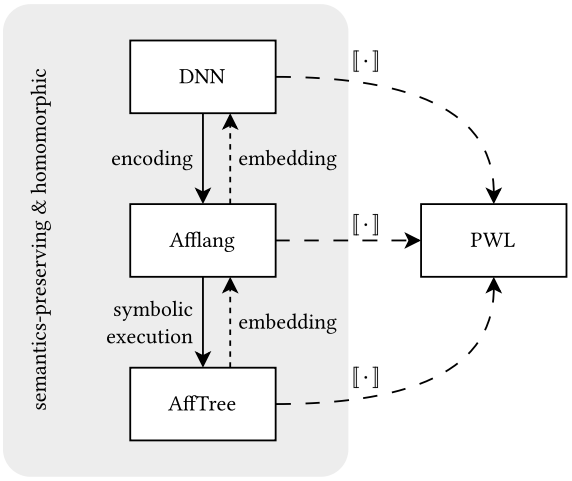


Figure 3.2: Algebraic structure of the three languages of this part: DNNs, `Afflang`, and `AffTrees`. The model transformations, i.e., *reduction* and *symbolic execution*, are semantics-preserving. They are also homomorphisms with respect to many operations, such as composition, addition, and scalar multiplication. Illustration inspired by [GS21].

tions. Based on the rich framework established in the first three sections, these new constructs can be expressed seamlessly inside `Afflang` by appending or prepending suitable code blocks. These constructs extend the analytical capabilities, enabling formal reasoning about neural networks in a compositional style.

(v) Finally, the CFGs of symbolic or concolic execution are converted into a concise decision tree structure, called `AffTree`. These are semantically equivalent to the original DNNs, but feature a representation that simplifies analysis tasks, making them a faithful yet interpretable surrogate model (FISM). Finally, the algebraic properties of `AffTrees` are discussed, which build the foundation for a simple and modular API for the practical application of this theory.

The sequence of steps (i) to (v) present a *formal* approach to distill PWL deep learning architectures into faithful and interpretable surrogate models. As will become evident in each of the five steps, the structure of DNNs is well-aligned with this approach, which is a result of the following three key properties:

1. Popular training methods like stochastic gradient descent require that DNNs are almost everywhere differentiable, and thus also (almost everywhere) continuous.
2. DNNs are typically overparameterized, resulting in huge parameter matrices and frequent use of matrix multiplication.
3. Popular activation functions, such as ReLU, are PWL.

It is therefore natural to use the extensive theory of linear algebra in all five steps as a unifying framework. Indeed, the first three steps evolve heavily around linear algebra and matrix operations. The methods developed here simplify the complexity

of neural networks while preserving their functional behavior, paving the way for interpretable and theory-grounded surrogate models.

Based on the rich algebraic structure present in DNNs, `Afflang`, and `AffTrees`, the framework is flexible and modular. As the encoding and symbolic execution preserve semantics, they are both a homomorphism with respect to any operation defined for PWL functions over all three domains (Figure 3.2). This especially includes function composition, which manifests itself in the composition of layers in DNNs, in the composition of macros in `Afflang`, and the composition of `AffTrees`. This is the foundation for the modular system: it is sufficient to state how each individual building black is transformed under the encoding or symbolic execution step. The global semantics are then automatically implied. This also enables to skip the transformation to `Afflang` in practical applications and directly map deep learning (DL) primitives to `AffTrees`.

CHAPTER 4

Operational Characterization of Piecewise Linear Functions

This chapter introduces a new language, called `Afflang`, that is simple yet expressive such that (piecewise linear) neural network architectures can be encoded in it. We will start with the foundational fully-connected feed-forward architectures and popular activation functions like ReLU. From there, we will gradually introduce more advanced and modern architectures like CNNs and RNNs. In this way, we will successively encode (piecewise linear) operators of the ONNX language within `Afflang` in a way that resembles how these operators are implemented in corresponding software.[1] Importantly, the encoding will be presented *constructively*, which is an essential prerequisite for implementing them and for discussing algorithmic properties—like running time or size—of the resulting `Afflang` programs.

This encoding condenses the huge number of deep learning constructs to a set of minimal routines, which we will refer to as *core instructions*. With that it will be easier to prove theoretical properties and also to define essential functions of the language in the next sections. Readers who are familiar with either of these topics may also find this approach easier to follow: (i) The mapping from ONNX to `Afflang` is inspired by real world DL implementations and (ii) as `Afflang` is similar to the `while` language (or similar languages used in formal research), many algorithms for formal analysis are directly applicable to it. In fact, we will directly start with a typical formal approach by defining the syntax of `Afflang` through a Backus–Naur Form (BNF), define its semantics through small-step structural operational semantics (SOS) rules, and provide a sequent calculus for typing.

The contents of this chapter might also be interesting from another perspective: How does a neural network compute its decision function from an algorithmic point of view? While the stochastic theory is valuable in understanding and analyzing the training of neural networks, it is often unclear how the parameters of a network interact with each other to create the desired function, which is complicated even

[1] It may be noted that the complete set of ONNX operators, with all its variations and configurations, is too extensive to be covered fully in this chapter. Therefore, a subset of the most popular operators is selected and presented.

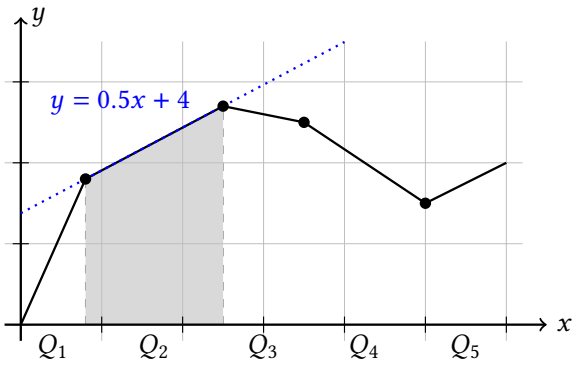


Figure 4.1: Example of a continuous PWL function $f\colon (-1,4) \to (2,4)$ with five linear regions. For the region $Q_2 = [-0.2, 1.5)$, f coincides with the linear function $g_2(x) = 0.5x+4$.

more by the fact that the desired function is in itself hard to formalize or even unknown. The analysis of what circuits or mechanisms inside a DNN govern its effective function is studied, for example, in the field of mechanistic interpretability and can help with understanding the inner structure and workings of DNNs. A typical question is where and how DNNs store facts. In this regard, we will also discuss by example how `Afflang` programs might be translated back into DNNs.

4.1 Core Instructions

We proceed by defining the minimal, imperative language `Afflang`, consisting of the *core instructions*, which are already sufficient for encoding all PWL functions. This language is designed for expressing PWL functions, which means that for (small) neighborhoods in the input, called *regions*, the function acts linearly (see example Figure 4.1). Given the nature of DNNs, the language is focused on arithmetic computations. Concretely, it consists of three kinds of *vector-valued* instructions:

- Affine transformations that allow to apply a realm of different geometric operations to their inputs, like rotations, translations, dilations, and shears;
- Conditional branching with respect to element-wise comparison;
- Referencing the input value of the program.

As all neural networks are pure, mathematical functions (i.e., without side effects like IO), the language takes inspiration from modern programming languages with a focus on expressions (which differentiates it, e.g., from the language studied in [Plo04] and also from prior work in this area [AP I: [Sch+23]]).

Definition 4.1.1 (Syntax of `Afflang`*).* The syntax of `Afflang` is defined by the following context-free grammar consisting of the basic instructions:

$$E ::= \boldsymbol{M}E + \vec{c} \mid \texttt{if}\ (E \leq \vec{c})\ \texttt{then}\ E\ \texttt{else}\ E \mid \vec{\mathbf{x}}$$

where $\boldsymbol{M}$ is a meta symbol for (type compatible) matrices, $\vec{c}$ for vectors, and $\vec{\mathbf{x}}$ is a fixed symbol for the input vector. The set of all words adhering to this grammar is

called $\mathscr{A}$ and we write $\mathscr{A}^{n\to m}$ for the subset of terms where the input $\vec{x} \in \mathbb{R}^n$ is of type n and the result is of type m. We will later define type correctness with rigor, but for now it should be clear what is meant by this informal description.

Using SOS rules, we can define the semantics of these expressions based on the transformation of configurations. We use t as a symbol for an expression *term* and $\vec{v}$ to denote the *value* of a fully-evaluated expression (to a vector with the value $\vec{v} \in \mathbb{R}^m$). Each derivation step emits a unique observable symbol from the alphabet $\Lambda = \{\text{lin}, \text{mem}, \text{if}_t, \text{if}_f\}$ as a means to track which rules were applied and in what order. The initial configuration for an `Afflang` term $t \in \mathscr{A}^{n\to m}$ evaluated for an input value $\vec{x} \in \mathbb{R}^n$ is $\langle t, \vec{\mathfrak{x}} \mapsto \vec{x}\rangle$. When a term contains subexpressions, these are evaluated first ($w \in \Lambda$):

$$\frac{\langle t, \sigma\rangle \xrightarrow{w} \langle t', \sigma'\rangle}{\langle \boldsymbol{W}t + \vec{c}, \sigma\rangle \xrightarrow{w} \langle \boldsymbol{W}t' + \vec{c}, \sigma'\rangle}$$

$$\frac{\langle t, \sigma\rangle \xrightarrow{w} \langle t', \sigma'\rangle}{\langle \texttt{if } (t \le \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \xrightarrow{w} \langle \texttt{if } (t' \le \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2, \sigma'\rangle}$$

When all subexpressions are evaluated, the current instruction is executed:

$$\frac{\forall i \in [m] : u_i = w_{i,:}\vec{v} + c_i}{\langle \boldsymbol{W}\vec{v} + \vec{c}, \sigma\rangle \xrightarrow{\text{lin}} \langle \vec{u}, \sigma\rangle}$$

$$\frac{\forall i \in [m] : v_i \le b_i}{\langle \texttt{if } (\vec{v} \le \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \xrightarrow{\text{if}_t} \langle s_1, \sigma\rangle}$$

$$\frac{\exists i \in [m] : v_i > b_i}{\langle \texttt{if } (\vec{v} \le \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \xrightarrow{\text{if}_f} \langle s_2, \sigma\rangle}$$

$$\langle \vec{\mathfrak{x}}, \sigma\rangle \xrightarrow{\text{mem}} \langle \sigma(\vec{\mathfrak{x}}), \sigma\rangle$$

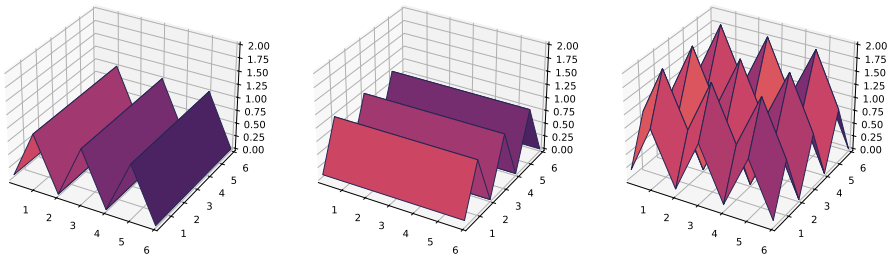
To differentiate between the syntax (arithmetic operators of `Afflang`) and the semantics (linear algebra operations), the premises are stated element-wise, which results in a distinct look.

The first two rules express the general concept that subterms can (and must) be evaluated first before one can apply the outer instruction. When a subterm is completely evaluated to a vector $\vec{v} \in \mathbb{R}^m$, the outer instruction might be applied, such as matrix multiplication (*lin*-rule) or comparison (*if*-rules). Therefore, these partially evaluated terms do not correspond to words of $\mathscr{A}$, as we did not allow to write a constant vector (we will come back to that in Section 4.2).

Notice that nested *lin* expressions have no direct benefit with respect to expressiveness (they might, however, be useful for organizational purposes) and can be simplified:

$$\boldsymbol{W_2}(\boldsymbol{W_1}\vec{\mathfrak{x}} + \vec{c_1}) + \vec{c_2} \equiv (\boldsymbol{W_2}\boldsymbol{W_1})\vec{\mathfrak{x}} + (\vec{c_1} + \vec{c_2})$$

where the matrix multiplication and vector addition on the right-hand side can be computed at 'compile time'.



(a) x-axis **(b)** y-axis **(c)** superposition

Figure 4.2: Triangle wave function along the x-Axis and y-Axis and their sum.

The language will be extended through new syntactical instructions in the next sections to match the profile of the different specialized neural architectures and analysis methods:

- Addition Extension Definition 4.4.1 (p. 50): Supports direct addition of expressions.
- Memory Extension Definition 5.3.1 (p. 69): Introduces memory operations like assignments.
- Loop Extension Definition 5.3.2 (p. 70): Adds for loops with fixed bounds for repetitive computations.
- Partial Evaluation Extension Definition 6.2.3 (p. 82): Allows to define paths as undefined.

Each of these extensions introduces new syntactical instructions with corresponding semantics and typing rules. These new rules do, however, not influence others, as each is only concerned with the integration of the new syntax. Thus, they can be considered independently. As a result, when the syntax of an extension does not appear in an `Afflang` term, like addition, then it is irrelevant whether the term is evaluated with or without the extension's semantics.

As the language is expression-focused and has neither statements nor sequential composition, we have no general rule to evaluate subexpressions first. The classic composition rule is specified in SOS as [Plo04]:

$$\frac{\langle s_1, \eta \rangle \to \langle s_1', \eta' \rangle}{\langle s_1; s_2, \eta \rangle \to \langle s_1'; s_2, \eta' \rangle}$$

where the concrete configurations depend on the language and use case, so η is a placeholder for that. We will add both statements and sequential composition later, but as they are orthogonal to the expressions already present in `Afflang` they will not influence these (will be added in Section 5.3). In the next paragraph it will be discussed why this seemingly more complex approach is suitable for the overall goal, and how other approaches have problems generalizing to dataflow graphs.

Example 4.1.2 (Triangle Wave). The *triangle wave* function is a periodic, non-sinusoidal waveform that looks like repeated triangles along an axis. It is a PWL continuous function $tr\colon \mathbb{R} \to \mathbb{R}$ with a slightly technical definition:

$$tr(x) := |(x+1 \mod 2) - 1| = \left| x - 2\left\lfloor \frac{x+1}{2} \right\rfloor \right|$$

Put simply, for $x \in [2n, 2n+1)$ the function has a slope of 1 and for $x \in [2n+1, 2n+2)$ the function has a slope of -1 (c.f. Figure 4.2a). For $x \in [0, 6)$ this function has six linear regions, and we can formulate it in `Afflang` as follows:

$$\begin{aligned}\mathtt{tr}_6 = \mathtt{if}\ (\vec{\mathbf{x}} \le 1)\ \mathtt{then}\ \vec{\mathbf{x}} \quad & \mathtt{else}\ \{\mathtt{if}\ (\vec{\mathbf{x}} \le 2)\ \mathtt{then}\ 1 - \vec{\mathbf{x}} \\ \mathtt{else}\ \{\ \mathtt{if}\ (\vec{\mathbf{x}} \le 3)\ \mathtt{then}\ \vec{\mathbf{x}} - 2 \quad & \mathtt{else}\ \{\mathtt{if}\ (\vec{\mathbf{x}} \le 4)\ \mathtt{then}\ 3 - \vec{\mathbf{x}} \\ \mathtt{else}\ \{\ \mathtt{if}\ (\vec{\mathbf{x}} \le 5)\ \mathtt{then}\ \vec{\mathbf{x}} - 4 \quad & \mathtt{else}\ \{5 - \vec{\mathbf{x}}\}\}\}\}\end{aligned}$$

Gray curly braces highlight the nesting of subexpressions as defined by the grammar for illustration purposes, and are not actually part of the term. Let's apply the SOS derivation steps for an input $\vec{\mathbf{x}} = 2.5$, so $\sigma = \vec{\mathbf{x}} \mapsto 2.5$:

$$\begin{aligned}& \langle \mathtt{if}\ (\vec{\mathbf{x}} \le 1)\ \mathtt{then}\ \vec{\mathbf{x}}\ \mathtt{else}\ \mathtt{if}\ (\vec{\mathbf{x}} \le 2)\ \mathtt{then} \cdots, \sigma \rangle \\ \xrightarrow{\text{mem}}\ & \langle \mathtt{if}\ (2.5 \le 1)\ \mathtt{then}\ \vec{\mathbf{x}}\ \mathtt{else}\ \ \mathtt{else}\ \mathtt{if}\ (\vec{\mathbf{x}} \le 2)\ \mathtt{then} \cdots, \sigma \rangle \\ \xrightarrow{\text{if}_f}\ & \langle \mathtt{if}\ (\vec{\mathbf{x}} \le 2)\ \mathtt{then}\ 1 - \vec{\mathbf{x}}\ \mathtt{else}\ \mathtt{if}\ (\vec{\mathbf{x}} \le 3)\ \mathtt{then} \cdots, \sigma \rangle \\ \xrightarrow{\text{mem}}\ & \langle \mathtt{if}\ (2.5 \le 2)\ \mathtt{then}\ 1 - \vec{\mathbf{x}}\ \mathtt{else}\ \mathtt{if}\ (\vec{\mathbf{x}} \le 3)\ \mathtt{then} \cdots, \sigma \rangle \\ \xrightarrow{\text{if}_f}\ & \langle \mathtt{if}\ (\vec{\mathbf{x}} \le 3)\ \mathtt{then}\ \vec{\mathbf{x}} - 2\ \mathtt{else} \cdots, \sigma \rangle \\ \xrightarrow{\text{mem}}\ & \langle \mathtt{if}\ (2.5 \le 3)\ \mathtt{then}\ \vec{\mathbf{x}} - 2\ \mathtt{else} \cdots, \sigma \rangle \\ \xrightarrow{\text{if}_t}\ & \langle \vec{\mathbf{x}} - 2, \sigma \rangle \\ \xrightarrow{\text{mem}}\ & \langle 2.5 - 2, \sigma \rangle \\ \xrightarrow{\text{lin}}\ & \langle 0.5, \sigma \rangle\end{aligned}$$

To make it more interesting, we may look at the superposition of two of these waves. For that, we define a function $\mathbb{R}^2 \to \mathbb{R}$ with

$$x, y \mapsto tr(x) + tr(y)$$

The resulting function now has $6 \cdot 6 = 36$ linear regions (c.f. Figure 4.2c). Notice that this term is not yet defined for `Afflang` (but will be with the next extension Definition 4.4.1). Instead, we copy the above structure for the second input, call it y, and place it at every position where a value for x is computed. The following term does not directly correspond to the grammar, but already uses some syntactic sugar

that will be formally defined later. When the intended semantics are not clear from the syntax, the example can be safely skipped.

$$
\begin{aligned}
\mathtt{tr}_{6,6}(x, y) = \mathtt{if}\ (\vec{\mathfrak{x}} \le 1)\ \mathtt{then}\ & \\
& \mathtt{if}\ (\vec{\mathfrak{y}} \le 1)\ \mathtt{then} \begin{bmatrix} \vec{\mathfrak{x}} \\ \vec{\mathfrak{y}} \end{bmatrix} \mathtt{else}\ \mathtt{if}\ (\vec{\mathfrak{y}} \le 2)\ \mathtt{then} \begin{bmatrix} \vec{\mathfrak{x}} \\ 1 - \vec{\mathfrak{y}} \end{bmatrix} \mathtt{else} \\
& \mathtt{if}\ (\vec{\mathfrak{y}} \le 3)\ \mathtt{then} \begin{bmatrix} \vec{\mathfrak{x}} \\ \vec{\mathfrak{y}} - 2 \end{bmatrix} \mathtt{else}\ \mathtt{if}\ (\vec{\mathfrak{y}} \le 4)\ \mathtt{then} \begin{bmatrix} \vec{\mathfrak{x}} \\ 3 - \vec{\mathfrak{y}} \end{bmatrix} \mathtt{else} \\
& \mathtt{if}\ (\vec{\mathfrak{y}} \le 5)\ \mathtt{then} \begin{bmatrix} \vec{\mathfrak{x}} \\ \vec{\mathfrak{y}} - 4 \end{bmatrix} \mathtt{else} \begin{bmatrix} \vec{\mathfrak{x}} \\ 5 - \vec{\mathfrak{y}} \end{bmatrix} \\
\mathtt{else}\ \mathtt{if}\ (\vec{\mathfrak{x}} \le 2)\ \mathtt{then}\ & \cdots
\end{aligned}
$$

The complete program has 35 occurrences of "if". This example illustrates that, while in `Afflang` all PWL functions can be expressed, the resulting programs can be repetitive and be hard to read. This is not a big problem as the language is intended as an intermediate language. On the other hand, on can also see that the language is a very explicit representation of the underlying PWL function, which is beneficial for the analysis.

4.2 Formal Definition

With the framework of SOS rules, the semantics are already sufficiently determined. Before moving on, we encode our intuition about these rules into mathematically precise definitions. Let $\mathscr{A}_v$ denote the set of all `Afflang` programs in which concrete values from $\mathbb{R}^+$ may appear as expressions (as is the case during evaluation). Let Γ denote the set of configurations $\langle t, \sigma \rangle$ where $t \in \mathscr{A}_v$ is an (partially evaluated) `Afflang` program and $\sigma \colon \mathscr{V} \to \mathbb{R}^+$ is a store where $\mathscr{V}$ denotes the set of variables (currently limited to $\vec{\mathfrak{x}}$). For reasoning about sequences of derivations, we introduce a notation for applying the SOS rules multiple times:

Definition 4.2.1 (Multistep Derivations). Let $\to$ be a derivation operator with observable outputs in Λ, then we define its *multistep* variant $\Rightarrow \colon \Gamma \times \Lambda^+ \to \Gamma$ as the smallest relation obeying the following laws ($\sigma \in \Lambda, w \in \Lambda^+$):

1. $c_1 \xrightarrow{\sigma} c_2 \quad \Longrightarrow \quad c_1 \overset{\sigma}{\Rightarrow} c_2$
2. $c_1 \xrightarrow{\sigma} c_2 \overset{w}{\Rightarrow} c_3 \quad \Longrightarrow \quad c_1 \overset{\sigma w}{\Longrightarrow} c_3$

Going further, we also introduce a notation that applies derivation steps holistically until a terminal state, i.e., a state where no further derivation can be applied, is reached.

Definition 4.2.2 (Terminal Derivations). Let $\to$ be a derivation operator with observable outputs Λ, then we define its *terminal* variant $\to_t \colon \Gamma \times \Lambda^+ \to \Gamma$ as the smallest relation obeying the following laws ($\sigma \in \Lambda, w \in \Lambda^+$):

$$c \overset{w}{\Rightarrow} c' \wedge \nexists \sigma : c' \xrightarrow{\sigma} c'' \quad \Longrightarrow \quad c \xrightarrow{w}_t c'$$

With these definitions, we can define a semantical functional for `Afflang` similarly to [AP I: [Sch+23]] based on derivations:

Definition 4.2.3 (Small-step Semantics of `Afflang`*).* `Afflang` terms are given a semantic function

$$[\![\cdot]\!] \colon \mathscr{A}^{n \to m} \to \mathbb{R}^n \to \mathbb{R}^m$$

which is defined as the holistic evaluation under SOS derivations:

$$[\![t]\!](\vec{x}) = \vec{v} \quad \iff \quad \langle t, \vec{\mathbf{x}} \mapsto \vec{x} \rangle \xrightarrow{w}_t \langle \vec{v}, \_ \rangle$$

When it is clear from the context we may use the shorthand notation $t(\vec{x}) := [\![t]\!](\vec{x})$.

🔍 **Excursion (Comparison with other approaches).** The approach followed here differs from the simpler approach presented in [AP I: [Sch+23]]. The following optional discussion highlights why the approach taken here is suitable for the desired goal and how it differs from [AP I: [Sch+23]]. If no such additional motivation is needed, this excursion can be skipped.

In [AP I: [Sch+23]], a set of SOS rules was presented that specified operational and denotational semantics of DNNs. In contrast to this work, there DNNs were understood as a sequence of affine functions (i.e., linear layers) and ReLU activation functions given by the BNF:

$$N ::= \varepsilon \mid \alpha; N \mid \phi; N \tag{4.1}$$

where α is a meta symbol for affine functions and ϕ for ReLUs. This language leverages the compositional structure of FNNs by closely mimicking their layered structure. As a consequence, their SOS rules are also compositional. However, this approach does not generalize to more complex architectures beyond FNNs, which are no longer organized in a strict layered form.

Instead, in this thesis DNNs are seen as dataflow graphs (Figure 3.1) over a set of (almost everywhere differentiable) operators. This view is found in the literature, for example the differentiation of DNNs is specified over computation trees [GBC16], and is also built in to the ONNX specification language.

For a concrete example, consider the addition operation, which is frequently used in modern DNNs for aggregating incoming information. We explore how this could be integrated with the layered view on DNNs as expressed by the grammar Equation (4.1). It could be stated as a big-step SOS rule as follows:

$$\frac{\langle t_1, \eta \rangle \to \langle v_1, \eta' \rangle \quad \langle t_2, \eta' \rangle \to \langle v_2, \eta'' \rangle}{\langle t_1 + t_2, \eta \rangle \to \langle v_1 + v_2, \eta'' \rangle}$$

where η is a placeholder for additional components of the configurations (like variable assignments, typing, symbol tables, scopes; see [Hüt10]). Big-step rules are, however, not well-suited for analysis and implementation. Instead, small-step rules are preferable.

Let's explore how the addition rule could be implemented as a small-step rule. One approach could be in terms of continuations, which are also well studied in the context of the lambda calculus, but also complicate reasoning. An possible

statement of the addition rule in this style would first compute the first subterm and then append a continuation in form of an addition with the resulting vector after the second subterm:

$$\frac{\langle t_1, \eta\rangle \xrightarrow{w} \langle \vec{v}_1, \eta'\rangle}{\langle t_1 + t_2, \eta\rangle \xrightarrow{w} \langle t_2; (\vec{x} \mapsto \boldsymbol{I}_n\vec{x} + \vec{v}_1), \eta'\rangle}$$

Here, the evaluation of t_1 yields a resulting vector v_1, but its effect (the act of addition) is postponed until t_2 has been fully evaluated. This is achieved by simply stating its effect as an affine function, which is queued for later evaluation.

Alternatively one can reduce it to the case of affine functions by rewriting it with block matrices:

$$\begin{bmatrix}\boldsymbol{I}_m & \boldsymbol{I}_m\end{bmatrix} \cdot \begin{bmatrix}t_1\\ t_2\end{bmatrix} \quad .$$

But what if t_1 or t_2 are not linear, i.e., cannot be written as a matrix? Then t_1 or t_2 must be decomposed into their linear regions first, which would again require to first evaluate them completely. Further, the syntax of terms appearing inside a matrix can be confusing, even if the semantics are clearly defined.

Another example where expressions are beneficial is how they allow modular combination without requiring function syntax and semantics. With the expression focus, we get functions for free through simple macro-like substitution, which also matches mathematical notation. Consider the expression

$$y = \mathrm{ReLU}(x)$$

In a non-expression based program one would write it as

$$\texttt{if}\ (x \leq 0)\ \texttt{then}\ y = 0\ \texttt{else}\ y = x$$

However, in an expression-based approach, we can equivalently write:

$$y = \texttt{if}\ (x \leq 0)\ \texttt{then}\ 0\ \texttt{else}\ x$$

which more directly mirrors the mathematical formulation. This allows us to define the ReLU function concisely as:

$$\mathrm{ReLU}(x) = \texttt{if}\ (x \leq 0)\ \texttt{then}\ 0\ \texttt{else}\ x \quad .$$

If implemented as a statement, this would instead take the form:

$$\mathrm{ReLU}(x) = \texttt{if}\ (x \leq 0)\ \texttt{then return}\ 0\ \texttt{else return}\ x$$

This representation requires function calling and with that more involved SOS rules and configurations. We can conclude that the expression-based paradigm aligns better with mathematical reasoning.

4.3 Theoretical Properties

This section discusses the theoretical properties of `Afflang`, such as its expressiveness, typing, and properties of derivations. This gives a general impression of the language, before we shift our focus to DL.

Termination. Through structural induction along the subterm relation, one can prove that the application of the above SOS rules results in finite derivation sequences for every possible initial configuration. Therefore, the evaluation of `Afflang` terms, i.e., the semantic functional $[\![\cdot]\!]$ of Definition 4.2.3, is well-defined.

Uniqueness. Every `Afflang` term has exactly one derivation sequence by the SOS rules, thus the semantic functional is unique and therefore evaluation is deterministic. This unique derivation sequence of rules also results in a unique word $w \in \Lambda^*$ of observable outputs. We may identify the derivation sequence with this word, e.g., when using the multistep derivation $c \overset{w}{\Longrightarrow} c'$.

4.3.1 Expressiveness

Although the language has only a few basic constructs, they are already sufficient for expressing all PWL functions. Intuitively, this requires the following properties:

- ability to represent linear functions,
- ability to express convex regions, and
- restriction of the domain of linear functions to convex regions.

All these properties are built into `Afflang`.

Theorem 4.3.1 (Expressiveness of `Afflang`*).* Any program $t \in \mathscr{A}^{n \to m}$ written in `Afflang` is equivalent to a (non-continuous) PWL function $f : \mathbb{R}^n \to \mathbb{R}^m$ and vice versa.

This property is the result of the design of `Afflang` and might therefore seem natural. Since this property is essential for the role of `Afflang` as our intermediate language, we prove the result in detail to illustrate all the implicit facets of the result.

Proof. Proving each direction separately:
(Sufficiency) We need to show that for any `Afflang` term $t \in \mathscr{A}$ its semantic function $[\![t]\!]$ is PWL. By Definition 2.3.1 we need to find a polyhedral partition $\mathscr{Q} \subseteq 2^{\mathbb{R}}$ (Definition 2.3.2) such that $[\![t]\!]$ is linear in each region $Q \in \mathscr{Q}$. Proof by structural induction over the subterms of t:

- $t = \vec{\mathbf{x}}$: This program corresponds to the identity function, as $\vec{\mathbf{x}}$ is the input vector. The identity function is linear and therefore also PWL.
- $t = t_1 + t_2$: By induction hypothesis $[\![t_1]\!]$ and $[\![t_2]\!]$ are PWL. As such polyhedral partitions $\mathscr{Q}_1, \mathscr{Q}_2 \subseteq 2^{\mathbb{R}}$ exists such that t_1 is linear over each region in $\mathscr{Q}_1$ and t_2 for each in $\mathscr{Q}_2$. Then $\mathscr{Q}_1 \cap \mathscr{Q}_2$ is also a polyhedral partition and for each class of this partition we know that both t_1 and t_2 are linear. Then their sum is also linear in each of these classes, and therefore $[\![t_1 + t_2]\!]$ is PWL.
- $t = \boldsymbol{M} t_1 + \vec{c}$: By induction hypothesis $[\![t_1]\!]$ is PWL, then the application of any affine function to t_1 is also PWL.
- $t = \texttt{if}\,(t_1 \leq \vec{b})\,\texttt{then}\,s_1\,\texttt{else}\,s_2$: Assume $[\![t_1]\!]$, $[\![s_1]\!]$, and $[\![s_2]\!]$ are PWL by induction hypothesis. As $[\![t_1]\!]$ is PWL there exists a polyhedral partition $Q_1, \ldots, Q_n$ such that each $f\,|_{Q_i}$ is linear. Refine this partition by splitting each Q_i into the convex regions $R_{2(i-1)+1} = \{\, \vec{x} \in Q_i \,|\, f(\vec{x}) < b \,\}$ and $R_{2i} = \{\, \vec{x} \in Q_i \,|$

$f(\vec{x}) > b\,\}$. After removing all empty regions $R_1, \ldots, R_{2n}$ is also a polyhedral partition and for each region R_i the predicate $t_1 \leq \vec{b}$ is either completely satisfied or violated for all points $x \in R_i$. By construction, it holds that for all $i \in [n]$:

$$\begin{aligned} t(x) &= s_1(x) \qquad \text{for all } x \in R_{2(i-1)+1} \\ t(x) &= s_2(x) \qquad \text{for all } x \in R_{2i} \end{aligned}$$

(Necessity) Let f be a PWL function, then there exists a polyhedral partition $\mathscr{Q} = \{Q_1, \ldots, Q_n\}$ such that f is linear in each region Q_i. For each such region, one can construct matrices $\boldsymbol{M_i}$ and $\boldsymbol{W_i}$ such that

$$\begin{aligned} Q_i &= \text{Models}(\boldsymbol{M_i}\vec{x} \leq \vec{b_i}) \\ f\,|_{Q_i} &= \boldsymbol{W_i}\vec{x} + \vec{c_i} \end{aligned}$$

We define the `Afflang` program $t_n \in \mathscr{A}$ corresponding to f recursively:

$$\begin{aligned} t_1(\vec{x}) &= \boldsymbol{W_1}\vec{\boldsymbol{x}} + \vec{c_1} \\ t_i(\vec{x}) &= \texttt{if } (\boldsymbol{M_i}\vec{\boldsymbol{x}} \leq \vec{b_i}) \texttt{ then } \boldsymbol{W_i}\vec{\boldsymbol{x}} + \vec{c_i} \texttt{ else } t_{i-1} \end{aligned}$$

where $f = t_n$. □

The proof shows that all PWL functions can be expressed in `Afflang` without needing addition of terms (Definition 4.4.1), which is one of the reasons why it is defined separately. It is now clear that all PWL DNNs can, in theory, be mapped to an equivalent `Afflang` term.

Interestingly, the reverse is also true: for every *continuous* `Afflang` term exists an equivalent DNN consisting of linear layers and ReLU activations. For the proof we refer to the following result from [Ovc02]:

Theorem 4.3.2 (Max-Min Representation). For any continuous PWL function $f\colon \Gamma \longrightarrow \mathbb{R}$ defined on a convex domain $\Gamma \subseteq \mathbb{R}^d$ with unique component functions $\{g_1, \ldots, g_k\}$, there exists a family of incomparable (with respect to $\subseteq$) index sets $\{S_j\}_{j \in J}$ such that

$$f(\vec{x}) = \max_{j \in J} \min_{i \in S_j} g_i(\vec{x})$$

holds for all $\vec{x} \in \Gamma$.

Based on this representation, one can encode any PWL function in a neural network given that one implements max and min operations. Both can be encoded as circuits of ReLU neural networks (see [Aro+18] for another circuit):

$$\begin{aligned} \max\{a, b\} &= a + \max\{0, b - a\} = a + \text{ReLU}(b - a) \\ \min\{a, b\} &= b - \max\{0, b - a\} = b - \text{ReLU}(b - a) \end{aligned}$$

Universal Approximation. Additionally, PWL functions have the *universal approximation property*, which states that any given continuous function f can be approximated with an error margin ϵ by a PWL function g with finitely many regions.

Theorem 4.3.3 (Universal Approximation). Let $f : \Omega \to \mathbb{R}$ be a continuous function on a compact domain $\Omega \subseteq \mathbb{R}^n$, and let $\epsilon \in \mathbb{R}_{>0}$ be a given error tolerance. Then, there exists a PWL function g such that:

$$\sup_{\vec{x} \in \Omega} \|f(\vec{x}) - g(\vec{x})\| \ < \ \epsilon \ .$$

Given the equivalence between `Afflang`, DNNs, and PWL functions, it follows that `Afflang` programs and DNNs also have this property. In case of DNNs, universal approximation is well-studied [HSW89; Cyb89; Les+93] and several bounds are known for the width and depth needed for approximation.

4.3.2 Continuity

As DNNs are usually continuous, it may be interesting to know when this is also true for `Afflang` terms. Of the four operations of `Afflang`, three (*lin*, *add*, *mem*) are continuous whenever all occurring subterms are continuous. However, the if-then-else construct can introduce discontinuity, for example:

$$\texttt{if}\ (\vec{\mathbf{x}} \leq 0)\ \texttt{then}\ \vec{\mathbf{x}}\ \texttt{else}\ \vec{\mathbf{x}} + 2 \ .$$

An if-then-else statement of the form $\texttt{if}\ (t \leq \vec{b})\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ is continuous iff it satisfies the following property:

$$\forall \vec{x} \in \mathbb{R}^n : t(\vec{x}) = \vec{b} \implies s_1(\vec{x}) = s_2(\vec{x}) \ .$$

This condition ensures continuity at the transition between s_1 and s_2 given by the solutions to $t(\vec{x}) = \vec{b}$. These are a subset of a hyperplane arrangement and thus a zero-volume in $\mathbb{R}^n$.

Continuity also naturally arises when combining continuous functions:

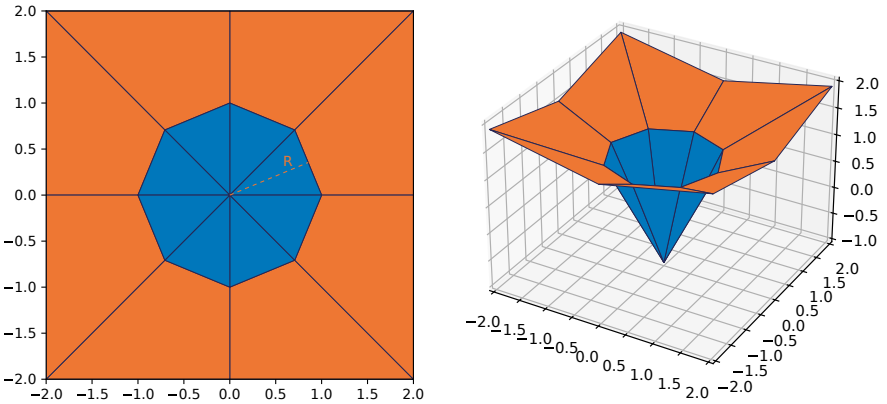
Lemma 4.3.4. A PWL function f is continuous iff there exists continuous PWL functions $f_1, \ldots, f_n$ (with $f_i \neq f$) such that

$$f = f_n \circ \cdots \circ f_1 \ .$$

Especially interesting are the cases where the functions f_i are restricted to certain function classes, like affine functions and some branching function like maximum, minimum, absolute value, or ReLU. These will be further explored in the next sections.

Lemma 4.3.5. Given a PWL function $f\colon \mathbb{R} \to \mathbb{R}$ and a continuous PWL function $\sigma\colon \mathbb{R} \to \mathbb{R}$ with two linear regions, then there exists an $n \in \mathbb{N}$ and affine functions $\alpha_1, \ldots, \alpha_n$ such that

$$f = \alpha_n \circ \sigma \circ \cdots \circ \alpha_2 \circ \sigma \circ \alpha_1$$



(a) Preimage Partition **(b)** Function Plot

Figure 4.3: A PWL function representing a flower-like shape with 16 linear regions. It has four axes of symmetry.

Example 4.3.6. A flower-like shape (Figure 4.3) with an octagonal structure transitioning from steeper slopes near the center to flatter slopes outward can be achieved by defining a *piecewise radial function*. Here's how we can define it: An octagonal norm can be defined by:

$$\begin{aligned} r_o(x,y) &= \max(|x| + \tan(\pi/8)|y|, |y| + \tan(\pi/8)|x|) \\ &= \max(|x|,|y|) + \tan(\pi/8)\min(|x|,|y|) \quad . \end{aligned}$$

That is, all level sets $\{\,(x,y) \in \mathbb{R}^2 \mid r_o(x,y) = R\,\}$ with $R \in \mathbb{R}^+$ are octagons. It can be seen as a rough approximation of the euclidean norm. Define a continuous height function $f(x,y)\colon \mathbb{R}^2 \to \mathbb{R}$ as:

$$f(x,y) = \begin{cases} k_1 \cdot r_o(x,y), & \text{if } r_o(x,y) \le R, \\ k_1 \cdot R + k_2 \cdot (r_o(x,y) - R), & \text{if } r_o(x,y) > R. \end{cases}$$

where R is the octagonal distance that marks the boundary between inner and outer half of the petals, k_1 is the slope for the inner side, and k_2 the slope for the outer side of the petals.

The function has 16 linear regions with 24 boundaries between the regions. The implied hyperplane arrangement by these regions is (c.f. Figure 4.3a)

$$\begin{aligned} x &= 0 \\ y &= 0 \\ x \pm y &= 0 \\ \pm x \pm \tan(\pi/8)y &= R \\ \pm\tan(\pi/8)x \pm y &= R \end{aligned}$$

Focusing on the upper right quadrant, the input space must be split three times. By first splitting along the main diagonal the code only requires three "if" expressions

with a depth of two. Define $\alpha = \tan\left(\frac{\pi}{8}\right)$ and let the input coordinates be given as $(\vec{\mathfrak{x}}, \vec{\mathfrak{y}}) \in \mathbb{R}^2$:

$$
\begin{aligned}
&\texttt{if } (\vec{\mathfrak{x}} - \vec{\mathfrak{y}} \leq 0) \texttt{ then} \\
&\quad \texttt{if } (\alpha\vec{\mathfrak{x}} + \vec{\mathfrak{y}} \leq R) \texttt{ then } k_1(\alpha\vec{\mathfrak{x}} + \vec{\mathfrak{y}}) \texttt{ else } k_2(\alpha\vec{\mathfrak{x}} + \vec{\mathfrak{y}} - R) + k_1 R \\
&\texttt{else} \\
&\quad \texttt{if } (\vec{\mathfrak{y}} + \alpha\vec{\mathfrak{x}} \leq R) \texttt{ then } k_1(\vec{\mathfrak{x}} + \alpha\vec{\mathfrak{y}}) \texttt{ else } k_2(\vec{\mathfrak{x}} + \alpha\vec{\mathfrak{y}} - R) + k_1 R
\end{aligned}
$$

In this mechanical form, the complete function can be written down. However, a better approach is to use the symmetry of the function and decompose the term into three PWL functions:

$$g \circ \left[(x, y) \mapsto (\max(x, y), \min(x, y))\right] \circ \left[(x, y) \mapsto (|x|, |y|)\right]$$

where

$$g(r) = \begin{cases} k_1 \cdot r, & \text{if } r \leq R, \\ k_1 \cdot R + k_2 \cdot (r - R), & \text{if } r > R. \end{cases}$$

can be encoded in `Afflang` has shown above. The `Afflang` encoding of max, min, and $|\cdot|$ will be shown in Section 5.1. This compositional thinking will become important later.

4.3.3 Typing

For the core instructions the typing rules encode compatibility of vectors and matrices:

Definition 4.3.7 (Type Correctness). An `Afflang` term $t \in \mathscr{A}$ is said to have type $n \in \mathbb{N}$ iff the judgment $\Gamma \vdash t : n$ can be proven through the following rules:

$$\frac{\Gamma \vdash t : n \qquad \boldsymbol{W} \in \mathbb{R}^{m \times n} \qquad \vec{c} \in \mathbb{R}^m}{\Gamma \vdash \boldsymbol{W}t + \vec{c} : m}$$

$$\frac{\Gamma \vdash s_1 : n \qquad \Gamma \vdash s_2 : n \qquad \Gamma \vdash t : m \qquad \vec{b} \in \mathbb{R}^m}{\Gamma \vdash \texttt{if } (t \leq \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2 : n}$$

$$\frac{\Gamma(x) = n}{\Gamma \vdash \vec{\mathfrak{x}} : n}$$

$$\frac{\Gamma \vdash t_1 : n \qquad \Gamma \vdash t_2 : n}{\Gamma \vdash t_1 + t_2 : n}$$

When we can prove that a term $t \in \mathscr{A}$ has type $\{x : n\} \vdash t : m$ we write $t \in \mathscr{A}^{n \to m}$.

The type must be preserved when evaluating:

$$\Gamma \vdash t : n \;\wedge\; \langle t, \sigma \rangle \xrightarrow{w}_t \langle \vec{v}, \_ \rangle \implies \vec{v} \in \mathbb{R}^n$$

Example 4.3.8. Consider the `Afflang` term t given by

$$\texttt{if } (\underbrace{\begin{pmatrix}1 & -1 & 1\end{pmatrix}}_{\boldsymbol{W_0}}\vec{\mathbf{x}}+\underbrace{0}_{\vec{b}_0} \leq 0) \texttt{ then } \underbrace{\begin{pmatrix}-1 & 1 & -1\\ 1 & -1 & 1\end{pmatrix}}_{\boldsymbol{W_1}}\vec{\mathbf{x}} + \underbrace{\begin{pmatrix}1\\-1\end{pmatrix}}_{\vec{b}_1} \texttt{ else } \underbrace{\begin{pmatrix}1 & -1 & 1\\ -1 & 1 & -1\end{pmatrix}}_{\boldsymbol{W_2}}\vec{\mathbf{x}} + \underbrace{\begin{pmatrix}1\\-1\end{pmatrix}}_{\vec{b}_2}$$

The type of t is $\{x : 3\} \vdash t : 2$, which can be proven using the sequent rules as follows ($\Gamma = \{x : 3\}$):

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = 3}{\Gamma \vdash \vec{\mathbf{x}} : 3} \quad \boldsymbol{W_1} \in \mathbb{R}^{2\times 3} \quad \vec{b}_1 \in \mathbb{R}^2}{\Gamma \vdash \boldsymbol{W_1}\vec{\mathbf{x}} + \vec{b}_1 : 2} \quad \dfrac{\vdots}{\Gamma \vdash \boldsymbol{W_2}\vec{\mathbf{x}} + \vec{b}_2 : 2} \quad \dfrac{\vdots}{\Gamma \vdash \boldsymbol{W_0}\vec{\mathbf{x}} + \vec{b}_0 : 1} \quad 0 \in \mathbb{R}^1}{\Gamma \vdash \texttt{if } (\boldsymbol{W_0}\vec{\mathbf{x}} + \vec{b}_0 \leq 0) \texttt{ then } \boldsymbol{W_1}\vec{\mathbf{x}} + \vec{b}_1 \texttt{ else } \boldsymbol{W_2}\vec{\mathbf{x}} + \vec{b}_2 : 2}$$

For the term t this is the only successful sequent derivation that exists.

4.4 Syntactic Sugar

Although `Afflang` is expressive on a language level, based on its minimal structure it is in some sense low-level. Now we complement the language with useful shorthands that raise the level and simplify thinking about programs. We treat these extensions as *macros* that will be substituted literally by their definition. Substitution order does not matter and will always result in the same macro-free `Afflang` program (diamond property).

4.4.1 Indexing and Slicing Operations

Frequent operations when working with vectors and matrices are indexing or slicing specific elements or rows. Interestingly, these operations are linear and can thus be introduced as syntactic sugar. Given an expression $t \in \mathscr{A}^{n\to m}$, we can slice the resulting vector by selecting the rows from $i \in \mathbb{N}$ up to (but excluding) $j \in \mathbb{N}$ as follows ($1 \leq i < j \leq m$):

$$t[i : j] \coloneqq \begin{bmatrix}\mathbf{0} & \boldsymbol{I}_{j-i} & \mathbf{0}\end{bmatrix} t$$

This block matrix corresponds to a subset of rows from the full identity matrix $\boldsymbol{I}_m$, selecting the rows where the index lies in $[i, j)$. Technically, the first $\mathbf{0}$ block matrix has shape $(j - i) \times (i - 1)$, and the last one has shape $(j - i) \times (m - j + 1)$ where m is the shape of the resulting vector of t. When i is left out, it is assumed that $i = 1$. Similarly, when j is omitted, we assume $j = m$. To select just a single element $i \in [m]$ we write:

$$t[i] \coloneqq \vec{e}_i^{\,\intercal} t$$

Here the multiplication with the unit vector allows again a reduction of the concept of indexing to linear expressions. Sometimes one wants to set the k-th element of a vector to a fixed value $c \in \mathbb{R}$. For that we introduce the following 'update' notation:

$$t[k \mapsto c] \coloneqq \boldsymbol{R_k}(\mathbf{0})t + c\vec{e_k} = \begin{bmatrix} t[: k] \\ c \\ t[k + 1 :] \end{bmatrix}$$

Here the matrix $R_k(\lambda) \in \mathbb{R}^{m \times m}$ acts solely on the k-th element ($1 \leq k \leq m$) of the input vector by multiplying it with the value $\lambda \in \mathbb{R}$, i.e.,

$$\left(\boldsymbol{R_k(\lambda)}\right)_{i,j} = \begin{cases} 0 & \text{if } i \neq j \\ \lambda & \text{if } i = j = k \\ 1 & \text{otherwise} \end{cases}$$

As the matrix is also useful later on for other values of $\lambda \neq 0$, we define it directly in this general form. The same syntax naturally extends to updates with non-constant values ($\vec{w} \in \mathbb{R}^m$), which are also linear ($k \in [m]$):

$$t[k \mapsto \vec{w}^\intercal t] \coloneqq \begin{bmatrix} \vec{e_1} & \cdots & \vec{e_{k-1}} & \vec{w} & \vec{e_{k+1}} & \cdots & \vec{e_m} \end{bmatrix}^\intercal t$$

In a more general way, we may also derive the new value from another term $t' \in \mathscr{A}^{n \to 1}$ with the addition extension:

$$t[k \mapsto t'] \coloneqq \boldsymbol{R_k(0)}t + \vec{e_k}t'$$

As t' is an arbitrary term, it may include matrix multiplications and therefore subsumes the previous cases.

4.4.2 Piecewise Linear Operators

The next set of syntactic sugar that we will add are related to standard algebraic operators of PWL functions, such as addition and scalar multiplication (Definition 2.1.11). While their direct use in neural network architectures is less frequent, they are used to implement complex operators and are also useful for analysis tasks, both of which will be shown later on.

They perform the specified operation element-wise on the resulting vector, as frequently encountered in standard linear algebra. Only the division operation is less frequent, but even this operation appears, for example, in the context of the pseudo inverse and singular-value-decomposition. Again, for a term $t \in \mathscr{A}^{n \to m}$ and a constant $c \in \mathbb{R}^+$, we define:

$$\begin{aligned} t + \vec{c} &\coloneqq \boldsymbol{I}_m t + \vec{c} \\ t - \vec{c} &\coloneqq \boldsymbol{I}_m t + (-\vec{c}) \\ \vec{c} \odot t &\coloneqq \operatorname{diag}(\vec{c})t + \vec{0} \\ \vec{c}^{-1} \odot t &\coloneqq \operatorname{diag}(\vec{c}^{-1})t + \vec{0} \\ -t &\coloneqq \operatorname{diag}(-1)t + \vec{0} \\ \vec{c} &\coloneqq \boldsymbol{0}\vec{x} + \vec{c} \end{aligned}$$

Remember that $\operatorname{diag}(\vec{c})$ is the diagonal matrix where the entries of the main diagonal are the elements of $\vec{c}$ in order. The operation of addition can be generalized further to include affine functions, as these are closed under (element-wise) addition. On the programming side, this means that we may not only add a fixed (or constant) vector to a term, but also two arbitrary terms. Thus, we can extend the expressions of `Afflang` with addition:

Definition 4.4.1 (Addition Extension of `Afflang`*).* We extend the syntax of `Afflang` (Definition 4.1.1) with the following new expression

$$E ::= \cdots \mid E + E$$

and define the semantics of this new operation in a similar small-step style:

$$\frac{\langle t_1, \sigma\rangle \xrightarrow{w} \langle t_1', \sigma'\rangle}{\langle t_1 + t_2, \sigma\rangle \xrightarrow{w} \langle t_1' + t_2, \sigma'\rangle}$$

$$\frac{\langle t_2, \sigma\rangle \xrightarrow{w} \langle t_2', \sigma'\rangle}{\langle \vec{v_1} + t_2, \sigma\rangle \xrightarrow{w} \langle \vec{v_1} + t_2', \sigma'\rangle}$$

$$\frac{\forall i \in [m] : u_i = (v_1)_i + (v_2)_i}{\langle \vec{v_1} + \vec{v_2}, \sigma\rangle \xrightarrow{\text{add}} \langle \vec{u}, \sigma\rangle}$$

Notice that a strict ordering is given here: evaluate the first subexpression t_1 completely before moving on to t_2. The order of operations is important, as subexpressions may contain side effects later on, which could violate *confluence*. It is also custom in many programming languages to evaluate (sub)expressions in the order that they appear in the program (top to bottom, left to right).

A similar extension for element-wise multiplication (i.e., $E \odot E$) is not possible, as this operation is not (in general) linear:

$$\begin{aligned}(\vec{x} \mapsto \boldsymbol{W}\vec{x}) \odot (\vec{x} \mapsto \boldsymbol{M}\vec{x}) &= \vec{x} \mapsto (\boldsymbol{W}\vec{x} \odot \boldsymbol{M}\vec{x}) \\ &= \left(\sum_i \sum_j w_{k,i} m_{k,j} x_i x_j\right)_k\end{aligned}$$

4.4.3 Case Differentiation

In the previous paragraph, we have seen that due to the expression-oriented design of our language, it is seamlessly possible to chain if-then-else blocks without explicitly defining 'else-if' as a keyword. Although chaining conditions in this manner is typical in programming, in mathematical texts another syntax is more common: *case differentiation*. The syntax requires a bit more space, but also leads to better readability. Therefore, we define how the mathematical notation can be converted into `Afflang`:

$$f(x) := \begin{cases} f_1(x) & \text{if } q_1(x) \\ f_2(x) & \text{if } q_2(x) \\ \vdots & \\ f_n(x) & \text{otherwise} \end{cases}$$

$$f(x) := \texttt{if } (q_1(x)) \texttt{ then } f_1(x) \texttt{ else if } (q_2(x)) \texttt{ then } f_2(x) \texttt{ else } \cdots f_n(x)$$

Where $f_1, \ldots, f_n \colon \mathbb{R}^n \to \mathbb{R}^m$ are type-compatible PWL functions and $q_1, \ldots, q_n$ are affine predicates. Whenever possible, the predicates should be mutually exclusive

when using the mathematical notation, as the concept of 'else if' is not clear at first glance (i.e., that the first case is taken whose condition is satisfied).

When f is continuous, there is no difference in whether the predicates are given as $\boldsymbol{A}\vec{x} \leq \vec{b}$ or $\boldsymbol{A}\vec{x} < \vec{b}$ as long as f is defined for $\boldsymbol{A}\vec{x} = \vec{b}$ by at least one case. When its value is defined by multiple cases, it is obvious that the cases have to match, as otherwise f could not be continuous at the boundary.

CHAPTER 5

Encoding of Piecewise Linear Deep Neural Networks

In this chapter, it will be shown how popular DL architectures, such as FNNs, CNNs, and RNNs, can be encoded in the previously introduced `Afflang` language. While FNNs provide a solid foundation for all other specialized architectures, CNNs and RNNs still benefit from a closer look at their specific architectural properties. These three architectures encompass various concepts and building blocks, which we will call *DL primitives*, that are typically used together, although the concrete application varies from case to case. Examples of DL primitives that will be shown in this chapter are (rectified) activation functions, convolutions, max pooling, residual connections, and dropout. Their encoding into `Afflang` programs serves two purposes:

- It is a constructive proof that the corresponding primitives are indeed PWL.[1]
- Each encoing will discuss characteristics of the corresponding DL primitives, preparing for the following chapters.

At the end, PWL operators of the ONNX format are presented that have the same semantics as the discussed DL primitives. This highlights the flexibility of `Afflang` to handle the different ways in which modern DNNs are build.

Today, deep learning encompasses a large family of different neural architectures designed and specialized for various application domains. Although these differ in details, they all share a few keep concepts that define DL. DNNs are commonly based on linear algebra enriched with non-linear but (almost everywhere) continuous and differentiable activation functions. Their architectures are optimized for fast execution on modern GPUs, making heavy use of matrix multiplication, element-wise activation functions, and parallel execution by minimizing computational dependencies [GBC16]. All of these can be seen by major breakthroughs in DL, e.g., in AlexNet [KSH12] or in self-attention with Key-Query-Value optimization [Wan+20]. For training with (stochastic) gradient descent, it is important that the network can be differentiated with respect to its learnable parameters. After training, however,

[1] Different proofs are already given in the corresponding literature, see Chapter 9.

this property is no longer relevant. Although the training of DNNs is not considered in this thesis, in the following the typical setup of *supervised learning* is assumed, which might entail assumptions that are not always met by DNNs used in unsupervised settings. We begin the encoding of DL primitives to `Afflang` programs with the foundational FNNs.

5.1 Feed Forward Neural Networks

Preliminaries (Feed Forward Neural Networks). Feed forward neural networks (FNNs) are one of the earliest architectures of DL. They consist of neurons built after Rosenblatt's perceptron model [Ros58] and synapses or connections between these neurons that transport and modify signals. In this architecture, neurons are stacked in parallel in a layer, and synapses connect all neurons from one layer to all neurons of the following layer. Input signals are passed through this architecture sequentially layer-by-layer. Although the initial inspiration were biological nervous systems, FNNs evolved over time based on empirical and theoretical observations regarding their performance.

Even today this architecture is still popular, as it is sufficient for smaller problems and provides a universal model for DNNs, as many other architectures are specializations of this model. Characteristic for FNNs is their layered structure, consisting of an alternation of (affine) parameter matrices h_ℓ and non-affine activation functions σ_ℓ. A FNN h with L layers is constructed as:

$$h : \vec{x} \mapsto \big(h_L \circ \sigma_{L-1} \circ \cdots \circ \sigma_1 \circ h_1\big)(\vec{x})$$

where $h_\ell(\vec{x}) = \boldsymbol{W_\ell}\vec{x} + \vec{b_\ell} \colon \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$ are affine functions, σ_i are non-affine, (almost everywhere) differentiable activation functions, and n_ℓ denotes the number of neurons in the ℓ-th layer ($1 \leq \ell \leq L$). A layer is the combination of the weights (synapses) h_ℓ and the activation σ_ℓ. Each component in the output vector of a layer can be interpreted as a neuron. One can directly infer from this equation that the semantic complexity of the represented function arises from the use of the activation functions. When all activation functions $\sigma_1, \ldots, \sigma_{L-1}$ are PWL, then so is the entire model h.

5.1.1 Parameter Matrices

One important primitive of deep learning is already a core part of `Afflang`: *linear layers*. In a linear layer, all neurons $a_{\ell,i}$ of the previous layer ℓ are connected with the neurons $a_{\ell+1,j}$ of the next layer $\ell + 1$. The weights of these connections are learnable parameters of the layer. In statistics and machine learning, it is custom to organize these in a weight matrix $\boldsymbol{W} \in \mathbb{R}^{n_{\ell+1} \times n_\ell}$ where $w_{j,i}$ is the weight that connects neuron $a_{\ell,i}$ with $a_{\ell+1,j}$. Additionally, a bias term $\vec{b}$ is added to the output:

$$\vec{x} \mapsto \boldsymbol{W}\vec{x} + \vec{b}$$

These are affine functions and can be specified in `Afflang` with the same syntax.

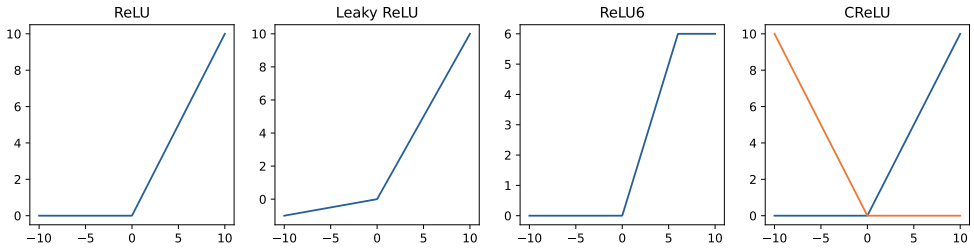


Figure 5.1: One-dimensional plots of the rectifier activation functions. For CReLU, the output is two-dimensional with the first component being shown in blue and the second in orange.

5.1.2 Activation Functions

Rectifier Activations. Today, the most used activation functions belong to the class of rectified functions (Figure 5.1), with the most prominent instance being the ReLU function. The ReLU operator is our first non-linear activation function. It is defined as the element-wise application of $\mathrm{ReLU}(x) := \max\{x, 0\}$. Through the application of the maximum it is a convex, PWL function with two linear regions $(-\infty, 0]$ and $(0, \infty)$. Applying these to an existing expression $t \in \mathscr{A}^{n \to m}$ can be written in `Afflang` as follows ($k \in [m], a \in \mathbb{R}$):

$$
\begin{aligned}
\texttt{Relu}[k](t) &:= \texttt{if } (t[k] \leq 0) \texttt{ then } t[k \mapsto 0] \texttt{ else } t \\
\texttt{LeakyRelu}[a, k](t) &:= \texttt{if } (t[k] \leq 0) \texttt{ then } \boldsymbol{R_k(a)}t \texttt{ else } t \\
\texttt{Relu6}[k](t) &:= \texttt{if } (t[k] \leq 0) \texttt{ then } t[k \mapsto 0] \texttt{ else } \backslash \\
&\qquad \texttt{if } (t[k] \leq 6) \texttt{ then } t \texttt{ else } t[k \mapsto 6] \\
\texttt{CRelu}[k](t) &:= \texttt{if } (t[k] \leq 0) \texttt{ then } t[k \mapsto 0][2k \mapsto -t[k]] \texttt{ else } t[2k \mapsto 0]
\end{aligned}
$$

Here we write $[k]$ to denote a 'compile-time' parameter of the macro.

Notice that each of the above macros act on a single element k of the input vector (result from evaluating t), which are commonly referred to as *partial* or *step* functions. In a real network, these activations are applied element-wise to the input, which makes their computation independent of each other. As this fact enables optimization potential—one can, for example, reorder the terms—it is custom in verification settings to define these partial variants. That is, the usual ReLU for $t \in \mathscr{A}^{n \to m}$ is given by

$$\texttt{Relu}(t) := \big(\texttt{Relu}[m] \circ \cdots \circ \texttt{Relu}[1]\big)(t)$$

This just corresponds to an element-wise application of ReLU. Here, the function composition is understood as

$$\big(\texttt{Relu}[2] \circ \texttt{Relu}[1]\big)(t) = \texttt{Relu}[2]\big(\texttt{Relu}[1](t)\big)$$

and can therefore also be used with our macros. When deemed necessary, one could also formalize this theory with the concept of *free variables*, but the meaning of composition should be clear from the context when we use it here. For CReLU, one first has to pad the resulting vector of t with m zeros:

$$\texttt{CRelu}(t) := \big(\texttt{CRelu}[m] \circ \cdots \circ \texttt{CRelu}[1]\big)(t \otimes \vec{0})$$

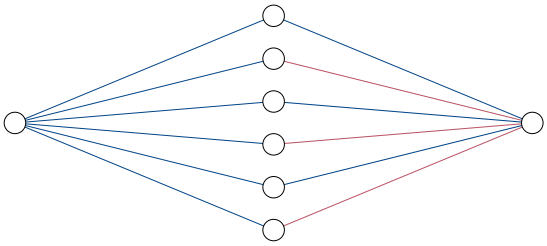


Figure 5.2: Neural Network that models the triangle wave function tr_6 on the interval $[0, 6)$. The first linear transformation (six blue lines) shift the input. The following ReLU applications each create one 'kink'. Then the resulting shifted ReLUs are scaled and added (three blue and three red lines).

Other variants, like *parametric ReLU* or *randomized ReLU*, are different during training from the previously introduced ReLUs, but behave like Leaky ReLU during evaluation. For example, parametric ReLU has a learnable parameter α for the slope in the negative part $(-\infty, 0]$. When training has completed, the parameter stays fixed. For randomized ReLU, the negative slope is sampled during training from a uniform distribution $\mathscr{U}(l, u)$ while at evaluation it is fixed to $\frac{l+u}{2}$.

Example 5.1.1 (Triangle Wave Network). Revisiting our triangle wave example (Example 4.1.2), we can define the function $tr(x)$ as a FNN with ReLU activations. The key idea is here to use the ReLU function to create one 'kink' of the wave by shifting its zero point appropriately:

$$\begin{aligned} tr(x) &= \mathrm{ReLU}(x) - 2\,\mathrm{ReLU}(x-1) + 2\,\mathrm{ReLU}(x-2) - 2\,\mathrm{ReLU}(x-3) + \cdots \\ &= \mathrm{ReLU}(x) + \sum_{i=1}^{\infty} (-1)^i 2\,\mathrm{ReLU}(x-i) \end{aligned}$$

When limiting $x \in [0, 6)$ it can be modeled by 6 ReLU applications in `Afflang`:

$$\begin{aligned} \mathtt{tr}_6 = \quad & \mathtt{Relu}[1](\vec{\mathbf{x}}) \quad - 2\mathtt{Relu}[1](\vec{\mathbf{x}} - 1) + 2\mathtt{Relu}[1](\vec{\mathbf{x}} - 2) \\ & - 2\mathtt{Relu}[1](\vec{\mathbf{x}} - 3) + 2\mathtt{Relu}[1](\vec{\mathbf{x}} - 4) - 2\mathtt{Relu}[1](\vec{\mathbf{x}} - 5) \end{aligned} \tag{5.1}$$

When expanding the ReLU macros we get the `Afflang` program

$$\begin{aligned} & \mathtt{if}\ (\vec{e}_1\vec{\mathbf{x}} \le 0)\ \mathtt{then}\ \boldsymbol{R_1(0)}\vec{\mathbf{x}} + 0\vec{e}_1\ \mathtt{else}\ \vec{\mathbf{x}} \\ + (-2)\ & \mathtt{if}\ (\vec{e}_1(\vec{\mathbf{x}} + (-1)) \le 0)\ \mathtt{then}\ \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e}_1\ \mathtt{else}\ (\vec{\mathbf{x}} + (-1)) \\ + (+2)\ & \mathtt{if}\ (\vec{e}_1(\vec{\mathbf{x}} + (-2)) \le 0)\ \mathtt{then}\ \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-2)) + 0\vec{e}_1\ \mathtt{else}\ (\vec{\mathbf{x}} + (-2)) \\ + (-2)\ & \mathtt{if}\ (\vec{e}_1(\vec{\mathbf{x}} + (-3)) \le 0)\ \mathtt{then}\ \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-3)) + 0\vec{e}_1\ \mathtt{else}\ (\vec{\mathbf{x}} + (-3)) \\ + (+2)\ & \mathtt{if}\ (\vec{e}_1(\vec{\mathbf{x}} + (-4)) \le 0)\ \mathtt{then}\ \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-4)) + 0\vec{e}_1\ \mathtt{else}\ (\vec{\mathbf{x}} + (-4)) \\ + (-2)\ & \mathtt{if}\ (\vec{e}_1(\vec{\mathbf{x}} + (-5)) \le 0)\ \mathtt{then}\ \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-5)) + 0\vec{e}_1\ \mathtt{else}\ (\vec{\mathbf{x}} + (-5)) \end{aligned}$$

Let's evaluate this term for $\vec{\mathbf{x}} = 2.5$ and see where it differs from the previous example. As the complete evaluation is very space intensive, we decompose it into multiple steps. First, consider one of the "if" branches:

$$
\begin{aligned}
& \texttt{if } (\vec{e_1}(\vec{\mathbf{x}} + (-1)) \leq 0) \texttt{ then } \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e_1} \texttt{ else } \vec{\mathbf{x}} + (-1) \\
\xrightarrow{\text{mem}} & \texttt{if } (\vec{e_1}(2.5 + (-1)) \leq 0) \texttt{ then } \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e_1} \texttt{ else } \vec{\mathbf{x}} + (-1) \\
\xrightarrow{\text{lin}} & \texttt{if } (1.5 \leq 0) \texttt{ then } \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e_1} \texttt{ else } \vec{\mathbf{x}} + (-1) \\
\xrightarrow{\text{if}_f} & \vec{\mathbf{x}} + (-1) \\
\xrightarrow{\text{mem}} & 2.5 + (-1) \\
\xrightarrow{\text{lin}} & 1.5
\end{aligned}
$$

And for an "if" branch that is satisfied:

$$
\begin{aligned}
& \texttt{if } (\vec{e_1}(\vec{\mathbf{x}} + (-3)) \leq 0) \texttt{ then } \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e_1} \texttt{ else } (\vec{\mathbf{x}} + (-1)) \\
\xRightarrow{\text{mem,lin}} & \texttt{if } (-0.5 \leq 0) \texttt{ then } \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e_1} \texttt{ else } (\vec{\mathbf{x}} + (-1)) \\
\xrightarrow{\text{if}_t} & \boldsymbol{R_1(0)}(\vec{\mathbf{x}} + (-1)) + 0\vec{e_1} \\
\xRightarrow{\text{mem,lin,lin}} & 0
\end{aligned}
$$

After applying these derivation steps to all if branches, we receive the partially evaluated term (i.e., from $\mathscr{A}_v$):

$$2.5 + (-2)1.5 + (+2)0.5 + (-2)0 + (+2)0 + (-2)0$$

Finally, after applying the *add* and *lin* rules, the term evaluates to 0.5. Thus, this formulation of the triangle wave requires more derivation steps, which is, however, not important for the analysis presented later.

We can reorganize the terms of Equation (5.1) in a way that closer resembles the layered structure of FNNs:

$$\begin{pmatrix} 1 & -2 & 2 & -2 & 2 & -2 \end{pmatrix} \text{ReLU}\left(\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \vec{\mathbf{x}} + \begin{pmatrix} 0 \\ -1 \\ -2 \\ -3 \\ -4 \\ -5 \end{pmatrix}\right)$$

This representation focuses more on the computations that happen in parallel, but it can be harder to read. The resulting network is shown in Figure 5.2.

This construction is used in [Mon+14] to prove bounds for the number of linear regions ReLU networks can exhibit. Take any continuous PWL function f that is defined for the hypercube $f\colon [0,1]^n \to \mathbb{R}^m$, then appending f to this layer results in many mirrored versions of f.

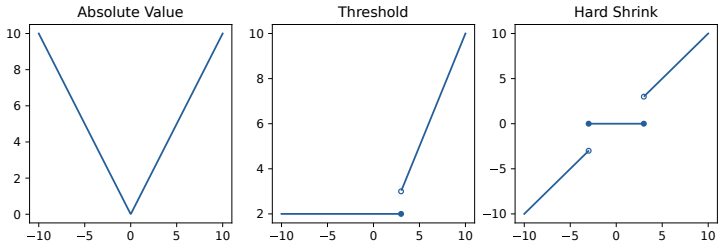


Figure 5.3: One-dimensional plots of common PWL activation functions. For the illustrated threshold function the parameters are set to $\lambda = 3$ and $a = 2$, and for hard shrink to $\mu = 3$.

PWL Activations. In addition to the rectifier family, a few other (less-frequent) activation functions are also PWL. Their function plots are shown in Figure 5.3. For $a, \lambda \in \mathbb{R}$ and $\mu \in \mathbb{R}_{>0}$ we define:

$$\texttt{Abs}[k](t) := \begin{cases} t[k \mapsto -t[k]] & \text{if } t[k] \leq 0 \\ t & \text{otherwise} \end{cases}$$

$$\texttt{Threshold}[\lambda, a, k](t) := \begin{cases} t[k \mapsto a] & \text{if } t[k] \leq \lambda \\ t & \text{otherwise} \end{cases}$$

$$\texttt{HardShrink}[\mu, k](t) := \begin{cases} t & \text{if } t[k] < -\mu \\ t[k \mapsto 0] & \text{if } -\mu \leq t[k] \leq \mu \\ t & \text{otherwise} \end{cases}$$

The absolute value function is continuous, the threshold function has a jump discontinuity at λ when $\lambda \neq a$, and the hard shrink function has two jump discontinuities at $-\mu$ and μ. In its exact form, the last one can only be defined when allowing negation in path conditions. In some cases a function called *thresholded ReLU* is defined, which is a special case of the above Threshold function where $a = 0$.

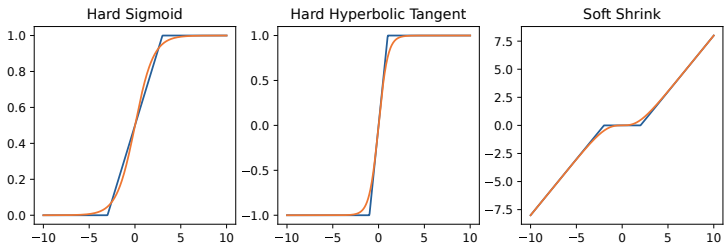


Figure 5.4: One-dimensional plots of common non-rectifier activation functions. PWL approximations of the original function are shown in orange. For soft shrink $\mu = 2$.

Activation Functions. For completeness, we also consider a few other choices for activation functions. Although the rectifier family is the predominant family of activation functions today, in the past many other functions were also successfully used in training, most notably *sigmoid* and *hyperbolic tangent*. Today, they are mostly used when the output of a neuron should be bounded, like in a memory gate. Although these are not PWL itself, there exist established PWL approximations (e.g.,

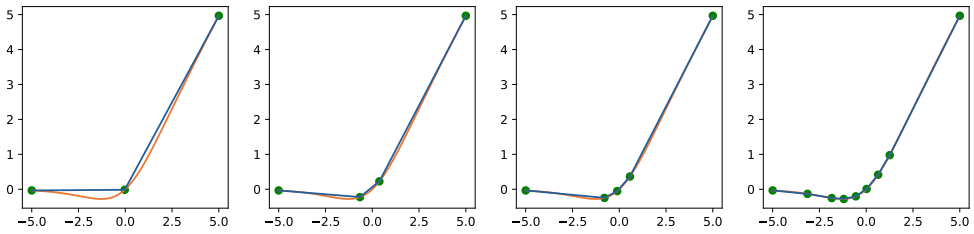


Figure 5.5: Approximation of the non-PWL *swish* activation function by 2, 3, 4, and 8 linear regions in the interval $[-5, 5]$.

in PyTorch) of them, which are usually denoted with the prefix 'hard'. The following three functions are plotted in Figure 5.4. They are defined as ($\mu \in \mathbb{R}_{>0}$):

$$\texttt{HardSigmoid}[k](t) := \begin{cases} t[k \mapsto 0] & \text{if } t[k] \leq -3 \\ t\left[k \mapsto \frac{t[k]}{6} + \frac{1}{2}\right] & \text{if } -3 < t[k] \leq +3 \\ t[k \mapsto 1] & \text{otherwise} \end{cases}$$

$$\texttt{HardTanh}[k](t) := \begin{cases} t[k \mapsto -1] & \text{if } t[k] \leq -1 \\ t & \text{if } -1 < t[k] \leq +1 \\ t[k \mapsto +1] & \text{otherwise} \end{cases}$$

$$\texttt{SoftShrink}[\mu, k](t) := \begin{cases} t + \mu\vec{e_k} & \text{if } t[k] \leq -\mu \\ t[k \mapsto 0] & \text{if } -\mu < t[k] \leq \mu \\ t - \mu\vec{e_k} & \text{otherwise} \end{cases}$$

All three functions are continuous.

In general, one can approximate any continuous function $\mathbb{R} \to \mathbb{R}$ (including other activation functions) using PWL functions. Based on the universal approximation property (Theorem 4.3.3), the error of the approximation can be bounded arbitrarily tight given enough regions. An example is shown in Figure 5.5 that approximates the *swish* [RZL17] function with an increasing number of regions.

5.1.3 Maximum and Minimum

In the realm of PWL functions, maximum and minimum are perhaps the most fundamental functions, as many normal forms for PWL functions are built from these (Section 2.3). With maximum, one can easily define convex functions and with minimum concave ones. We may define them in `Afflang` as follows:

$$\texttt{max}(t_1, t_2) := \texttt{if } (t_1 - t_2 \leq 0) \texttt{ then } t_2 \texttt{ else } t_1$$
$$\texttt{min}(t_1, t_2) := \texttt{if } (t_1 - t_2 \leq 0) \texttt{ then } t_1 \texttt{ else } t_2$$

An n-ary application of the maximum can be defined by nesting the binary version in a way that reassembles a tournament tree:

$$\begin{aligned} \texttt{max}(t_1, \dots t_n) &= \texttt{max}\left(\texttt{max}(t_1, \dots, t_r), \texttt{max}(t_{r+1}, \dots, t_n)\right) \\ \texttt{min}(t_1, \dots t_n) &= \texttt{min}\left(\texttt{min}(t_1, \dots, t_r), \texttt{min}(t_{r+1}, \dots, t_n)\right) \\ r &= 2^{\lceil \log_2 n \rceil - 1} \end{aligned}$$

where r is the biggest power of two that is smaller than n.

5.1.4 Argmax and Softmax

Preliminaries (Softmax). Another function that is very important in deep learning, specifically in classification tasks, is the softmax: $\mathbb{R}^K \to (0,1)^K$ function defined as:

$$\text{softmax}(x)_k := \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} \quad .$$

It transforms the real input vector $\vec{x} \in \mathbb{R}^K$ into a normalized vector of probabilities $(0,1)^K$, implying a probability distribution over K possible outcomes. Therefore, it is frequently used in multiclass classification settings, such as multinomial logistic regression or DNNs.

It integrates well into the gradient-based optimization of DNNs due to its differentiability. Additionally, properties like monotonicity and its closeness to log probabilities are beneficial for convergence during training. In DNNs trained for classification over K categories, it is typically added as the last layer to transform the emitted real vectors (logits) into probabilities. During (supervised) training, the loss is computed over the probability vector with respect to the labels, while at evaluation, a winner-takes-it-all rule is applied to derive the predicted class $c \in \{1, \ldots, K\}$. Therefore, it is indistinguishable from the PWL argmax function at evaluation. However, when it is used for its probabilistic interpretation, like in attention heads, it cannot be replaced as easily.

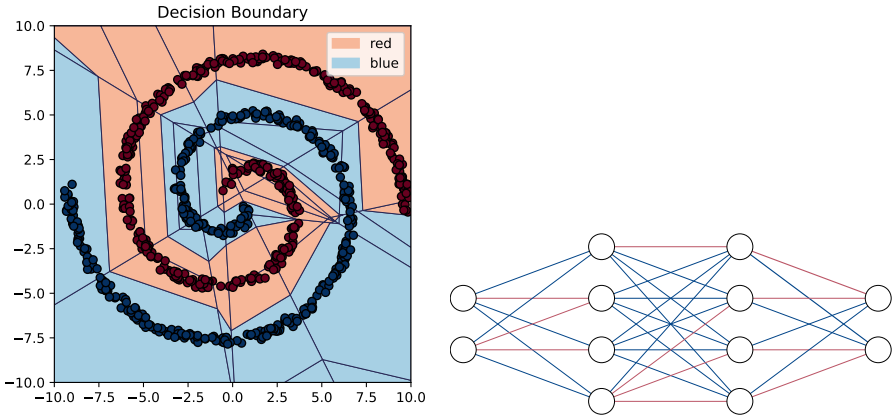
We define the argmax for a term $t \in \mathscr{A}^{n \to m}$ as a tournament tree using a recursive definition based on two indices k_1 and k_2 with $1 \leq k_2 < k_1 \leq m + 1$:

$$\texttt{argmax}(t) := \texttt{argmax}[2,1](t)$$

$$\texttt{argmax}[k_1,k_2](t) := \begin{cases} \texttt{argmax}[k_1+1,k_1](t) & \text{if } t[k_2] - t[k_1] \leq 0 \wedge k_1 \leq m \\ \texttt{argmax}[k_1+1,k_2](t) & \text{if } t[k_2] - t[k_1] > 0 \wedge k_1 \leq m \\ k_2 & \text{otherwise} \end{cases}$$

The indices k_1 and k_2 list the two components of the input vector that are compared in this round, where the winner will proceed to the next round. The first index k_1 denotes the indices up to which all rounds have been performed. The second index $k_2 < k_1$ denotes the current ongoing maximum value of all previous rounds, i.e., of indices 1 to $k_1 - 1$. And m denotes the number of components in t, i.e., t will evaluate to a vector $\mathbb{R}^m$. This definition is carefully designed to return the first (i.e., smallest) index that contains the maximal element of t in case of ties. For example, in the vector $(3,4,1,4)$ index 2 would be returned instead of index 4.

Example 5.1.2. Let's evaluate the argmax definition for the vector $\vec{v} = (3,2,4,1)$. As the complete program is long and therefore hard to read, we just follow the path that is relevant for the concrete input $\vec{v}$. Therefore, we only expand the necessary macros:



(a) Dataset and Decision Boundary **(b)** FNN

Figure 5.6: Spiral dataset and decision boundary of a trained FNN.

$$
\begin{aligned}
\texttt{argmax}(\vec{v}) &= \texttt{argmax}[2,1](\vec{v}) \\
&= \texttt{if}\ (v_2 - v_1 \leq 0)\ \texttt{then}\ \texttt{argmax}[3,1](\vec{v})\ \texttt{else}\ \texttt{argmax}[3,2](\vec{v}) \\
&\xrightarrow{\text{if}_t} \texttt{argmax}[3,1](\vec{v}) \\
&= \texttt{if}\ (v_3 - v_1 \leq 0)\ \texttt{then}\ \texttt{argmax}[4,1](\vec{v})\ \texttt{else}\ \texttt{argmax}[4,3](\vec{v}) \\
&\xrightarrow{\text{if}_f} \texttt{argmax}[4,3](\vec{v}) \\
&= \texttt{if}\ (v_4 - v_3 \leq 0)\ \texttt{then}\ \texttt{argmax}[5,3](\vec{v})\ \texttt{else}\ \texttt{argmax}[5,4](\vec{v}) \\
&\xrightarrow{\text{if}_t} \texttt{argmax}[5,3](\vec{v}) \\
&= v_3
\end{aligned}
$$

So the maximum is computed to be at index three, which is correct.

Concat. Concatenation can be useful in DNNs to combine the result of independent computations that should be processed together from this point forwards. It can be quickly defined using block matrices ($t_1 \in \mathscr{A}^{n \to m_1}$, $t_2 \in \mathscr{A}^{n \to m_2}$):

$$t_1 \otimes t_2 := \begin{bmatrix} \boldsymbol{I}_{m_1} \\ \boldsymbol{0} \end{bmatrix} t_1 + \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{I}_{m_2} \end{bmatrix} t_2$$

Here, we use the fact that in `Afflang` vectors can easily change their length when multiplied with the right matrix. This operation can be interpreted as a product, hence the symbol $\otimes$.

Example 5.1.3. Now we conclude this section with a small FNN that was trained on a two-dimensional synthetic dataset. First, the dataset consists of points from two intertwined spirals with randomly added noise (Figure 5.6a). For an angle $\theta \in (0, 3\pi)$, the red and blue spirals are described by:

$$
\begin{aligned}
r &= (-\cos(\theta) \cdot \theta, \quad \sin(\theta) \cdot \theta) \\
b &= (\ \ \cos(\theta) \cdot \theta, -\sin(\theta) \cdot \theta)
\end{aligned}
$$

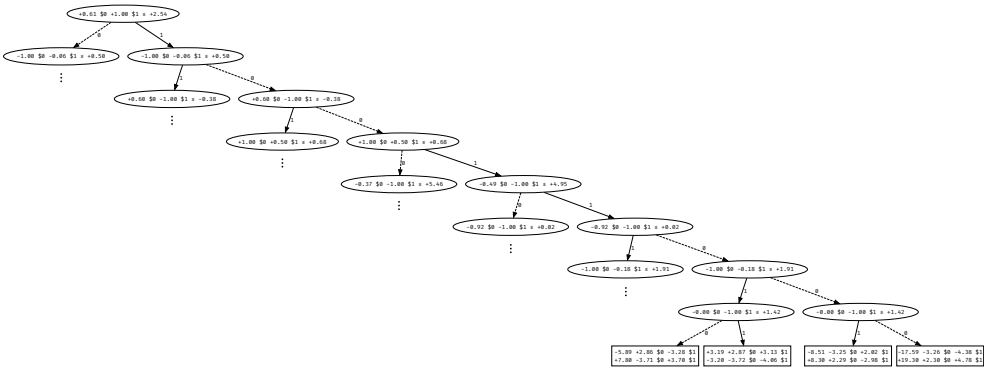
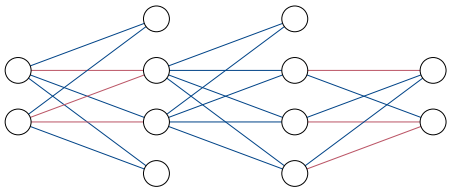


Figure 5.7: Decisions implied by each neuron of the spiral DNN along the path that the point $p = (-2.56, -2.49)$ takes. Decisions diverging from this path are left out for readability.

The task is to separate the red points from the blue points. As can be seen, they are effectively half a circle apart. Then, a fully-connected feed-forward neural network consisting of two hidden layers with five neurons each and ReLU activation is trained over this datatset. Here we may skip training details. The resulting classifier achieves an accuracy of 100% on training and test set. The learned decision function is shown in Figure 5.6.

Consider the point $p = (-2.56, -2.49)$, which is classified as "red" by the DNN. When passing p through the layers of the network, some ReLUs will be *active* when the value is greater than zero and some will be *inactive* when the value is lower than or equal to zero. For p, these *activation patterns* of the hidden neurons can be expressed as (i, a, a, i) and (i, a, a, a) where a stands for active and i for inactive. Alternatively, one can visualize the effect by plotting only the edges of the DNN that go out of active neurons:



In the `Afflang` encoding of the DNN, we can find the active and inactive neurons directly: the corresponding condition $\langle \vec{w}, \vec{u} \rangle \leq 0$ in the if expression is false when the neuron is active and true otherwise. The mapping from if expressions to neurons is also direct: the nesting level of the if expression corresponds to the position of the neuron in the DNN. The path p takes through the `Afflang` encoding is visualized as a tree in Figure 5.7. In Chapter 6, it will be show how such representations can be generated from an `Afflang` program.

We have seen how many foundational components of FNNs can be encoded in `Afflang` programs, by which we also proved their piecewise linearity. With that we prepared them for a thorough analysis with formal methods, which will be explored starting with symbolic execution in Section 6.1.

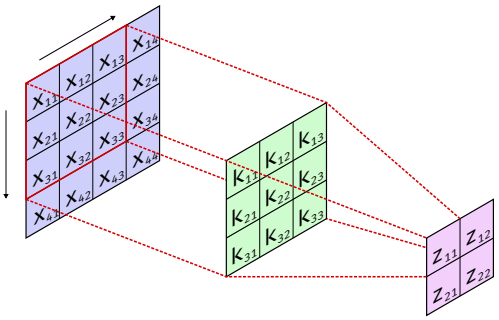


Figure 5.8: Example of a convolution step over a $1 \times 4 \times 4$ input image with a kernel of shape $1 \times 1 \times 3 \times 3$. The receptive field (red) selects a 3×3 window of values from the image, which are then convolved with the filter resulting in a linear combination. The resulting value makes up one pixel z_{11} in the output. The channel indices are left out for readability purposes.

5.2 Convolutional Neural Networks

While FNNs establish important concepts and methods for deep learning, other architectures expand on their ideas and bring them to specific application domains. CNNs were one of the first specialized architectures focusing on image data. Convolutions extract patterns from images, like edges, by sliding a fixed window of coefficients, called *kernel* or *filter*, over the image. Although many interesting filters were handcrafted by humans, the true power of DL comes by parameterizing these filters so that end-to-end optimization can be applied. CNNs are further improved with techniques like padding, pooling, dropout, normalization, and residual connections. A detailed introduction is given in Section 2.4.2 of the preliminaries.

In this section, we will use tensor notation—a concise and efficient way to specify index-based access to multi-dimensional arrays. Tensor notation is often used in the field of CNNs to represent data—like input images, feature maps, and kernels—typically featuring three to four indices. All arrays are assumed to be stored and indexed in row-major order. As discussed in Section 2.4.1, multi-dimensional arrays can be emulated by one-dimensional arrays by converting the indices accordingly. Thus, to ease notation, we use the tensor notation and expect that the indices are transformed to one-dimensional arrays to be compatible with the definitions of `Afflang`.

5.2.1 Convolution

To give a brief recapitulation, we outlined step-by-step the different convolutions that lead to *strided, multichannel convolution* for three-dimensional input arrays $i^{\beta,\mu,\nu}$ (Figure 5.8). It is expressed mathematically as:

$$z^{\alpha,i,j} = \sum_{\beta=1}^{C_{in}} \sum_{\mu=1}^{F} \sum_{\nu=1}^{F} i^{\beta,s(i-1)+\mu,s(j-1)+\nu} k^{\alpha,\beta,\mu,\nu} \qquad \text{(2.6 revisited)}$$

where $k^{\alpha,\beta,\mu,\nu}$ is a four-dimensional filter or kernel of learnable parameters and $z^{\alpha,i,j}$ is the resulting output image (with channels).

As discussed there, strided, multichannel convolution is a linear operation and can therefore be expressed as a matrix, in this specific case a Toeplitz matrix (see Equation (2.3) on page 24). Although the conversion to Toeplitz matrices is intriguing, it results in very large matrices as it scales almost quadratic with the input size of the image. Let us illustrate this on a concrete example by analyzing the first convolutional layer of the popular AlexNet released in 2011. This layer has 3 input channels, 96 output channels, and a kernel of size 11×11. It expects an input image of shape $3 \times 224 \times 224$ and produces an output of shape $96 \times 54 \times 54$. Converted to a Toeplitz matrix, it would have around $42 \cdot 10^9$ elements, out of which more than $54 \cdot 10^6$ would be non-zero. It would be very costly even when stored as a sparse matrix. Note that this does only concern the representational overhead of the convolution layer, i.e., this price is paid even when we would analyze the network for a constant input. On the other hand, it still represents the same linear function, so it would not introduce new linear regions.

As the conversion to a Toeplitz matrix is too costly, implementations typically take a different route. To make use of the highly optimized matrix multiplication algorithms, the computation is still posed as a matrix-matrix product, however, with more involved rewriting of both the kernel and the input image. One such approach, called *im2col*, is presented in the next paragraph for the reader interested in implementation details. However, these details are not required for the understanding of the rest of the thesis.

From a pure mathematical point of view, the fact that convolution is linear is sufficient, as any proof over `Afflang` must prove the property at hand for the affine construct of the core language, from which its validity for convolution automatically follows. With Equation (2.6) we also have a specific, constructive formula that tells us what we can expect as the output of a convolutional layer. Thus we define

$$\mathsf{Conv}[k^{\alpha,\beta,\mu,\nu}, s](t) \coloneqq C(k, s)t$$

where $C(k, s)$ is the corresponding Toeplitz matrix; while keeping in mind that the actual implementation is optimized.

Excursion (Im2Col). Direct computation of convolution is often inefficient due to repeated access to overlapping regions in the input $\boldsymbol{I}$. The *im2col* (image to columns) technique is commonly used to optimize this process. Notice that each convolution essentially corresponds to the dot product of the kernel with a sliding window of values from the input. This window is shifted over both axes of the input according to the stride. In im2col, the input tensor $\boldsymbol{I}$ is rearranged into a matrix $\hat{\boldsymbol{I}}$ where each column represents $C_{in} \cdot F \cdot F$ values for one such sliding window. This reshaped input matrix $\hat{\boldsymbol{I}}$ has dimensions $(C_{in} \cdot F \cdot F) \times (H_{out} \cdot W_{out})$, allowing the entire convolution to be computed as a single matrix multiplication:

$$I \star K = \hat{\boldsymbol{K}}\hat{\boldsymbol{I}}$$

where the kernel tensor $\hat{\boldsymbol{K}}$ is reinterpreted as shape $C_{out} \times (C_{in} \cdot F \cdot F)$.

From the desired shape $(C_{in} \cdot F \cdot F) \times (H_{out} \cdot W_{out})$ we can derive that each column of $\hat{\boldsymbol{I}}$ corresponds to one receptive field of the convolution (respectively the chaining of C_{in} many receptive fields), thus for each column of $\hat{\boldsymbol{I}}$ the receptive field

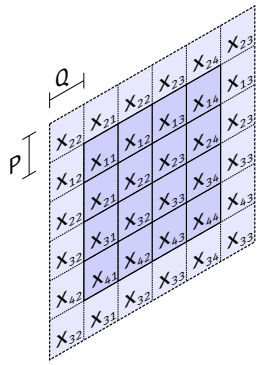


Figure 5.9: Example of mirror padding applied to a $1 \times 4 \times 4$ input image. Here $P = 1$ and $Q = 1$ meaning that only one pixel is added as padding in each of the four directions.

does not move and has therefore a fixed form. Since the height F and width F of the field are also fixed it is uniquely determined by the starting coordinates (i, j) inside the image. And as these stay fixed for the entire column these coordinates only dependent on the column index of $\hat{\boldsymbol{I}}$. The following formulas are easier to write with zero-based indexing. Assume the column index has value $c = iW_{out} + j$ with $j < W_{out}$, then the receptive field begins at (i, j). This follows from row-major order, which results exactly in these lexicographic access patterns where the lowest index is incremented the most and when it reaches its maximum, it increments the next highest and is reset to zero. Next, we calculate the offset inside this receptive field. Assume the row index is $r = \mu F + \nu$ with $\nu < F$, then the relative offset inside the kernel is (μ, ν). Finally, the channel number states how many kernels of size F^2 must be skipped. This results in the following formula for computing the values:

$$\hat{\boldsymbol{I}}^{\beta F^2 + \mu F + \nu, iW_{out} + j} = \boldsymbol{I}^{\beta, \mu + i, \nu + j}$$

where each variable has the ranges

$$0 \leq \beta < C_{in} \qquad 0 \leq i < H_{out}$$
$$0 \leq \mu < F \qquad 0 \leq j < W_{out}$$
$$0 \leq \nu < F$$

5.2.2 Padding

Preliminaries (Padding). When applying convolution, the output image shrinks with respect to the input as the receptive field $F \times F$ must be contained completely in the input. Therefore, not all pixels appear in the same number of receptive fields: While central pixels appear in F^2 many receptive fields (essentially one time at each position of the field), border pixels appear at most in F many, and corner pixels in exactly one. This asymmetry can cause problems such as information loss.

A simple technique to mitigate this is to pad the borders of the image. There are different methods for deciding what values are used as padding: One can simply fill the margin with constant values such as zero, or one can reflect or wrap the original image (Figure 5.9).

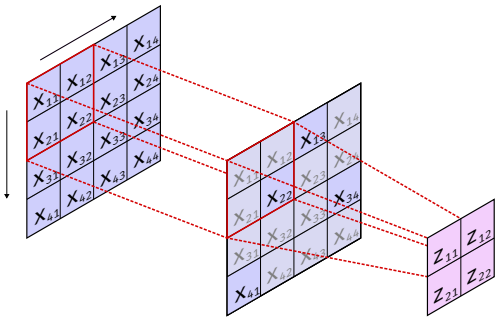


Figure 5.10: Example of 2×2 max pooling applied to a $1 \times 4 \times 4$ input image. In the second plane, maximal values are highlighted while the other three (smaller) values of the window are grayed out. The output only corresponds to the maximum value of each window in the same order.

For an input tensor $x^{\alpha,\mu,\nu}$ (where $\alpha \in [C_{in}]$, $\mu \in [H]$, $\nu \in [W]$), we define in the padding operations in `Aflang` with a horizontal padding of $Q \in \mathbb{N}$ pixels and a vertical of $P \in \mathbb{N}$ pixels to either side as $(-P < i \leq H + P, -Q < j < W + Q)$:

$$\texttt{PadZero}[P,Q](x)^{\alpha,i+P,j+Q} := \begin{cases} x^{\alpha,i,j} & \text{if } 0 < i \leq H \wedge 0 < j \leq W \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{PadReflect}[P,Q](x)^{\alpha,i+P,j+Q} := x^{\alpha,i',j'} \quad \text{where}$$

$$i' := \begin{cases} 2 - i & \text{if } -P < i \leq 0 \\ i & \text{if } 0 < i \leq H \\ 2H - i & \text{if } H < i \leq H + P \end{cases}$$

$$j' := \begin{cases} 2 - j & \text{if } -Q < j \leq 0 \\ j & \text{if } 0 < j \leq W \\ 2W - j & \text{if } W < j \leq W + Q \end{cases}$$

It is important to note that all case distinctions only concern the index and do not introduce conditional branches based on the value of the input. Therefore, they only have one linear region.

5.2.3 Max Pooling

Preliminaries (Pooling). Max Pooling [LeC+98] is a downsampling technique that also divides the input image into square regions $F \times F$ over which it computes the maximum (Figure 5.10). In contrast to convolution, the stride is usually not one, but the size of the receptive field F. Thus, it divides the input into non-overlapping regions while still covering the complete image. Its output has therefore shape

$$H_{out} = \left\lfloor \frac{H}{F} \right\rfloor \qquad W_{out} = \left\lfloor \frac{W}{F} \right\rfloor$$

The max pooling operation is permutation invariant, meaning the order of pixels inside each receptive field is irrelevant. A special case is vertical or horizontal shift of elements.

Similar to ReLU, we define Max Pooling in `Afflang` as a partial function with a compile-time parameter k such that optimizations may be applied to individual applications. The input term $t \in \mathscr{A}$ is interpreted as a flattend vector, which means that the typical grid indices of image data are explicitly converted below through p and l. It contains, as usual, the whole input from which the receptive field must be extracted. The start of the receptive field in t is denoted by (i, j) and has size $F \times F$. The relative internal offset from the start of the receptive field is given by (μ, ν). Similarly to the argmax definition, the index k denotes all elements which have been checked in previous cases (note that the values of k increase continuously and are later converted to the respective internal offset (μ, ν)). The index l denotes the current maximal value.

$$
\begin{aligned}
\texttt{MaxPool}[i, j](t) &= \texttt{MaxPool}[i, j, 2, (i-1)W + j](t) \\
\texttt{MaxPool}[i, j, k, l](t) &= \begin{cases} \texttt{MaxPool}[i, j, k+1, p](t) & \text{if } t[l] - t[p] \leq 0 \wedge k \leq F^2 \\ \texttt{MaxPool}[i, j, k+1, l](t) & \text{if } t[l] - t[p] > 0 \wedge k \leq F^2 \\ t[l] & \text{otherwise} \end{cases} \\
\mu &= \left\lfloor \frac{k-1}{F} \right\rfloor \\
\nu &= (k-1) \mod F \\
p &= (i + \mu - 1)W + j + \nu
\end{aligned}
$$

The values adhere to the following ranges (H height, W width, F kernel size):

$$
\begin{aligned}
1 \leq \mu \leq F \qquad & \qquad 1 \leq i \leq H \\
1 \leq \nu \leq F \qquad & \qquad 1 \leq j \leq W \\
1 \leq k \leq F^2 \qquad &
\end{aligned}
$$

5.2.4 Normalization

Preliminaries (Normalization). Over the last decades, the (technical) realization of *deeper* neural networks was an import stepping stone in increasing their performance. Through their additional depth, it is assumed that networks can learn hierarchical patterns, where each layer contributes to one level of the hierarchy. It was further shown that deeper networks can, in theory, express more complex functions than comparable shallow networks [Goo+13]. At the same time, deeper networks introduce practical problems like numeric stability and gradient stability. For example, as the computation of the gradient of a network requires the application of the chain rule for each layer, deeper networks are more vulnerable to *vanishing* or *exploding gradients*.

Another problem that can occur is *internal covariate shift* where the networks manipulation of the input data through the layers changes their distribution, which can cause problems.

A simple yet effective technique to counteract these trends in deeper networks is to introduce *normalization layers* [Iof15], which normalize the input mini-batches with different methods. For example, batch normalization is defined as [GBC16]:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$
$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2$$
$$y = \gamma\frac{x-\mu}{\sigma} + \beta$$

where variance is here unbiased (Bessel's correction) like in PyTorch and γ, β are learnable parameters. This is typically computed for each component of the input independently. The running index i iterates over the different samples contained in the current mini-batch, hence the name batch normalization.

Normalization layers again fall into the category of functions that behave differently during training and evaluation. In evaluation mode, the mean μ and standard deviation σ are no longer actively computed for each batch, but are precomputed over the training set and therefore fixed. The parameters γ and β are, as all learnable parameters, also fixed during evaluation. Therefore, these layers simplify to affine functions after training has completed.

5.2.5 Dropout

Preliminaries (Dropout). Dropout [Sri+14] is a simple yet effective regularization method that makes the neural network less likely to overfit. It is also the DL equivalent of ensemble methods, as it dynamically removes different parts of the network from training [GBC16]. Its implementation is simple: draw a Bernoulli vector $r \sim \text{Bernoulli}(1-p)$ with probability $1-p$ (so $p \in [0,1]$ is the probability that a neuron is zeroed) and multiply it elementwise with the input.

$$y = x \odot r$$

This operation is completely disabled in evaluation mode and does therefore not require any attention.

5.2.6 Residual Connections

Preliminaries (Residual Connections). Residual or skip connections [He+16] are a simple yet effective tool to improve gradient flow through deeper neural networks. They bypass the instructions of a layer, essentially acting as the identity function, and provide the value after the layer. The final value is then computed by an aggregation function, usually addition, that combines the effect of the layer with the original input. As such, the gradient becomes more stable, as deeper layers are no longer only depending the final value after all previous layers, but also through the skip connections by a much shorter path. In turn, these layers are more open to parameter updates by gradient descent.

In `Afflang`, assuming $t' \in \mathscr{A}^{m \to m}$ is a layer that should be equipped with a skip connection, it can be encoded as ($t \in \mathscr{A}^{n \to m}$):

$$\texttt{skip}(t')(t) = (t' \circ t) + t$$

5.3 Outlook: Recurrent Neural Networks

The family of Recurrent Neural Networks is specialized to handle sequential data like time series or natural language. To adapt to input of variable length, they use a recurrent structure that is unrolled as needed. Data can also flow backwards to allow that new data can be seen in light of previous elements of the sequence. A detailed introduction is given in Section 2.4.3.

5.3.1 Memory Model

While expression terms currently include a lot of duplicate subexpressions, one can simplify them by allowing variable assignments. This in turn complicates the language as one has to store multiple values and also define special variables for the initial input value and for the output vector. It, however, significantly simplifies the definition of memory cells.

Definition 5.3.1 (Memory Extension of `Afflang`*).* In the BNF of `Afflang`, we add a new construct

$$E ::= \cdots \mid \mathscr{V}$$

where $\mathscr{V}$ is the set of variable names. The special variable $\vec{\mathfrak{x}} \in \mathscr{V}$ is *read-only* and always stands for the input vector. Whenever variables from $\mathscr{V}$ appear in an `Afflang` program, we write them in Fraktur, e.g., $\vec{\mathfrak{x}}, \vec{\mathfrak{y}}, \vec{\mathfrak{z}}$. Two new constructs are introduced to `Afflang` for memory management. The assignment is the first statement of `Afflang`, and the expression block allows performing statements as long as they are (eventually) followed by an expression.

$$\begin{aligned} E &::= \cdots \mid \{S\,;E\} \\ S &::= V := E \end{aligned}$$

The semantics are close to the usual definitions of these operators.

$$\frac{\langle t, \sigma\rangle \xrightarrow{w} \langle t', \sigma'\rangle}{\langle x := t, \sigma\rangle \xrightarrow{w} \langle x := t', \sigma'\rangle}$$

$$\frac{-}{\langle x := \vec{v}, \sigma\rangle \xrightarrow{\text{assign}} \langle \epsilon, \sigma[x \mapsto \vec{v}]\rangle}$$

$$\frac{\langle s, \sigma\rangle \xrightarrow{w} \langle s', \sigma'\rangle}{\langle \{s\,;t\}, \sigma\rangle \xrightarrow{w} \langle \{s';t\}, \sigma'\rangle}$$

$$\frac{-}{\langle \{\epsilon\,;t\}, \sigma\rangle \xrightarrow{\text{seq}} \langle t, \sigma\rangle}$$

Notice that an expression block does not introduce a new scope, as we have no variable declarations. For completeness, the typing rules are also extended

$$\frac{\Gamma \vdash t' : m \qquad \Gamma, \{x : m\} \vdash t : n}{\Gamma \vdash \{x := t' \,;\, t\} : n}$$

We may refrain from writing curly braces in nested expression blocks for convenience reasons:

$$\{s_1 \,;\, \{s_2 \,;\, t\}\} \equiv \{s_1 \,;\, s_2 \,;\, t\}$$

5.3.2 Unrolling Recurrences

With this extension we can write simple RNNs naturally. Consider the following recurrence defining an example RNN:

$$h^{(t)} := \mathrm{ReLU}\left(\boldsymbol{U}x^{(t)} + \boldsymbol{W}h^{(t-1)} + \vec{b}\right)$$
$$\hat{y}^{(t)} := \mathrm{softmax}\left(\boldsymbol{V}h^{(t)} + \vec{c}\right)$$

It may be written in `Afflang` as

$$r(i) := \begin{cases} \vec{\mathfrak{h}} := \mathtt{Relu}\left(\boldsymbol{U}\vec{\mathfrak{x}}^{(i)} + \boldsymbol{W}\vec{\mathfrak{h}} + \vec{b}\right) ; \vec{\mathfrak{y}} := \vec{\mathfrak{y}} \otimes (\mathtt{argmax}\ \boldsymbol{V}\vec{\mathfrak{h}} + \vec{c}) \,;\, r(i+1) & \text{if } i \leq t \\ \vec{\mathfrak{y}} & \text{if } i > t \end{cases}$$

The notation can get a bit convoluted as language constructs are mixed with an outer recursive definition that is neither part of the language nor possible to express with language instructions. For that matter, we introduce a new instruction that does not alter the expressiveness of `Afflang` on a language level, but still makes it possible to encode new constructions.

Definition 5.3.2 (For Loop Extension of `Afflang`*).* The syntax of `Afflang` is extended by a finite for loop:

$$S ::= \cdots \mid \mathtt{for}\ i\ \mathtt{in}\ [k\ ..\ n)\ \ S$$

where $k, n \in \mathbb{N}$ with $k < n$ are constant numbers that neither depend on the input nor can be substituted by macros. It is associated with the following semantic rules

$$\frac{k < n}{\langle \mathtt{for}\ i\ \mathtt{in}\ [k\ ..\ n)\ \ s, \sigma\rangle \xrightarrow{\text{unfold}} \langle \{s[i \mapsto k] \,;\, \mathtt{for}\ i\ \mathtt{in}\ [k+1\ ..\ n)\ \ s\}, \sigma\rangle}$$

$$\frac{k \geq n}{\langle \mathtt{for}\ i\ \mathtt{in}\ [k\ ..\ n)\ \ s, \sigma\rangle \xrightarrow{\text{unfold}} \langle \epsilon, \sigma\rangle}$$

At first glance, the definition of the for loop may seem to contradict the expression focus of `Afflang`. However, it actually gives more flexibility when introduced as statement instead of an expression. If one wants to repeat an expression t multiple times, where variable $\vec{\mathfrak{z}}$ is treated as input inside t, it can be written as:

$$\{\mathtt{for}\ i\ \mathtt{in}\ [1\ ..\ n)\ \ \vec{\mathfrak{z}} = t \,;\, \vec{\mathfrak{z}}\}$$

where $\vec{\mathfrak{z}}$ acts as an accumulator. The mentioned flexibility comes from the fact that the user can individually choose what variables act as accumulators throughout the iterations, which would not be the case when the for loop would be implemented as an expression, as then the accumulator would be part of the language definition.

Table 5.1: List of compatible ONNX operators. Legend: (∗) denotes operators that are only supported in evaluation mode, (†) denotes operators where not all ONNX *attributes* are supported. Operators with explicit mentioning of "axis" or "dtype" attributes are marked with (‡).

(a) Linear operators of ONNX

Operator	Supported
Add	✓
BatchNormalization	✓∗†
Concat	✓
Constant	✓‡
Conv	✓†
CumSum	✓‡
Div	✓
Dropout	✓∗
EyeLike	✓‡
Gemm	✓
Identity	✓
MatMul	✓
Mul	✓
Neg	✓
Pad	✓†,‡
ReduceSum	✓‡
Slice	✓‡
Sub	✓
Sum	✓
Transpose	✓

(b) Piecewise linear operators of ONNX

Operator	Supported
Abs	✓
And	✓
ArgMax	✓‡
ArgMin	✓‡
Clip	✓
HardSigmoid	✓
Hardmax	✓‡
If	✓
LSTM	✓†
LeakyRelu	✓
LessOrEqual	✓
Max	✓
MaxPool	✓†
Min	✓
PRelu	✓
RNN	✓†
Relu	✓
ThresholdedRelu	✓
Where	✓†

5.4 Comparison with ONNX

The ONNX language allows the specification of complex dataflow graphs for expressing modern neural network architectures in a modular fashion. The nodes of these graphs are the *ONNX operators* specifying different building blocks used in DNNs like activation functions, matrix multiplication, or even long short-term memory (LSTM) cells. While some operators are simple low level routines which perform one elementary function, like "Mul" or "If", others are complex high level routines, like "RNN" or "LSTM". Notably, these operators are not strictly minimal, as, e.g., "ReLU" can be built from "Max", "Constant", and "CastLike". The edges of the graph define how the output from one operator flows to the input of another. They can be used to model diverging and joining dataflow as, e.g., is part of residual connections or inception modules. Data within ONNX is modeled as multi-dimensional tensors where multiple underlying element types, such as boolean, integer, unsigned integer, and floating-point, are supported.

The edges in an ONNX graph can be represented in `Afflang` using the concatenation operator ⊗, followed by an aggregation function. In many cases, addition is

used for aggregation, in which case the diverging dataflow can be directly modeled in `Afflang` as:

$$t_1 + t_2 + \cdots + t_n$$

Many of the ONNX operators are PWL and can therefore be mapped to `Afflang` based on its expressiveness (Theorem 4.3.1). Futhermore, we have seen the concrete implementation of many ONNX operators in this chapter. Tables 5.1a and 5.1b list ONNX operators compatible with `Afflang`.

Tensors can be implemented in `Afflang`, as was demonstrated in Section 5.2 where CNNs were encoded in `Afflang`. However, to increase readability, most encodings presented in this thesis were deliberately not stated for multi-dimensional tensors, but for one-dimensional vectors as inputs and outputs. Therefore, ONNX operators like "ConstantOfShape", "Cast", or "Flatten" are omitted from the tables, as these were not explicitly discussed here. Consequently, in Table 5.1a the attributes "axis" and "dtype" are not taken into account for the use of (†), as otherwise all operators would be marked with it. For that the symbol (‡) is used. For example, the operator "Conv" also supports setting the dilation of the convolution, which was not discussed in Section 5.2 explicitly, therefore the mark † is applied. On the other hand, the operator "Constant" also supports datatypes like strings, therefore the symbol ‡ is added.

Similarly, operators that can be well-approximated by PWL functions, such as "Swish", are similarly excluded. Depending on the use case, the approximation error of PWL approximations—such as the presented `HardSigmoid`—may be acceptable, or can be acceptable when accompanied by a corresponding error estimation. As this was not discussed here, these operators are not listed.

For "PRelu", the ONNX implementation expects the slope for negative inputs as an additional input argument, sourcing the drawing of a random vector out. Therefore, the operator has the same behavior during training and testing, but the given slope would differ between the two phases as explained in Section 5.1.

CHAPTER 6

Distillation into Surrogate Models

In the previous sections, we have defined a new imperative language that can express PWL functions, especially including PWL DNNs. We extended the core language with syntactic sugar for simplifying reasoning about it and to bring it closer to standard mathematical syntax. In the end, it allowed to define popular activation functions from DL, such as ReLU or HardSigmoid, in a concise and clear way, while at the same time having a unique representation as a code fragment of the language. All these steps were stated constructively, meaning their implementation is straightforward.

In Section 6.1, we will build on that and define symbolic execution for the language. For that, we will show how affine functions and affine predicates can be lifted over the language. After that, Section 6.2 presents advanced analysis methods. Finally, the CFG is rewritten into a decision graph structure (Section 6.3), and algebraic operations are defined over this structure.

6.1 Symbolic Execution

Symbolic Execution [Kin76] is a technique in (formal) program analysis that treats the inputs to a program as variables (*symbols*) and proceeds with an execution that keeps track of all updates to those variables in terms of (symbolic) formulas (dataflow). The control flow of the program in question cannot be directly inferred from the variables, as these correspond to formulas instead of concrete values. Instead, one either explores all possible program paths or uses techniques like SMT solving to determine which paths can possibly be reached. As a result, a graph called control flow graph (CFG) emerges where every (feasible) branch is logged with corresponding necessary conditions (*path conditions*).

Based on the profile of DNNs presented in Section 2.4, a natural representation of a DNN for symbolic execution are affine functions $\vec{x} \mapsto \boldsymbol{W}\vec{x} + \vec{b}$. These are battle-proven in the realm of linear algebra and are both compositional and memory

efficient. The goal of the following calculus is to transform `Afflang` programs (and with that all associated DNNs) into affine functions under boundary conditions. Such conditions are necessary to capture the non-linearity of the activation functions. Many activation functions are PWL, meaning that the boundary conditions are affine predicates. In these cases, also the conditions fall under the compositionality of linear algebra. For a comprehensive introduction to linear algebra see Section 2.1.

6.1.1 Overview

A central building block of this calculus are configurations of the form [Plo04], which extend the previous configurations:

$$\langle \textit{program}, \textit{path condition}, \textit{store} \rangle$$

As in the former model, the *program* $t \in \mathscr{A}_v$ is a (partially evaluated) `Afflang` term. The *path condition* is a new component. It is a formula over affine predicates with negation that characterizes over which inputs the derivation holds true. Finally, the *store* $\sigma \colon \mathscr{V} \to (\mathbb{R}^+ \to \mathbb{R}^+)$ models the effect of the instructions for the given path (as characterized by the path conditions) for each variable. Now it will store affine functions instead of vectors.

In this model, each configuration explores one concrete program path, which is uniquely characterized by its path condition. To describe the whole program, all paths must be explored. As we have discussed in Section 4.3, there are only finitely many paths in `Afflang` programs. This will still be the case for symbolic execution.

6.1.2 Lifting

As already mentioned, for symbolic execution, we need to track the value of variables (in our case subexpressions) symbolically through formulas. To achieve that, we now define all operations, which occur in the SOS rules specifying `Afflang`'s semantics (Definitions 4.2.3, 4.4.1, 5.3.1, and 5.3.2), for affine functions $\alpha \colon \mathbb{R}^+ \to \mathbb{R}^+$ in place of vectors $\vec{v} \in \mathbb{R}^+$. Concretely, every term now evaluates to an affine function. So each occurrence of $\vec{v}$ in the rules is now replaced by an affine function α and all operations (such as matrix multiplication) must be updated to reflect that.

In fact, the transition is straightforward, as one can always *lift* operations on vectors from $\mathbb{R}^m$ to affine functions $\mathbb{R}^n \to \mathbb{R}^m$ [GS21]. The following theorem is an adaptation of [AP I: [Sch+23]]:

Theorem 6.1.1 (Lifting). Any function $g \colon \mathbb{R}^m \times \cdots \times \mathbb{R}^m \to \mathbb{R}^m$ can be *lifted* to $G \colon (\mathbb{R}^n \to \mathbb{R}^m) \times \cdots \times (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to \mathbb{R}^m)$ where the arguments to G are affine functions $\mathbb{R}^n \to \mathbb{R}^m$ such that

$$G(\alpha_1, \ldots, \alpha_k)(\vec{x}) \coloneqq g(\alpha_1(\vec{x}), \ldots, \alpha_k(\vec{x}))$$

Essential for this construction is the concept of higher-order functions [Cur+58]: The lifted function G expects an argument $\vec{x} \in \mathbb{R}^n$ and passes it on to all the α_i functions, which then return an element from $\mathbb{R}^m$. These results are then passed on to g and thus producing the final result from $\mathbb{R}^m$. To be of practical relevance, we must consider how we can represent G, as a literal interpretation of the above definition

would require higher-order functions. For that, let us recall what instructions are part of `Afflang` that actually compute linear algebra terms (instead to the ones that correspond to term rewriting):

- $\boldsymbol{W}\alpha + \vec{c}$
- $\alpha \leq \vec{b}$
- $\alpha_1 + \alpha_2$

Where α, α_1, and α_2 are affine functions, $\boldsymbol{W}$ is a type compatible matrix, and $\vec{c}$ and $\vec{b}$ are type compatible vectors. The account of the first and last instructions was already discussed in the preliminaries (Theorem 2.1.11) where it was shown that the resulting affine function can be computed independently from the input value. In the second case, a new affine predicate is formed:

$$(\vec{x} \mapsto \boldsymbol{W}\vec{x} + \vec{b}) \leq \vec{c} \;\equiv\; \boldsymbol{W}\mathbf{u} \leq (\vec{c} - \vec{b})$$

Composition To avoid repeating too many rules, we now define a composition rule for subterms. The desired semantics should be clear from the previous definitions: We evaluate the subterms in the order that they appear in the term. For that, we define the *first reducible subexpression* as follows:

Definition 6.1.2 (Fredux). A *fredux* is the first reducible subexpression of a term, if it exists. For `Afflang`, we define ($t \in \mathscr{A}_v$, $\vec{v} \in \mathbb{R}^+$):

$$\begin{aligned}
\mathrm{Fredux}(\boldsymbol{M}t + \vec{c}) &= \{t\} \\
\mathrm{Fredux}(\boldsymbol{M}\vec{v} + \vec{c}) &= \{\} \\
\mathrm{Fredux}(\vec{\mathbf{x}}) &= \{\} \\
\mathrm{Fredux}(t_1 + t_2) &= \{t_1\} \\
\mathrm{Fredux}(\vec{v}_1 + t_2) &= \{t_2\} \\
\mathrm{Fredux}(\vec{v}_1 + \vec{v}_2) &= \{\} \\
\mathrm{Fredux}(\texttt{if } (t \leq \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2) &= \{t\} \\
\mathrm{Fredux}(\texttt{if } (\vec{v} \leq \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2) &= \{\} \\
\mathrm{Fredux}(\{s\,;t\}) &= \{s\} \\
\mathrm{Fredux}(\{\epsilon\,;t\}) &= \{\} \\
\mathrm{Fredux}(\vec{\mathbf{x}} := t) &= \{t\} \\
\mathrm{Fredux}(\vec{\mathbf{x}} := \vec{v}) &= \{\}
\end{aligned}$$

Then composition can be stated as

$$\frac{\langle t', \pi, \sigma\rangle \xrightarrow{w} \langle t'', \pi', \sigma'\rangle \qquad t' \in \mathrm{Fredux}(t)}{\langle t, \pi, \sigma\rangle \xrightarrow{w} \langle t[t''/t'], \pi', \sigma'\rangle}$$

where $[t''/t']$ denotes, as usual, the substitution of t'' in place of t' (see [Plo04]). Since this rule might be applied again in its own premise, we could also directly

allow the reduction of any subterm that is in the transitive closure of the fredux relation. Concretely, the closure Fredux* is made of the elements:

$$t' \in \text{Fredux}^*(t) \implies \text{Fredux}(t') \subseteq \text{Fredux}^*(t)$$

Now we use the meta symbol α for a completely evaluated term instead of $\vec{v}$ in order to remind ourselves that in symbolic execution, all terms evaluate to affine functions.

$$\frac{\alpha' = \boldsymbol{W}\alpha + \vec{c}}{\langle \boldsymbol{W}\alpha + \vec{c}, \pi, \sigma\rangle \xrightarrow{\text{lin}} \langle \alpha', \pi, \sigma\rangle}$$

$$\langle \texttt{if } (\alpha \leq \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2, \pi, \sigma\rangle \xrightarrow{\text{if}_t} \langle s_1, \pi \wedge \alpha \leq \vec{b}, \sigma\rangle$$

$$\langle \texttt{if } (\alpha \leq \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2, \pi, \sigma\rangle \xrightarrow{\text{if}_f} \langle s_2, \pi \wedge \neg(\alpha \leq \vec{b}), \sigma\rangle$$

$$\langle \vec{\mathbf{x}}, \pi, \sigma\rangle \xrightarrow{\text{mem}} \langle \sigma(\vec{\mathbf{x}}), \pi, \sigma\rangle$$

$$\frac{\alpha_3 = \alpha_1 + \alpha_2}{\langle \alpha_1 + \alpha_2, \pi, \sigma\rangle \xrightarrow{\text{add}} \langle \alpha_3, \pi, \sigma\rangle}$$

$$\langle x := \alpha, \pi, \sigma\rangle \xrightarrow{\text{assign}} \langle \epsilon, \pi, \sigma[x \mapsto \alpha]\rangle$$

$$\langle \{\epsilon\ ;\ t\}, \pi, \sigma\rangle \xrightarrow{\text{seq}} \langle t, \pi, \sigma\rangle$$

Let us take a look at why only the if rule changes path conditions. For the *mem* rule, we only access the result of a computation that has already taken place, therefore it does not introduce any branches, even when the term whose result is stored in the variable does branch. In the *add* rule, one might expect that the path conditions must change, as adding two PWL functions can result in a PWL function where the number of linear regions is the product of the number of regions of the addends. This is implicit in the way we evaluate the sum: both terms are evaluated before their sum is taken. So each path corresponds to the conjunction of one of the path conditions of the first addend with one of the second. So the total number of paths and configurations, respectively, is indeed the product of the paths of each addend. For *assign*, the path associated with the term that is assigned to the variable was again first evaluated, so all respective conditions were already added to π.

To summarize, the path conditions track the internal state of the program and not the observable state from outside. Therefore, whenever any term is evaluated that has more than one linear region, the configurations will reflect this and also diverge. Even before the result of the computation is ever used. For some cases, this might be finer than necessary, e.g., when a variable is dead.

Example 6.1.3. Now we briefly apply these rules to the partial ReLU term for the k-th neuron.

$$\langle \texttt{if } (\vec{e_k}^{\mathsf{T}}\vec{\mathbf{x}} \leq \vec{0}) \texttt{ then } \boldsymbol{R_k}(0)\vec{\mathbf{x}} \texttt{ else } \vec{\mathbf{x}}, \top, \vec{\mapsto}\,\text{id}\rangle$$

The two derivations for this configuration are shown in Figure 6.1.

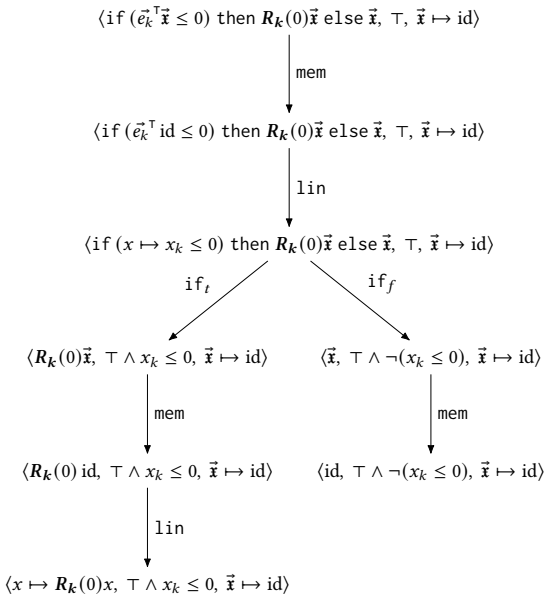


Figure 6.1: Two derivations for `Relu` using the symbolic SOS rules. The shared prefix of both derivations is aggregated into a tree.

6.1.3 Lifting of Tensor Operations

For convolution, input data is represented as tensors. The principles of lifting generalize also to tensors in form of multiaffine functions, which will be described in the following. Since the inputs and outputs of multichannel convolution are typically order three tensors, the lifting now needs to be a (multi-)affine function with type $\mathbb{R}^{C_{in} \times H \times W} \longrightarrow \mathbb{R}^{C_{out} \times H_{out} \times W_{out}}$. Although the notation will require considerably more indices (a total of six), the core idea is the same as for the previous cases. There, a 1D input vector is multiplied by a matrix and shifted by a bias vector, which is typically denoted as:

$$\vec{y} = \boldsymbol{W}\vec{x} + \vec{b}$$

Another view would be that each output value is the result of a linear combination of the input with a set of coefficients stored in $\boldsymbol{W}$:

$$y_i = w_{i,1}x_1 + \cdots + w_{i,n}x_n + b_i$$

This idea generalizes to higher dimensions more easily. For each position (β, μ, ν) of the output $\boldsymbol{Y}$ of the affine function, a linear combination is made with the order three input tensor $\boldsymbol{X}$ and coefficients from some tensor $\boldsymbol{W}$:

$$f\colon\ y^{\beta,\mu,\nu} = \sum_{k=1}^{C_{in}} \sum_{i=1}^{H} \sum_{j=1}^{W} w_{k,i,j}^{\beta,\mu,\nu} x^{k,i,j} + b^{\beta,\mu,\nu}$$

where $\beta \in [C_{out}]$, $\mu \in [H_{out}]$, and $\nu \in [W_{out}]$. Now that the output tensor $\boldsymbol{Y}$ is determined, it is beneficial to change the view from each element of $\boldsymbol{Y}$ to slices: for each output channel β it is an image of shape $H_{out} \times W_{out}$, which may be notated as

$$\boldsymbol{y}^{\beta,:,:}$$

and for such two-dimensional images we defined (strided) convolution.

$$\begin{aligned} \boldsymbol{z}^{\boldsymbol{\alpha},:,:} &= \left(\mathsf{Conv}(k^{\alpha,\beta,\mu,\nu}, s) \circ f\right)^{\alpha,:,:} \\ &= \sum_{\beta=1}^{C_{out}} \left(\boldsymbol{y}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right) \end{aligned}$$

Now we plug in the definition of $\boldsymbol{Y}$, which is the result of applying the multiaffine function f to $\boldsymbol{X}$

$$\boldsymbol{z}^{\boldsymbol{\alpha},:,:} = \sum_{\beta=1}^{C_{out}} \left(\left(\sum_{k=1}^{C_{in}} \sum_{i=1}^{H} \sum_{j=1}^{W} \boldsymbol{w}_{\boldsymbol{k,i,j}}^{\boldsymbol{\beta},:,:} x^{k,i,j} + \boldsymbol{b}^{\boldsymbol{\beta},:,:}\right) \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right)$$

As convolution is linear, it distributes over linear combinations.

$$\boldsymbol{z}^{\boldsymbol{\alpha},:,:} = \sum_{\beta=1}^{C_{out}} \sum_{k=1}^{C_{in}} \sum_{i=1}^{H} \sum_{j=1}^{W} \left(\left(x^{k,i,j} \boldsymbol{w}_{\boldsymbol{k,i,j}}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right) + \left(\boldsymbol{b}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right)\right)$$

Now, each $x^{k,i,j} \in \mathbb{R}$ is a scalar (in contrast to $\boldsymbol{w}_{\boldsymbol{k,i,j}}^{\boldsymbol{\beta},:,:}$ and $\boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}$, which are order two tensors) and can be factored out.

$$\boldsymbol{z}^{\boldsymbol{\alpha},:,:} = \sum_{\beta=1}^{C_{out}} \sum_{k=1}^{C_{in}} \sum_{i=1}^{H} \sum_{j=1}^{W} \left(x^{k,i,j} \left(\boldsymbol{w}_{\boldsymbol{k,i,j}}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right) + \left(\boldsymbol{b}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right)\right)$$

We can express the convolution of a multiaffine function as convolutions over the weights of the function by changing the order of the summations

$$\begin{aligned} \boldsymbol{z}^{\boldsymbol{\alpha},:,:} &= \sum_{k=1}^{C_{in}} \sum_{i=1}^{H} \sum_{j=1}^{W} \left(x^{k,i,j} \underbrace{\left(\sum_{\beta=1}^{C_{out}} \boldsymbol{w}_{\boldsymbol{k,i,j}}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}\right)}_{=u_{k,i,j}^{\alpha,:,:}} + \underbrace{\sum_{\beta=1}^{C_{out}} \boldsymbol{b}^{\boldsymbol{\beta},:,:} \star_s \boldsymbol{k}^{\boldsymbol{\alpha},\boldsymbol{\beta},:,:}}_{=c^{\alpha,:,:}}\right) \\ &= \sum_{k=1}^{C_{in}} \sum_{i=1}^{H} \sum_{j=1}^{W} \left(\boldsymbol{u}_{\boldsymbol{k,i,j}}^{\boldsymbol{\alpha},:,:} x^{k,i,j} + \boldsymbol{c}^{\boldsymbol{\alpha},:,:}\right) \end{aligned}$$

Where $\boldsymbol{u}_{\boldsymbol{k,i,j}}^{\boldsymbol{\alpha},:,:}$ is the updated weight tensor and $\boldsymbol{c}^{\boldsymbol{\alpha},:,:}$ the updated bias term of the multiaffine function.

6.2 Advanced Analysis Methods

In this section, we extend the syntax of `Afflang` again to incorporate new paradigms suitable for expressing analysis tasks. `Afflang` already contains all necessary building blocks to fix some inputs to concrete constant values while others stay symbolic, thus enabling *concolic execution* [GKS05]. Preconditions will be implemented in `Afflang` by adding support for expressing partial functions, which enables the restriction to inputs that satisfy the precondition. As a result, it will be possible to append or prepend certain code blocks to an existing `Afflang` program to change

its semantics to reflect a given analysis while powered by the previously defined symbolic execution.

Originally, these concepts were presented in [AP II: [Nol+23]] (concolic mode was later covered more explicitly in [AP V: [SS24]]). In these works, the ideas were directly defined for `AffTrees`, which are here introduced in Section 6.3. In the following, both techniques will be introduced for `Afflang`.

6.2.1 Concolic Execution

While symbolic execution is in principle an effective tool to generate abstract graphs that capture all possible execution paths of the program, which can be used to decide semantic properties, its scalability is limited due to a phenomenon called *path explosion*. Even in the case of `Afflang`, we have seen that the number of paths through the program is exponential in the program size. As such, unconstrained symbolic execution can quickly become practically infeasible. To mitigate this fact, *concolic execution* was introduced [GKS05; SMA05; Cad+08], which combines concrete and symbolic execution to achieve a best-of-both-worlds solution. In this analysis technique, parts of the input are fixed while others are treated symbolically. The constant parts correspond to a normal (concrete) execution of the program, so branching behavior only depending on these values can be fully computed, and only one path has to be explored. On the other hand, all branches including the value of at least one symbolic variable, are treated as in symbolic execution. Depending on the ratio between fixed and symbolic variables as well as their respective importance in the program, concolic execution can achieve significantly better performance, as fewer branches have to be explored.[1]

For `Afflang`, concolic execution can benefit from linear algebra concepts like subspaces and orthogonality, as we will now illustrate. Starting simple, suppose we have an `Afflang` program $t \in \mathscr{A}^{4\to 2}$ and a concrete input vector $(c_1, c_2, c_3, c_4)^\intercal \in \mathbb{R}^4$. If we want to use its second and third element and let the first and fourth be symbolic, i.e.,

$$\hat{c} = \begin{pmatrix} X_1 \\ c_2 \\ c_3 \\ X_4 \end{pmatrix}$$

with symbolic variables X_1 and X_4, then we can simply append a corresponding affine function to t. This function $f\colon \mathbb{R}^2 \to \mathbb{R}^4$ is given by

$$f(\vec{x}) = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}}_{=\boldsymbol{W}} \cdot \vec{x} + \underbrace{\begin{pmatrix} 0 \\ c_2 \\ c_3 \\ 0 \end{pmatrix}}_{=\vec{b}} = \begin{pmatrix} x_1 \\ c_2 \\ c_3 \\ x_2 \end{pmatrix} .$$

[1] In general, another benefit is also that the representation of the symbolic variables, which are SMT formulas, can be shorter and can therefore result in faster solving. Specifically in `Afflang`, the SMT formulas are actually linear programs and the effect of concolic execution is that the programs contain fewer variables and inequalities, which positively affects the solving time.

The Afflang program $(\boldsymbol{W}t + \vec{b}) \in \mathscr{A}^{2\to 2}$ then already implements the desired concolic execution. The forwarding and simplifying of constants happens naturally through the previously defined semantics. More generally, when we have a vector $\vec{c} \in \mathbb{R}^n$ of observations and want to keep the indices $I \subseteq [n]$, then we first define a mask

$$m_i = \begin{cases} 1 & \text{if } i \in I \\ 0 & \text{otherwise} \end{cases}$$

and define its negation as $\bar{m}_i = 1 - m_i$. Then the desired values for the Afflang expression

$$\boldsymbol{W}t + \vec{b}$$

are $\vec{b} = m \odot c$ and $\boldsymbol{W} = \bar{m} \odot \boldsymbol{I}_n$ where zero columns have been removed.[2]

While this approach fits tabular data well, where each component (feature) of the input vector has a clear and distinct interpretation, it can be improved for other input data like pixels in images. Here, we may also analyze what happens when we change multiple pixels in a dependent way, for example, if we evenly increase the brightness of the input image. Here again the theory of linear algebra brings suitable tools, for example, with the concept of linear independence, rank, and orthogonality.

Assume we are interested in all inputs of an r-dimensional (affine) subspace $U \subseteq \mathbb{R}^n$ spanned by an orthonormal basis $\vec{u}_1, \ldots, \vec{u}_r$ and an origin $\vec{o}$ (using affine coordinates). That means we have coupled the inputs from $\mathbb{R}^n$ in such a way that only r degrees of freedom remain. Then we can define

$$f(\vec{x}) = \begin{bmatrix} \vec{u}_1 & \cdots & \vec{u}_r \end{bmatrix} \cdot \vec{x} \; + \; \vec{y}$$

where $\vec{y} \in U$ is the unique vector with the shortest distance to U

$$\vec{y} = \operatorname*{arg\,min}_{\vec{x} \in U} \|\vec{x}\| \quad .$$

This mapping $f \colon \mathbb{R}^r \to \mathbb{R}^n$ is affine, injective, distance-preserving, and its image is equal to U. Thus, when we prepend f to an Afflang term t, we essentially restrict t to inputs from U.

Although it is not strictly necessary that the basis $\vec{u}_1, \ldots, \vec{u}_r$ is orthonormal and that $\vec{y}$ has the shortest distance to U, it (i) makes the representation as unique as possible and (ii) results in desirable properties of the mapping, like preserving distances. Also, every basis can be converted to an orthonormal basis with Gram-Schmidt and $\vec{y}$ can always be minimized, so these restrictions can always be satisfied.

Example 6.2.1 (PCA). A general approach for reducing the dimensionality and with that the complexity of the verification task is Principal Component Analysis (PCA). PCA is a well-known technique for reducing the dimensionality of input signals through a linear transformation. More importantly for our approach, PCA is an orthogonal transformation, and can therefore be used as discussed above. Let the row vectors of $\boldsymbol{U}$ be the principal components of the dataset, then $\boldsymbol{U}^\intercal\boldsymbol{U}$ is an orthogonal projection matrix. For dimensionality reduction, typically only the first

[2] In the context of analyzing neural networks, it is especially interesting when $\vec{c}$ is realistic, e.g., is in-distribution.

$k \ll n$ rows of $\boldsymbol{U}$ are considered, giving a non-square matrix $\boldsymbol{U_k} \in \mathbb{R}^{k \times n}$. Through its projection matrix $\boldsymbol{U_k^\intercal U_k}$ it gives rise to the subspace

$$\begin{aligned} U_k &= \left\{ \boldsymbol{U_k^\intercal U_k} \vec{x} \;\middle|\; \vec{x} \in \mathbb{R}^n \right\} \\ &= \left\{ \boldsymbol{U_k^\intercal} \vec{x} \;\middle|\; \vec{x} \in \mathbb{R}^k \right\} \qquad \subseteq \mathbb{R}^n \end{aligned}$$

This linear subspace should be shifted by the mean μ, which results in the affine subspace $U_k + \mu$. To summarize, with this approach we can create a lower-dimensional (affine) subspace $U_k + \mu$ that is embedded in the full input space $\mathbb{R}^n$. By construction, this subspace includes the most important directions of the data, i.e., the principal component axes sorted by explained variance. On the other hand, as it is a lower-dimensional subspace, appending the affine projection into the space to an `Afflang` program realizes concolic execution, which is faster than full symbolic execution. That is, we focus computational resources on the most promising input regions. An analysis of DNN robustness with this approach was presented in [AP II: [Nol+23]].

6.2.2 Preconditions

Preconditions are very important in the verification of DNNs, as their input space is almost always unstructured (*subsymbolic*): In contrast to previous machine learning (ML) approaches, DNNs no longer receive features (like human concepts) as inputs, but instead high-dimensional, raw, sensor-like data. For example, all $H \cdot W$ pixels of an image or n-grams of a document of text. This results in two challenges for verification of universally quantified formulas over the whole input space:

(i) The first problem is that the input space $\mathbb{R}^n$ is unbounded,[3] which makes no sense for most application domains. For example, in case of images, the colors are typically in the range $[0, 255]$, which are somewhat calibrated at human perception.

(ii) The input space includes representations that are ill-formed, for example, an image consisting solely of noise. Even a completely white image cannot meaningfully represent an object for classification.[4] It is not hard to imagine that one can find counterexamples to almost any semantic property of DNNs in these ill-formed inputs. Excluding them in analyses is therefore important.

In ML, it is generally believed that proper input data lies on a much lower-dimensional *manifold* embedded in the complete input space [BCV13]. Adding noise to a valid input image is seen as an orthogonal transformation with respect to the data manifold.[5] This aspect can be formalized with measure theory. This concludes the motivation for precondition-based verification of DNNs, now we define it formally.

Definition 6.2.2 (Precondition Verification). Let $Q_1, \ldots, Q_m$ be a set of bounded, non-overlapping, and convex polytopes $Q_i \subseteq \mathbb{R}^n$ and let $\mathcal{Q} = \cup_{i=1}^m Q_i$ denote their union,

[3] We may consider the bounds of their representation on GPUs, which typically is a 16-bit or 32-bit float (half-precision and single-precision IEEE 754). The former has a maximum value of 65504, the latter of $\sim 10^{38}$.

[4] That is not to say that such an image cannot appear in real life applications (sensor bugs, imaging a strong light source, etc.). Nevertheless, without further information, the image itself cannot be mapped to a distinct physical object.

[5] More precisely, it is a vector field that points towards the nearest point on the data manifold.

then precondition-based verification of a DNN $\mathscr{N}$ is the process of checking whether the formula

$$\forall \vec{x} \Big(\vec{x} \in \mathscr{Q} \implies \varphi\big(\vec{x}, \mathscr{N}(\vec{x})\big) \Big)$$

holds for some semantic property φ.

Checking the property can be decomposed into m separate tasks ($i \in [m]$):

$$\forall \vec{x} \Big(\vec{x} \in Q_i \implies \varphi\big(\vec{x}, \mathscr{N}(\vec{x})\big) \Big)$$

Now the question remains on how the regions are chosen. For some analysis tasks, the regions are based on the ℓ_1 or ℓ_∞ norm. Alternatively, one can choose the regions Q_i such that they represent the most likely data using either analytical or statistical methods. Suppose the input space has a known probability distribution with density function f and probability measure P. For choosing the distinct regions Q_i, one can use the measure P:

$$P(\mathscr{Q}) = \sum_{i=1}^{m} P(Q_i)$$

Generally, sets with higher probability density would be preferable, which could be derived from level sets of the corresponding density function f, i.e.,

$$\mathscr{Q} = \{\, \vec{x} \in \mathbb{R}^n \mid f(x) \geq c \,\}$$

for some constant $c \in \mathbb{R}$. If that is not feasible, another approach could be minimizing the Lebesgue measure μ of $\mathscr{Q}$:

$$\min \mu(\mathscr{Q}) \quad \text{such that} \quad P(\mathscr{Q}) = p$$

for some given constant $p \in (0, 1)$. Note that the connected components of $\mathscr{Q}$ are not necessarily polyhedral in this formulation.

This leads to another extension of `Afflang` for enabling the encoding of partial functions $\mathbb{R}^n \hookrightarrow \mathbb{R}^m$ that are not defined for every input point.

Definition 6.2.3 (Partial Evaluation Extension of `Afflang`*).* `Afflang` terms are extended by a new symbol $\perp\!\!\!\perp$ (syntax of bottom) that is an expression:

$$E ::= \cdots \mid \perp\!\!\!\perp$$

It symbolizes that the program is *undefined* for the given input. On a semantic level, it translates to the symbol $\bot$ (semantics of bottom), which we add to $\mathscr{A}_v$:

$$\langle \perp\!\!\!\perp, \pi, \sigma \rangle \xrightarrow{\text{undef}} \langle \bot, \bot, \sigma \rangle$$

Here, $\bot$ in the path condition refers to logic false. When $\bot$ appears as a subterm, it acts as a persistent state, i.e., applying any instruction to it results again in the value $\bot$:

$$\langle \boldsymbol{W} \bot + \vec{c}, \pi, \sigma \rangle \xrightarrow{\text{lin}_\bot} \langle \bot, \pi, \sigma \rangle$$

$$\langle \texttt{if}\ (\bot \leq \vec{b})\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \pi, \sigma \rangle \xrightarrow{\text{if}_\bot} \langle \bot, \pi, \sigma \rangle$$

$$\langle \bot + t, \pi, \sigma \rangle \xrightarrow{\text{add}_\bot} \langle \bot, \pi, \sigma \rangle$$

$$\langle \alpha + \bot, \pi, \sigma \rangle \xrightarrow{\text{add}_\bot} \langle \bot, \pi, \sigma \rangle$$

While these are the rules for symbolic execution, the rules for normal execution are identical after removing the path condition π from each configuration.

This extension allows us to prepend a convex precondition $Q \subseteq \mathbb{R}^n$ to an `Afflang` program $t \in \mathscr{A}^{n \to m}$ in the following way. First, define Q in terms of an affine predicate $\boldsymbol{A}\mathbf{u} \leq \vec{b}$. Then the precondition can be applied to t as

$$
\begin{aligned}
&\texttt{if } (A_{1,:}\vec{\mathbf{x}} \leq b_1) \texttt{ then} \\
&\texttt{if } (A_{2,:}\vec{\mathbf{x}} \leq b_2) \texttt{ then} \\
&\quad \vdots \\
&\texttt{if } (A_{q,:}\vec{\mathbf{x}} \leq b_q) \texttt{ then } t \\
&\texttt{else } \perp\!\!\!\perp \texttt{ else } \perp\!\!\!\perp \cdots \texttt{else } \perp\!\!\!\perp
\end{aligned}
$$

This analysis method was used in [AP II: [Nol+23]] to analyze the robustness of DNNs. There, preconditions were directly defined for `AffTrees`, while here they are defined for the intermediate `Afflang` language.

6.2.3 Postconditions

Similar to preconditions, `Afflang` already provides the facilities to apply any PWL postcondition g to an `Afflang` term $t \in \mathscr{A}^{n \to m}$ simply by composing it:

$$g \circ t \ .$$

As demonstrated in the previous sections, g can also be a partial function or represent a polyhedral set $Q \subseteq \mathbb{R}^n$ in the form of an indicator function:

$$g(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in Q \\ 0 & \text{else} \end{cases} \ .$$

In the second case, the corresponding `Afflang` term would be

$$
\begin{aligned}
&\texttt{if } (A_{1,:}\vec{\mathbf{x}} \leq b_1) \texttt{ then} \\
&\texttt{if } (A_{2,:}\vec{\mathbf{x}} \leq b_2) \texttt{ then} \\
&\quad \vdots \\
&\texttt{if } (A_{q,:}\vec{\mathbf{x}} \leq b_q) \texttt{ then } 1 \\
&\texttt{else } 0 \texttt{ else } 0 \cdots \texttt{else } 0
\end{aligned}
$$

where $Q = \text{Models}(\boldsymbol{A}\mathbf{u} \leq \vec{b})$. In case the original input is required for verifying the postcondition, one can add a skip connection around t and aggregate the two paths with concatenation:

$$g \circ (\boldsymbol{I} \otimes t)$$

6.3 Faithful Surrogate Trees

AffTrees or TADS [AP I: [Sch+23]], [AP II: [Nol+23]], [AP IV: [SN23]] are a decision tree structure specialized to represent PWL functions. They have a simple structure: all decision nodes are affine predicates and all terminal nodes are affine functions. They also inherit many algebraic properties and compositionality from algebraic decision diagrams (ADDs). In the following, we will formally define AffTrees before we proceed with the transformation from Afflang to AffTrees. The following definition is based on [AP IV: [SN23]]:

*Definition 6.3.1 (*AffTree*).* An AffTree $T = (N, \rightarrow, \zeta)$ 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 affine predicates $q = \langle \vec{w}, \mathfrak{u} \rangle + b \leq 0$. Decision nodes have two successors, one if the condition is satisfied and one if not.

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

All affine functions α and affine predicates q in an AffTree 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. We define the set of all AffTrees with input dimension n and output dimension m as $\mathscr{T}^{n \to m}$.

AffTrees are sequentially evaluated like decision trees [AP IV: [SN23]]:

*Definition 6.3.2 (*AffTree *Evaluation).* The semantic functional of AffTrees

$$[\![\cdot]\!] \colon \mathscr{T}^{n \to m} \to (\mathbb{R}^n \to \mathbb{R}^m)$$

is inductively defined as

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

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

AffTrees are a symbolic representation of a piecewise 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 in an AffTree corresponds to one linear region of the network. At the end of each path lies a terminal node that stores the respective affine function of the region.

6.3.1 Graph Rewritting

To define the CFG for our language, we need to consider nine rules. Out of these nine, only two 'if-then-else' rules branch. For the remaining seven rules, the control flow is simple, as four correspond to subterm evaluation, two of these just correspond to arithmetic computations, and one to value lookup. For a given `Afflang` term $t \in \mathscr{A}^{n\to m}$, the initial configuration for symbolic execution is $\langle t, \top, \vec{\mathbf{x}} \mapsto \mathrm{id}\rangle$. The full CFG can be defined recursively over the observable symbols Λ as follows:

Definition 6.3.3. The transition system defined by the SOS derivation operator $\to : \Gamma \times \Lambda \to \Gamma$ from `Afflang` can be rewritten into a decision tree by the function

$$v \colon \Gamma \to \mathscr{T}$$

recursively defined over a configuration $c = \langle t, p \wedge \pi, \sigma\rangle$ as ($w \in \Lambda \setminus \{\mathrm{if}_t, \mathrm{if}_f\}$):

$$\begin{aligned} c \xrightarrow{w} c' &\implies v(c) = v(c') \\ c \xrightarrow{\mathrm{if}_t} c_t \wedge c \xrightarrow{\mathrm{if}_f} c_f &\implies v(c) = (\pi, v(c_t), v(c_f)) \\ c = \langle \vec{v}, \pi, \sigma\rangle \not\to c' &\implies v(c) = \vec{v} \end{aligned}$$

For an `Afflang` term $t \in \mathscr{A}^{n\to m}$ we define the shorthand

$$v(t) = v(\langle t, \top, \vec{\mathbf{x}} \mapsto \mathrm{id}\rangle)$$

From this definition, we can clearly see that v descents along the relation $\to$, which terminates. Further, we know that derivation sequences from $\to$ are unique, and thus v is well-defined.

Lemma 6.3.4. For any `Afflang` term $t \in \mathscr{A}$, the corresponding `AffTree` $v(t)$ exists and is uniquely determined.

An alternative way of defining the CFG is based on the existing path conditions inside the configurations. We can organize all path conditions into a tree $(N, E, \top)$:

$$\begin{aligned} c \to^* \langle t, \pi, \sigma\rangle &\implies \pi \in N \\ \phi \in N \wedge (\phi \wedge \neg\pi) \in N &\implies (\phi, 0, \phi \wedge \neg\pi) \in E \\ \phi \in N \wedge (\phi \wedge \pi) \in N &\implies (\phi, 1, \phi \wedge \pi) \in E \end{aligned}$$

To see that this leads indeed to a tree, notice that path conditions increase monotonously throughout all nine rules, i.e., whenever the path condition is changed it is by adding a new conjunction clause to the existing path condition. Further, each state of N keeps an exact log of all its preceding states and is therefore unique. Finally, to simplify reasoning over the tree, we rename the states after the next branch condition through the following bijection:

$$f(\phi) = \begin{cases} (\pi, \phi) & \text{if } (\phi, 1, \phi \wedge \pi) \in E \\ (\vec{v}, \phi) & \text{if } \langle \vec{v}, \phi, \sigma\rangle \text{ is terminal} \end{cases}$$

For a precise definition, we may keep the original name as the second element of the pair, but only visualize the first one. Otherwise, nodes with the same name might be identified, which could impact the graph structure.

Size. The length of any execution path (with respect to the CFG) is clearly only influenced by if-then-else expressions, has can be seen in Definition 6.3.3. And since a path can only explore either branch of an if-then-else expression, it follows that its length is determined by the number of (nested) if-then-else expressions along its path. Formalized, this leads to the following inductively defined function height: $\mathscr{A} \to \mathbb{N}$:

$$\begin{aligned}
\text{height}(\vec{\mathbf{x}}) &:= 0 \\
\text{height}(\boldsymbol{W}t + \vec{c}) &:= \text{height}(t) \\
\text{height}(\texttt{if } (t \leq \vec{b}) \texttt{ then } s_1 \texttt{ else } s_2) &:= \text{height}(t) + \max\{\text{height}(s_1), \text{height}(s_2)\} + 1 \\
\text{height}(t_1 + t_2) &:= \text{height}(t_1) + \text{height}(t_2) \\
\text{height}(\vec{\mathbf{x}} := t) &:= \text{height}(t) \\
\text{height}(\{s\,;\,t\}) &:= \text{height}(s) + \text{height}(t)
\end{aligned}$$

The depth of the CFG is therefore most influenced by *if*, *add*, and *seq* instructions. When only the first three instructions are used, which are already sufficient to express all PWL functions, the depth of the CFG is smaller than the length of its program, a very interesting and unusual property.

Lemma 6.3.5. For any `Afflang` term $t \in \mathscr{A}$, it holds that for any valid derivation sequence ($w \in \Lambda^+$):

$$\langle t, \top, \vec{\mathbf{x}} \mapsto \text{id} \rangle \overset{w}{\Longrightarrow} \langle t', \pi, \sigma \rangle$$

the path condition π has at most $\text{height}(t)$ many conjunction clauses.

Corollary 6.3.6. For any `Afflang` term $t \in \mathscr{A}$, the size of the corresponding `AffTree` is upper bounded by

$$\begin{aligned}
|\nu(t)|_{\text{terminals}} &\leq 2^{\text{height}(t)} \\
|\nu(t)|_{\text{nodes}} &\leq 2^{\text{height}(t)+1} - 1
\end{aligned}$$

Decidability and Complexity. In the `AffTree`, all path conditions are affine predicates of the form $\boldsymbol{A}\mathbf{u} \leq \vec{b}$. Thus, the path conditions are of a simpler kind than those of typical program analysis. To check whether such a formula is *satisfiable* (respectively if the path is *feasible*), it is sufficient to use LP solving methods, such as the simplex algorithm or the interior points method. As a result, the problem of path feasibility is decidable in polynomial time for `AffTrees`.

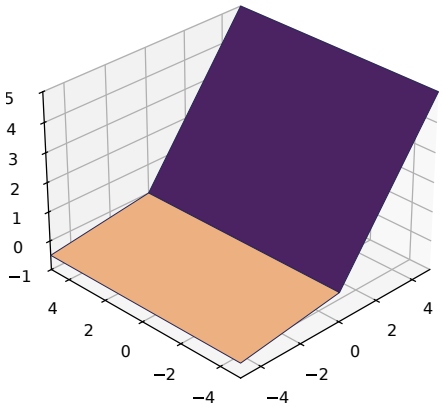
This leads to the following rather comfortable result:

Theorem 6.3.7. Let φ be a property that is decidable for a convex polytope $Q \subseteq \mathbb{R}^d$ in time $T(r, d)$ where d is the dimension of the input space and r the number of constraints of the input polytope. Let $t \in \mathscr{A}^{n \to m}$ be an `Afflang` term. Then one can decide the following implication

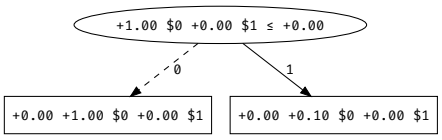
$$\forall \vec{x}\, (\vec{x} \in C \implies \varphi(\vec{x}, t(\vec{x})))$$

with the worst-case runtime

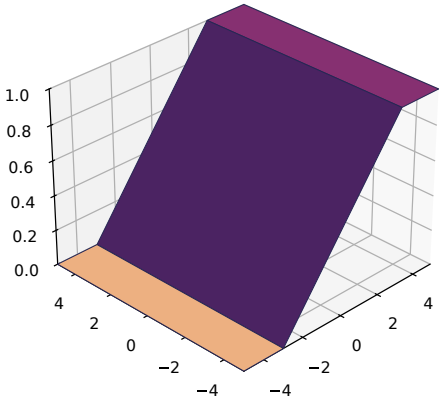
$$\mathscr{O}\left(T(q + h, n) \cdot 2^h\right)$$



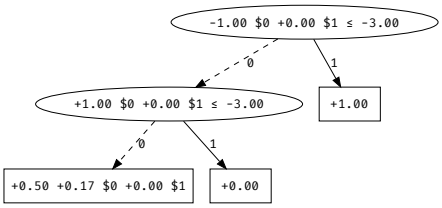
(a) Plot of (partial) Leaky ReLU



(b) `AffTree` of (partial) Leaky ReLU



(c) Plot of (partial) Hard Sigmoid



(d) `AffTree` of (partial) Hard Sigmoid

Figure 6.2: Plots and `AffTrees` of PWL activation functions. The `AffTree` is a direct result of applying ν on the `Afflang` encodings of the respective function.

where C is a convex polytope with q constraints and $h = \text{height}(t)$ is the maximal depth of the `AffTree` of t.

Finally, components of neural networks can be distilled into equivalent `AffTrees` through ν. For the case of the activation functions Leaky ReLU and Hard Sigmoid, an example is shown in Figure 6.2. One can see that the `AffTrees` show the respective function in a concise manner.

6.3.2 Algebraic Properties

`AffTrees` have a rich algebraic structure, which they inherit from PWL functions. Indeed, as each path characterizes one linear region, the extensive theory of linear algebra can be leveraged in many ways.

Algorithmically, one general, abstract schema is the basis for implementing many (binary) operations on `AffTrees`, such as function composition, addition, or equality. The schema is a direct consequence of the fact that `AffTrees` are decision trees with two kinds of nodes—inner nodes and terminal nodes representing affine

predicates and affine functions, respectively—and therefore need one update function for each kind. The following schema is a generalization of the one presented in [AP I: [Sch+23]].

Definition 6.3.8. Let α and β denote two update functions for inner and terminal nodes of an `AffTree`. Further, let $t_1, t_2 \in \mathscr{T}$ be two `AffTrees`, then their $\alpha\beta$-composition $t_1 \square_{\alpha,\beta} t_2$ is defined as:

$$\begin{aligned} n \square_{\alpha,\beta} n' &:= \beta(n', n) \\ n \square_{\alpha,\beta} (p, l, r) &:= (\alpha(p, n), n \square_{\alpha,\beta} l, n \square_{\alpha,\beta} r) \\ (p, l, r) \square_{\alpha,\beta} t &:= (p, l \square_{\alpha,\beta} t, r \square_{\alpha,\beta} t) \end{aligned}$$

where n, n' are terminal nodes and (p, l, r) represents an inner node with predicate p and left child l and right child r.

This schema can be realized through a depth-first search in the first tree until a terminal n is reached, which is stored for reference. Then, another depth-first search is performed for the second tree. Here, all encountered inner nodes p are updated by $\alpha(p, n)$ and then added to the first tree as a subtree under n. When a terminal n' is reached, it is instead updated with $\beta(n', n)$. Optionally, one might check for infeasible edges before adding new nodes.

In case of *addition*, this schema can be directly applied:

$$t + t' := t \square_{\pi_1,+} t'$$

where

$$\pi_1(x, y) = x$$

To see why this works, it can be helpful to recapitulate the semantics of addition for two PWL functions $f, g \colon \mathbb{R}^n \to \mathbb{R}^m$:

$$(f + g)(\vec{x}) = f(\vec{x}) + g(\vec{x})$$

One can directly identify that the evaluation of f is independent from the evaluation of g and vice versa. Next, we look at the linear regions of $f + g$. From the above equality we can conclude that when f and g are linear on $Q \subseteq \mathbb{R}^n$, then so is $f + g$. When $\mathscr{Q}$ is a polyhedral partition of f and $\mathscr{R}$ is a polyhedral partition of g, then $\mathscr{Q} \cap \mathscr{R}$ is a polyhedral partition of $f + g$. An example of this can be seen in the superposition of the wave functions discussed in Example 4.1.2 (page 38). Intersection of polytopes corresponds to conjunction of the corresponding affine predicates:

$$\text{Models}(\pi_1) \cap \text{Models}(\pi_2) = \text{Models}(\pi_1 \wedge \pi_2)$$

And finally, conjunction of predicates can be achieved in `AffTrees` by concatenating the paths. In a similar way, one can define other algebraic operations, such as *subtraction*, *multiplication*, or *division*.

For the case of *function composition*, evaluation is no longer independent:

$$(f \circ g)(\vec{x}) = f(g(\vec{x}))$$

Here, g acts essentially as a 'lens' that transforms how f perceives the input space. Keeping in mind that g is PWL, then for a local region it is indeed affine and affine functions are geometric transformations. So the metaphor with the lens is actually not that far of. Therefore, composition can be implemented by

$$t \circ t' \coloneqq t \; \square_{\circ,\circ} \; t'$$

The α update of a an inner node p from t' with a terminal n from t is a form of composition. Since p is an affine predicate and n is an affine function

$$(p \circ n)(\vec{x}) = p(n(\vec{x}))$$

is again an affine predicate. In this manner, the lens-effect is realized. The composition of the terminals is straightforward. The ability to compute sequential composition directly on `AffTrees` is essential for the algebraic properties discussed in Figure 3.2 (page 33). With that we have [AP I: [Sch+23]]:

$$v(t_1 \circ t_2) = v(t_1) \circ v(t_2) \quad .$$

This in turn also means that we can directly apply any `Afflang` term $t \in \mathscr{A}$ to an existing `AffTree` $T \in \mathscr{T}$ by computing:

$$v(t) \circ T \quad .$$

Finally, relations like *equality* can also be expressed in this schema:

$$t = t' \coloneqq t \; \square_{\pi_1,=} \; t'$$

A derived concept called *epsilon-equality* can be stated as

$$|t_1 - t_2| \leq \epsilon$$

where $t \leq \epsilon$ can be implemented in `Afflang` simply as

$$\texttt{if}\ (t \leq \epsilon)\ \texttt{then}\ 1\ \texttt{else}\ 0 \quad .$$

Algebraic properties of `AffTrees` are discussed in depth in [AP I: [Sch+23]].

Part II

Practice

CHAPTER 7

Semantics-Preserving Optimizations

In the previous part, we have seen how symbolic execution can be applied to a number of deep neural network architectures. We proceeded to define a tree representation (`AffTree`) based on the CFG of symbolic execution in such a manner that the resulting decision tree acts as a faithful yet interpretable surrogate model (FISM) of the original network. Further, we proved various semantic and algebraic properties of these structures.

Now we want to focus on a practical realization of the computation of these decision trees. Specifically, this section will discuss optimizations for the distillation approach that impact the running time and memory complexity. As we will see, both aspects are closely related, as the computation is polynomially bounded with respect to the size of the computed decision tree.

In the SOS rules of the calculus, we have seen that the symbolic execution treats the parallel application of the independent activation functions in a sequential and thus conditional way. As such, the number of branches is not additive but multiplicative, which results in an overall exponential growth with respect to the number of hidden neurons with branching activation functions. The number of branches/regions per activation function defines the base of this exponent. Using the general model of the fully-connected neural network architecture

$$\mathscr{N} = \sigma_L \circ h_{L-1} \circ \cdots \circ \sigma_1 \circ h_1$$

where h_ℓ are affine functions and σ_ℓ are non-affine activation functions ($\ell \in [L-1]$). Assume that all activation functions σ_ℓ are PWL and are an element-wise application of some underlying function $\varsigma \colon \mathbb{R} \to \mathbb{R}$, i.e.,

$$\sigma(\vec{x}) = (\varsigma(x_1), \varsigma(x_2), \ldots, \varsigma(x_n))^\intercal$$

For example, ReLU is an element-wise application of $\varsigma(x) = \max\{x, 0\}$. Then each ς must also be PWL with linear regions $Q_1, \ldots, Q_k \subseteq \mathbb{R}$ and associated linear functions $\ell_1, \ldots, \ell_k$. Essentially, that means each ς consists of up to k linear functions, which

it can attain for some regions of the input. In the case of ReLU, the regions are $Q_1 = (-\infty, 0]$ and $Q_2 = (0, \infty)$ and the linear functions are $\ell_1(x) = 0$ and $\ell_2(x) = x$. Now, define a new function η that takes on the linear function of the index it is given:

$$\eta(i)(x) = \ell_i(x) = m_i \cdot x$$

So with the index i we can select the appropriate linear function. Using the characteristic function, we can then restrict each of them again to their domain:

$$\varsigma(x) = \mathbb{I}(x \in Q_1) \cdot \ell_1 + \cdots + \mathbb{I}(x \in Q_k) \cdot \ell_k$$

Then we can characterize each activation function by a tuple $\vec{a} \in [k]^n$ of numbers from 1 to k:

$$\begin{aligned}\eta(\vec{a})(\vec{x}) &= (\eta(a_1)(x_1), \eta(a_2)(x_2), \ldots, \eta(a_n)(x_n))^\intercal \\ &= (m_{a_1} x_1, m_{a_2} x_2, \ldots, m_{a_n} x_n)^\intercal \\ &= \hat{m}_{\vec{a}} \odot \vec{x}\end{aligned}$$

From there, one can apply the same concept to all hidden layers to describe the network:

$$\mathcal{N}(\vec{a_1}, \ldots, \vec{a_{L-1}}) = \sigma_L \circ \eta_{L-1}(\vec{a_{L-1}}) \circ \cdots \circ \eta_1(\vec{a_1}) \circ h_1$$

The collection of all $\vec{a_i}$ corresponds to an activation pattern of the network that fixes all PWL components to one of their linear regions. As a result, the whole composition is linear. Since the network is guaranteed to be linear for each activation pattern, it is clear that the number of activation patterns is at least as large as the number of linear regions. It is also strictly larger, which can be seen by setting the last activation function to the constant zero function. Then, the network has only one linear region, but can have arbitrarily many activation patterns. When all activation functions have k regions, the total number of activation patterns is

$$k^{n_1 + \cdots + n_{L-1}}$$

where n_i is the number of neurons in the i-th (hidden) layer.

However, not all activation patterns are attainable be the network. Consider the simple example:

$$\mathrm{ReLU}(3\,\mathrm{ReLU}(x) + 2)$$

It is impossible for the outer ReLU to be in the region $(-\infty, 0]$ while the inner ReLU is in region $(0, \infty)$, as both share a clear dependency on the initial value of x. It is possible to characterize what initial values of x must be given such that a specific region is attained. In this case, the predicates are:

$$x > 0 \wedge 3x + 2 \leq 0$$

which are infeasible. These predicates are exactly the ones that were computed in the symbolic execution in Section 6.1. Thus, there is a one-to-one correspondence between activation patterns and the paths in `AffTrees`.

The exponential growth of activation patterns is problematic, as the number of neurons tends to be large and is increasing more and more in state-of-the-art

models. This is not surprising, as it is assumed that deep neural networks with more neurons can (i) express a wider family of functions, and (ii) have better convergence to minima in the loss space (*lottery ticket hypothesis*) [FC18]. Thus, it is important to consider how one can reduce the number of branches that are considered. For that, one can either (a) follow a semantics-preserving approach, which is generally preferable based on the requirements stated in Section 1.1.4, or (b) look at more aggressive reduction methods that alter the semantics. In the latter case, we still want to be able to analyze the network, thus one needs formulas to bound the introduced error. Here we will focus on semantics-preserving optimizations. The already discussed concolic execution and preconditions (Section 6.2) are correct but not complete analyses, which can also be seen as a form of optimization.

In general, when trying to reduce the number of branches without altering the represented semantics, one needs to find *redundancies* in the current representation. For that, we will consider techniques of reducing redundancy in the decision tree representation, which have been proposed in the context of program analysis or decision diagrams. We will see how these optimizations play out for `AffTrees`, which are in many ways different from typical decision trees. With respect to decision trees, the following properties are distinct to our problem:

- The domain of possible predicates is uncountable
- The domain of possible terminal values (affine functions) is uncountable (before classification)
- The probability that any two predicates are the same is equal to zero
- Predicates interact with each other due to semantic dependency
- Predicates and terminals interact with each other due to semantic dependency

With respect to program analysis (especially symbolic execution), the following points differ from the typical case:

- The value of all variables can be stated as an affine formula
- Feasibility checks are decidable in polynomial time (linear program (LP) solving)
- The CFG is acyclic (for RNNs after unrolling)

7.1 Identifying and Pruning Infeasible Paths

In a labeled graph (N, E, ζ), a path $(\zeta, v_1, \ldots, v_n)$ is called *infeasible* when the conjunction of all decisions on that path is not satisfiable:

$$\text{Models}\left(\bigwedge_{i=1}^{n} \pi_i\right) = \emptyset$$

As no input could ever take such a path due to contradictory constraints, these paths can be safely removed from the graph (N, E, ζ) without altering its semantics (i.e., without changing the represented PWL function). Infeasible path elimination is the corresponding optimization that prunes infeasible paths in a graph or program. While it is mostly discussed in the setting of program optimization and symbolic execution, the optimization is also applicable to decision structures, such as ADDs, where predicates share semantic dependencies [AP III: [Mur+23]].

7.1.1 General Definition

Historically, infeasible path elimination was first considered in form of eliminating partial and total redundancies in programs. The following definition is based on [MR79]:

Definition 7.1.1 (Partial and Total Redundancy). In a labeled graph (N, E, ζ), an edge $e = (v, l, u) \in E$ with associated predicate π is *partially redundant* iff there exists at least one path $\rho = (\zeta, v_1, \ldots, v_n, v)$ from the root node ζ to the edge e with associated path condition P such that the logical formula

$$\forall \vec{x} \in \mathbb{R}^n : P(\vec{x}) \implies \pi(\vec{x})$$

is valid. Conversely, the edge e is *totally redundant* if the same property holds for all paths ρ from the root node ζ to the edge e.

While totally redundant edges can be removed safely from the graph without changing the represented function, partial redundancy is harder to remove [Ste96]. When the considered graph is a tree, the notion of partial and total redundancy coincide, as every node is then reachable from the root node by exactly one path. A very similar concept is called *infeasibility*, where an edge e is infeasible iff

$$\forall \vec{x} \in \mathbb{R}^n : P(\vec{x}) \implies \neg\pi(\vec{x})$$

holds. For binary tree structures where the outgoing edges of any node have mutual exclusive predicates of the form π and $\neg\pi$, it is the case that one edge is redundant iff the other is infeasible. In such cases, we can remove the infeasible edge and forward or skip the redundant edge and thereby remove the associated node altogether. The subtree of the infeasible edge, if present, is also infeasible by the transitivity of the infeasible relation and can therefore be removed without altering the represented function.

7.1.2 Infeasible Paths in Decision Trees

With these observations it becomes clear that infeasible path elimination can have a significant optimization potential depending on how many paths are infeasible. In a tree, paths can be identified with nodes, therefore the optimization potential is dependent on how many nodes are infeasible and also their position within the tree (i.e., distance to the root node). For example, consider that on the second level of a tree already a path is infeasible, then (assuming a balanced tree) a quarter of the tree is infeasible and can be pruned.

As all predicates in `AffTrees` are affine predicates, the path conditions P, which are themselves just conjunctions of the individual occurring affine predicates along the path, are again affine predicates. Therefore, each path implies a polytope in the input domain of points that will follow the path correspondingly. When the path is infeasible, this region is empty. The problem of determining feasibility can therefore be reduced to solving a LP.

Theorem 7.1.2. Let $(N, E, \zeta) \in \mathscr{T}$ be an `AffTree` and let $\rho = (\zeta, v_1, \ldots, v_n)$ be a path of this tree with associated path condition P. The path ρ is infeasible iff the LP

$$\begin{aligned} \min \quad & c^\mathsf{T}\vec{x} \\ \text{s.t.} \quad & P(\vec{x}) \end{aligned}$$

has no solutions (the choice of $\vec{c} \in \mathbb{R}^n$ is indifferent).

Let us consider a brief proof sketch. (Necessity) Proof by contradiction over the cases: Suppose the program has an optimal solution, then it is a vertex of Models P that satisfies P, so the path would be feasible. Suppose the program is unbounded, then it would have infinitely many solutions (but no optimal solution). Then the set Models P is unbounded in the direction where $\vec{c}$ decreases and therefore not empty. (Sufficiency) When P is infeasible, the associated polytope Models P is empty, so no solutions exist.

Therefore, the problem of determining infeasibility can be solved in polynomial time. A very similar LP can be used that gives more information about the structure of the polytope:

Definition 7.1.3 (Chebyshev Center). For a polytope $P \subseteq \mathbb{R}^n$ given by an inequality $\boldsymbol{A}\vec{x} \leq \vec{b}$, a *Chebyshev center* is one solution of the LP

$$\begin{aligned} \max \quad & \begin{bmatrix} 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \vec{x} \\ r \end{bmatrix} \\ \text{s.t.} \quad & A_{i,:}\vec{x} + \|A_{i,:}\| r \leq b_i \\ & r \geq 0 \end{aligned}$$

where $A_{i,:}$ is the i-th row of $\boldsymbol{A}$. Each solution $\begin{bmatrix} \vec{x} & r \end{bmatrix}^\mathsf{T}$ implies a *maximally contained* sphere S with center point $\vec{x} \in \mathbb{R}^n$ and radius $r \in \mathbb{R}_{\geq 0}$

$$S := \{\, \vec{y} \in \mathbb{R}^n \mid \|\vec{y} - \vec{x}\| \leq r \,\} \subseteq P$$

Maximally contained refers to the property that every sphere contained in P has a radius not larger than r.

An example of using the Chebyshev centers for all (feasible) paths of an `AffTree` is given in Figure 7.1. As the LP of the Chebyshev center only has one additional variable with respect to Theorem 7.1.2, its asymptotic running time only marginally increases with respect to the original LP. However, in practice algorithms usually calculate the optimal solution of a linear program step-by-step. In many LP solvers computing the solution is split into two phases [SZ15]:

1. Find a valid solution
2. Iteratively improve the current solution to converge to the optimal solution

Thus, having a constant cost function makes the whole second phase obsolete, improving on the running time. By how much depends on the method and implementation of the LP solver, and should be measured.

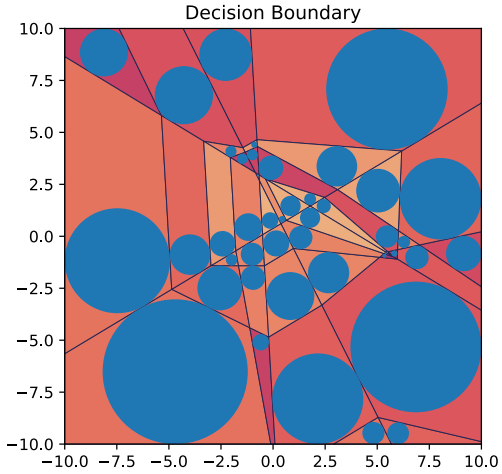


Figure 7.1: Example of enclosed inspheres (in blue) computed with the Chebyshev centers. The example shows the partitioning (before argmax) for the same network depicted in the spiral example in Figure 5.7.

7.1.3 Global Approach

We have identified a form of redundancy, called infeasibility, that concerns paths in `AffTrees`. We also have determined how *individual* infeasible paths can be identified by solving LPs. Next, we want to find a method for efficiently identifying all infeasible paths in an `AffTree`. While one could check each path of the tree individually, this is not the most efficient way. Formally, this is based on the monotonicity of infeasibility.

Lemma 7.1.4. In an `AffTree` (N, E, ζ), whenever a path $(\zeta, v_1, \ldots, v_n)$ is infeasible, then so is every continuation of this path $(\zeta, v_1, \ldots, v_n, u_1, \ldots, u_m)$.

Therefore, paths should be checked for infeasibility in an order that respects the subpath relation (or, from the perspective of the nodes, respecting topological ordering). This ensures that an infeasible prefix is found and eliminated from the tree before any of the descendants of the path have been considered. Both depth-first search and breadth-first search respect the subpath ordering. A naive implementation of this approach, however, introduces a new problem: The total number of LPs solved increases considerably. Suppose an `AffTree` has $n > 1$ terminal nodes, then the number of all paths from the root node to a terminal is also n. However, the number of paths from the root node to any other node in the tree is equal to $2n - 2$.

This problem can be solved by storing the result from LP solving. When we check the path $(\zeta, v_1, \ldots, v_n)$ with associated path condition P for feasibility with an LP solver, then we receive a solution $\vec{x}$ for P if the path is feasible. When v_n is not a terminal node, it is an inner node with two children. Call them u_1 and u_2. When π is the path condition from v_n to u_1, then $\neg\pi$ is the path condition from v_n to u_2. Through elementary logical transformations, one can show that the solution $\vec{x}$ to P must hold for one of the two outgoing edges of v_n:

$$\begin{aligned}(P \wedge \pi)(\vec{x}) \vee (P \wedge \neg\pi)(\vec{x}) &\equiv (P \wedge (\pi \vee \neg\pi))(\vec{x}) \\ &\equiv P(\vec{x})\end{aligned}$$

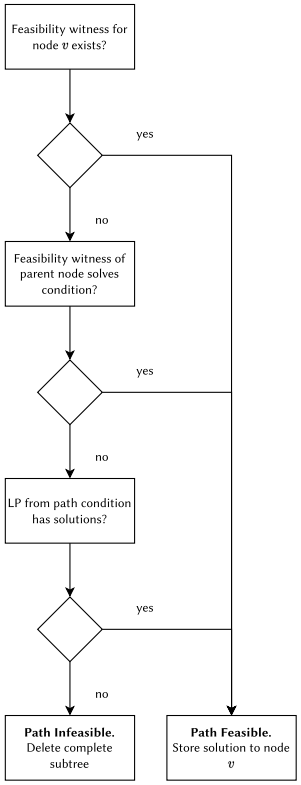


Figure 7.2: Flow chart of the process of elimianting infeasible paths in an `AffTree`.

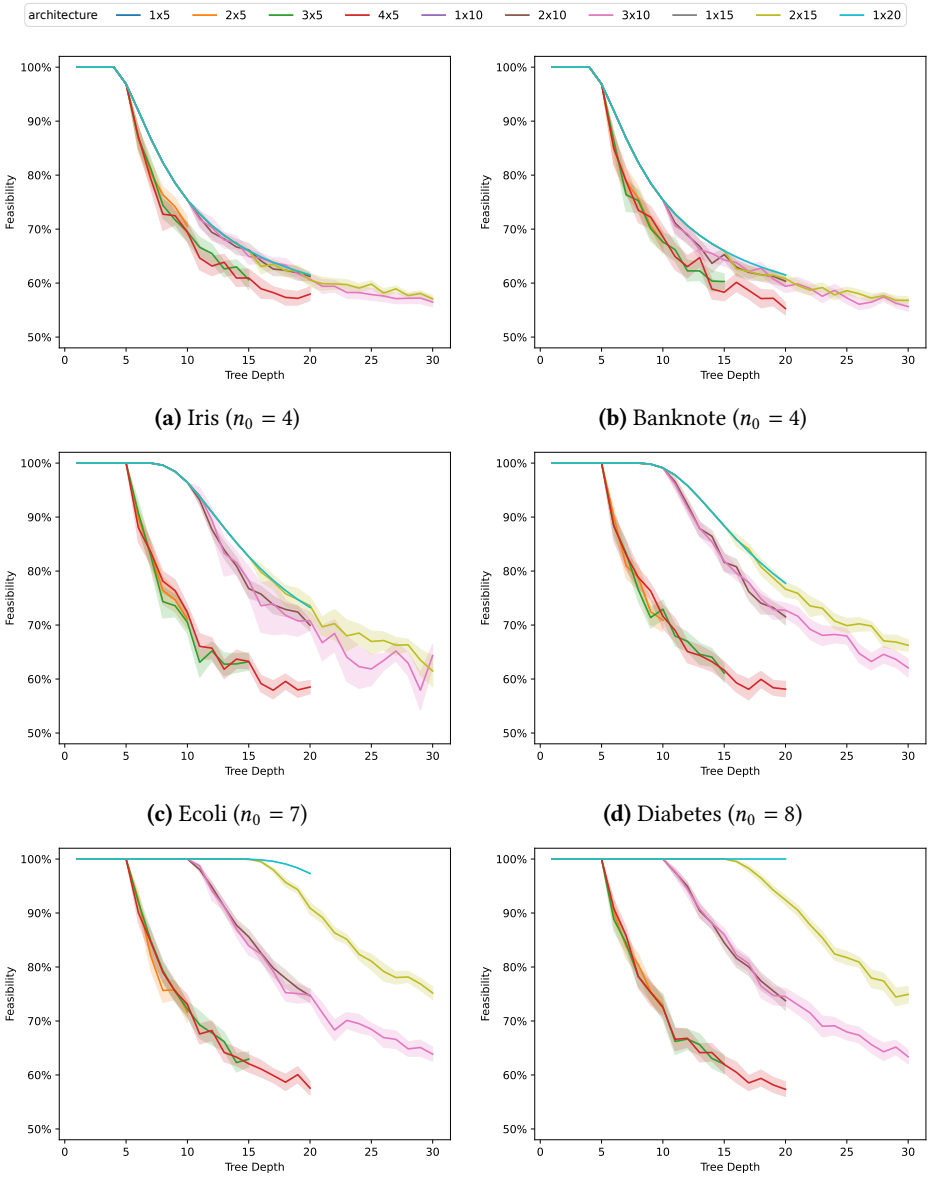
This leads to the improved approach: Whenever we reach the node v_n, we first check if $\pi(\vec{x})$ is true.[1] If this is the case, the path $(\zeta, v_1, \ldots, v_n, u_1)$ is feasible. If this is not the case, the path $(\zeta, v_1, \ldots, v_n, u_2)$ is feasible. After that, we proceed to solve the LP for the remaining path. We call $\vec{x}$ a *witness* for the feasibility of P and say that u_1 or u_2 *inherit* the solution from v_n. With that, we can check infeasible paths in an `AffTree` with $n > 1$ terminals by checking exactly n LPs. When the tree is balanced, one can also easily compute the number of constraints for each of these n LPs. For each $k \in [\log_2 n]$, we solve

$$2^{k-1} \text{ linear programs with } k \text{ constraints}$$

We also need to solve one additional LP with 1 constraint, as for the first two outgoing edges of the root node no saved witness exist that could be inherited. The resulting algorithm is depicted in Figure 7.2.

Finally, one can improve the performance by checking feasibility using depth-first instead of breadth-first search. For one, because depth-first search is generally more performant than breadth-first search on full binary trees (e.g., better space complexity). Additionally, some LP solvers support adding a new constraint to a solved LP, which can then be solved faster. For example, matrix factorizations and tableaus can be reused or iteratively updated.

[1] This operation can essentially be considered as "free" or $\mathcal{O}(1)$ in comparison with LP solving. Using a linear algebra library like BLAS, it can be computed by a single GEMM call.



(a) Iris ($n_0 = 4$)

(b) Banknote ($n_0 = 4$)

(c) Ecoli ($n_0 = 7$)

(d) Diabetes ($n_0 = 8$)

(e) Wine ($n_0 = 13$)

(f) Breast Cancer ($n_0 = 30$)

Figure 7.3: Comparison of layer-wise feasibility over `AffTrees` distilled from various DNNs. Feasibility is computed as the percentage of splits in that layer that are infeasible. Each color represents one specific network architecture (depth, width), for which 50 different initialized DNNs were distilled. Their mean and 95% confidence intervals are reported.

7.1.4 Empirical Evaluation

To conclude, a few experiments are considered regarding the effectiveness of infeasible path elimination. In these experiments, `AffTrees` have been distilled for 3000 neural networks. The networks were trained with varying random seeds for initialization, depth ($2 - 4$), and width ($5 - 20$). Further, the complete setup was repeated for six datasets with different numbers of input features. Distillation was performed iteratively by mapping each layer to an `AffTree` and then composing them. After

each composition step, infeasible path elimination was applied. Figure 7.3 shows the ratio of the number of nodes after infeasible path elimination over the number of nodes before it was applied. In other words, the effect is shown with respect to a doubling of nodes of the previous layer. Importantly, the effect is shown for each layer after all previous layers have been pruned, therefore compound effects are not included: When a node at layer 10 is infeasible but not pruned, it would have a subtree of size $2^{20} \approx 10^6$ at the end. The experiments clearly show that:

1. Infeasible paths exist in all network architectures
2. Infeasibility increases with the depth of the network
3. The input dimension n_0 influences infeasibility
4. The width of previous layers influences infeasibility in the layers thereafter
5. Infeasible paths can only occur for the neurons in the first hidden layer when their number surpasses the input dimension n_0

The third point is a result of how many regions can occur in hyperplane arrangements where the number of hyperplanes surpasses the dimension of the ambient space [Zas81], which is studied for a while in the context of upper bounding the number of linear regions of neural networks, e.g., [Mon+14]. The fourth item was pointed out in [STR18], where it is called a *bottleneck*. The effect is well-known and can be easily demonstrated when just considering linear functions. Suppose $f, g, h \colon \mathbb{R}^8 \to \mathbb{R}^8$ are linear functions where f and h are invertible (have full rank) and g has rank 2. Then, their composition

$$h \circ g \circ f$$

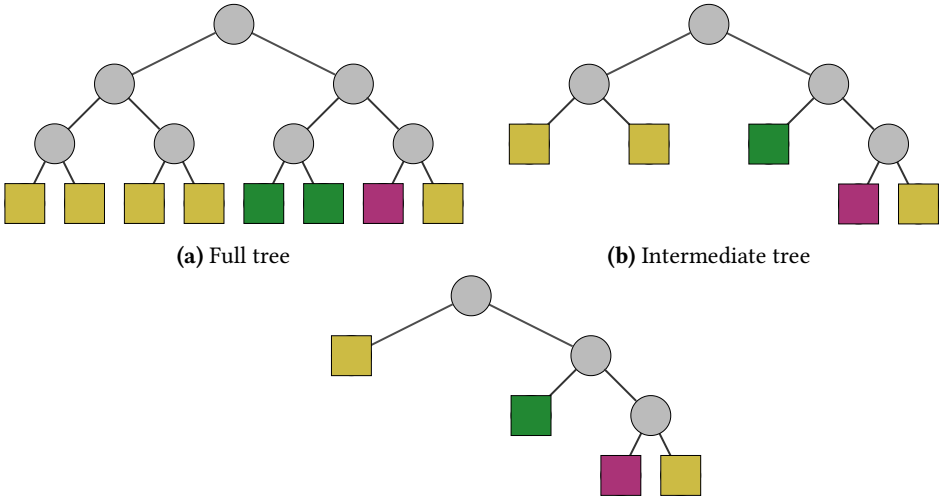
also has rank 2, i.e., its image is a linear space with dimension 2 embedded in $\mathbb{R}^8$. The lost dimensions through the bottleneck g cannot be recovered by the following function h. Although the argument is more complicated for PWL functions, the general concept can be mapped to that domain as well [STR18].

Overall, the experiments show that infeasible path elimination is a powerful, semantics-preserving optimization that is applicable to a wide range of neural networks. As the number of pruned nodes depends on the input dimension, it is especially effective when combined with concolic execution. For larger networks, non-semantics-preserving optimizations might be necessary.

7.2 Redundant Predicates

A decision can also be redundant in a different sense: when its left and right subtree compute the same function under the possible inputs of the node. In ADDs, the ordering of predicates results in a normal form that makes it easier to detect semantic equivalency of subtrees through a syntactic comparison [Bry86]. Sorting the predicates in `AffTrees` would come with great cost, as almost all of its predicates are different. By the same argument, this form of redundancy—where two subtrees compute the same function—is very unlikely to occur in real scenarios.

However, the argument changes for classification settings. Then, even when all *predicates* of the tree are different, subtrees can compute the same function as long as their terminals agree, which is more likely for this discrete domain. In such cases,



(a) Full tree **(b)** Intermediate tree

(c) Optimized tree

Figure 7.4: Merging of equivalent subtrees results in a more efficient decision tree. On the left: Full decision tree with 8 terminals belonging to three different classes as represented by color. On the right: Reduced tree where semantically equivalent subtrees have been merged.

a simple bottom-up search can be already effective. For every second to last node in an `AffTree`, check whether its left and right node (which are terminals) are equal. In that case, replace the node with either of the terminals and remove the other. Nodes should be considered in reverse topological order to ensure that the left and right child have been simplified as much as possible. This is due to the fact that, in this version of merging, only terminals can be merged.

An example of the result of applying this optimization to an `AffTree` for classification is shown in Figure 7.4. In the first pass of the third layer of nodes (the four gray circles that each have a depth of two in the tree), merging is applied three times. After that, the second layer is checked (two gray circles with a depth of one), in which the first can be merged as both children are now also yellow terminals, while the second cannot. In the first layer (root node), no merging can be performed.

After a preliminary empirical evaluation, it seems that merging rarely passes up the tree for many layers, but instead can only be successfully applied to the last few layers (as is to be expected). However, given that `AffTrees` are always full binary trees, most of the nodes are located near the bottom of the tree anyway. Furthermore, this optimization is fast, as it is linear in the number of nodes of the tree and only performs two comparisons of matrices for each node.

7.3 Order of Operations

The order of operations can also influence the number of activation patterns. Consider the equivalent terms

$$\max\{\mathrm{ReLU}(x), \mathrm{ReLU}(y)\} = \mathrm{ReLU}(\max\{x, y\})$$

While the left-hand side has the following eight activation patterns:

$$
\begin{array}{lcll}
x \leq 0 \land y \leq 0 \land 0 \leq 0 & \mapsto & 0 & \\
x \leq 0 \land y \leq 0 \land 0 > 0 & \mapsto & 0 & \dagger \\
x \leq 0 \land y > 0 \land 0 \leq y & \mapsto & y & \\
x \leq 0 \land y > 0 \land 0 > y & \mapsto & 0 & \dagger \\
x > 0 \land y \leq 0 \land x \leq 0 & \mapsto & 0 & \dagger \\
x > 0 \land y \leq 0 \land x > 0 & \mapsto & x & \\
x > 0 \land y > 0 \land x \leq y & \mapsto & y & \\
x > 0 \land y > 0 \land x > y & \mapsto & x &
\end{array}
$$

where infeasible patterns are marked with †, the right-hand side has only four:

$$
\begin{array}{lcl}
x \leq y \land y \leq 0 & \mapsto & 0 \\
x \leq y \land y > 0 & \mapsto & y \\
x > y \land x \leq 0 & \mapsto & 0 \\
x > y \land x > 0 & \mapsto & x
\end{array}
$$

As this simple example demonstrates, the order of operations can influence the number of activation patterns.

There is currently no automated way to detect this form of redundancy in a generic way. It is also not appearing often, as usually learnable parameters are inbetween the application of such functions, i.e., maximum and ReLU are often used as activation functions. However, for example in CNNs, Max Pooling and ReLU are sometimes used in direct succession. In such cases, their order can easily be improved manually for efficiency.

CHAPTER 8

Use Cases: Explainability Applications

This section demonstrates how the presented theory, in particular the faithful distillation approach of Chapter 6 and the optimizations of Chapter 7, can be used in different practical scenarios to gain *explanatory information* [Lan+21]. For that, this thesis is accompanied by the tool `Affinitree`[1], which is an open-source Rust implementation for the faithful distillation of neural networks. `Affinitree` is designed to be modular and compositional, supporting the framework mindset: Users can choose from the building blocks `Affinitree` provides to create their own interpretability pipeline to express many different kinds of analyses (e.g. based on slices Figure 8.1). Where necessary, users can also easily define their own blocks, such as custom activation functions. The extracted surrogate models correspond to the presented `AffTrees` (Section 6.3), guaranteeing that, on a mathematical level, the resulting information is always correct. Instead of constructing the full tree, users can choose, on a case-by-case basis, to focus on specific regions of the input, thereby trading off completeness for execution time (Section 6.2).

Each of the following scenarios highlights how `Affinitree` can be used in conjunction with established tools and techniques from data analysis, dimensionality reduction, and visualization to achieve interpretability. These examples follow a holistic approach, illustrating both `Affinitree`'s capabilities and its interoperability with established techniques.

All experiments described in this chapter were conducted using `Affinitree` on an AMD EPYC 7763 CPU with 256 cores à 2.45 GHz and 2 TB of RAM. The training of DNNs was done with PyTorch on an NVIDIA GeForce RTX 3090 GPU.

In [AP V: [SS24]], similarly designed experiments were already presented. The following discussion differs from it in the following ways: A new in-depth introduction to the respective settings has been added, presenting technical details about fairness and adversarial attacks, as well as background information about the involved datasets. For fairness, a new scenario based on a synthetic dataset is added.

[1] `https://github.com/Conturing/affinitree-py`

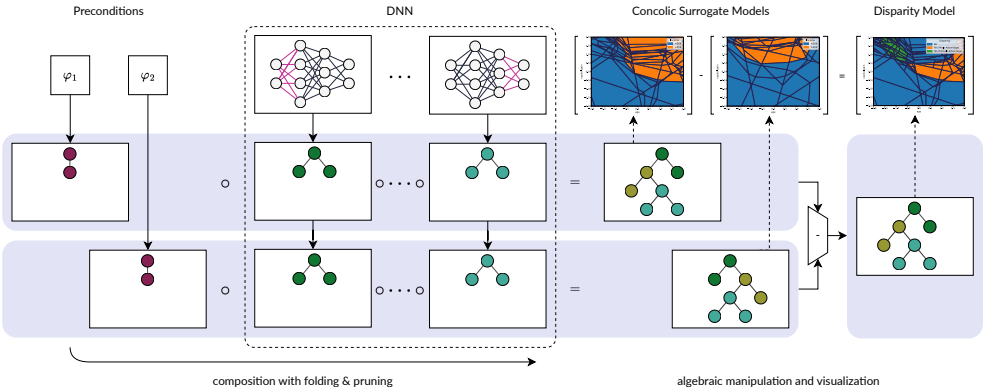


Figure 8.1: Overview of the general interpretation pipeline in `Affinitree` for slice-based visualizations. Preconditions can be added that restrict the input space. Then follows a standard distillation layer-by-layer with optimizations. For each precondition, a separate tree is produced. For the disparity model, the difference of these trees is calculated using their algebraic properties. Slightly improved version of a figure published in [AP V: [SS24]].

For the second fairness scenario, UCI adult, the same experiments are reported as in [AP V: [SS24]]. For the robustness scenario, experiments were repeated with a different adversarial attack method. An artifact is available for the experiments of [AP V: [SS24]] at `https://zenodo.org/records/12707712`.

8.1 Fairness Analysis

Problem Definition. Algorithmic fairness has recently received much attention motivated by the increasing large scale application of (artificial intelligence (AI)) software systems in safety-critical areas [WZZ22]. Fairness involves addressing systematic disparities in error rates or other conditional probabilities across sub-populations differing in a sensitive attribute [Meh+21; Olt+19]. One popular metric, *demographic parity*, states that the model predicts a positive outcome for equally many inputs across all demographic groups (e.g., men and woman). Formally, it is the difference between the *positive rate (PR)* between two groups differing in a sensitive attribute x_s, which can be stated as:

$$\left|P(h(X) = 1 \mid x_s = 0) - P(h(X) = 1 \mid x_s = 1)\right|$$

where $h(X) = 1$ refers to a positive prediction for the random variable X by the DNN h. These probabilities are typically estimated using their empirical counterparts, i.e., derived from finite samples, such as the training set.

DNNs are particularly susceptible to fairness issues due to their chaotic nature and reliance on good training data (as pointed out by the saying “garbage in, garbage out”). Achieving fairness in DNNs remains an open but rewarding challenge, as it is a prerequisite for the adoption of DNNs in safety-critical applications. Not only does it foster an ethical and responsible use, but it is also necessary to comply with legal regulations (specifically EU non-discrimination law [WMR21], American anti-discrimination law [BS16]; more generally GDPR [GF17], liability [Bor23; Vla14]). Existing strategies to mitigate bias in DNNs include (i) regularization, (ii) data pre-processing, and (iii) retraining with extended datasets [Loh+22].

The following two subsections each present a fairness evaluation case demonstrating `Affinitree`'s capabilities in these matters. In Section 8.1.1, a synthetic dataset is introduced with three input attributes, out of which one is protected. The data distribution is chosen to be simple to follow, and the optimal decision boundary is a hyperplane. As such, the number of external factors that can affect the performance and behavior of trained DNNs over this dataset is limited, which makes it an optimal introduction case. Due to its low dimensionality, it is possible to visualize and analyze the fairness performance of trained DNNs across the complete input space. In contrast, Section 8.1.2 uses an established dataset based on real-world census data to train a DNN. The dataset features greater complexity, such as complex data distributions, spurious correlations, and class imbalance. Consequently, the analysis of trained models is harder to comprehend and also more challenging. Given the higher input dimensionality, a direct visualization of the complete input space is impossible. Instead, `Affinitree`'s concolic mode is utilized to create plots of two-dimensional slices of the input space. While holistic fairness performance cannot be visualized, one can instead judge individual fairness along each slice.

8.1.1 Synthetic Dataset: Logistic Model with Bias

To start, let us consider a toy example that illustrates the technical aspects of the distillation-based approach to fairness analysis, while being easier to follow than noisy, high-dimensional real world data. For that, we construct a synthetic dataset based on a stochastic model with a built-in bias. This ensures a controlled environment with predictable outcomes while at the same time simulating two realistic sources of bias: *historic bias* and *representation bias* [Meh+21]. Concretely, the dataset will exhibit a linear correlation between the three input features and the target prediction.

Dataset. The setting of our synthetic dataset is placed at a bank that wants to use a DNN to automate deciding loan approvals based on three features they deemed relevant: *yearly income* (ranging from 25 to 120, in thousands of dollars), *age* (18-70 years), and *gender* (0 encoding 'male' and 1 encoding 'female').[2] Suppose that the past behavior of human operators in this bank can be modeled by the following stochastic model that states the probability of loan approval:

$$p(\mathit{loan}) = \sigma(0.5 \cdot \mathit{income} + 0.08 \cdot \mathit{age} - 4.0 \cdot \mathit{gender} - 30)$$

where $\sigma(x) = \frac{1}{1+e^{-x}} : \mathbb{R} \to (0, 1)$ is the sigmoid function. Stochastic models of this form, e.g., where the probability is given by the application of the sigmoid function over a linear combination of features, are known as *logistic models* [HLS13]. When factoring in the ranges of the features, yearly income has the most discriminatory power w.r.t. loan approval. Next, age has only a slight impact on the loan approval. Finally, the gender term has a non-zero coefficient, which shows that the (fictional) loan approval practice of this bank is biased. In other words, the feature gender is used to discriminate (in a technical sense) between individuals who get the loan

[2] For consistency with the following case based on the older UCI adult dataset, non-binary genders were not added.

and ones who do not. When this is indeed necessary for correct prediction—e.g. gender might be useful when estimating the body height of an individual—it is called *explainable discrimination*, otherwise it is called *unexplainable discrimination* [Meh+21]. For the purposes of this hypothetical scenario, we define this term as unexplainable discrimination based on poor decision-making of the bank in the past. The intercept term is chosen to create a plausible decision boundary, which can be directly stated as a formula by solving for $p(loan) = 0.5$:

$$income = 60 - 0.16 \cdot age + 8.0 \cdot gender \ . \tag{8.1}$$

To complete the setting, we must also state the customer demographic of this bank, i.e., select distributions of the three features income, age, and gender. To balance simplicity with plausibility, the attributes gender and age are uniformly sampled from their range, while yearly income is modelled by a log-normal distribution with parameters $\mu = \ln(40), \sigma = 0.4$. Based on these distributions, a dataset consisting of 10000 points was created by randomly sampling from the described stochastic model. The randomness is important to establish a realistic setting that also contains spurious correlations.

Model. We train 50 DNNs $h^{(1)}, \ldots, h^{(50)} : \mathbb{R}^3 \to \mathbb{R}^2$, each consisting of a single hidden layer with 10 neurons and ReLU activations, initialized with different random seeds using the Kaiming normal method. Each DNN is trained for 30 epochs over batches of size 64 with cross entropy loss and the AdamW optimizer with learning rate 0.01 and learn rate scheduling. This setup allows comparing the consistency and variance of the analysis results over a series of models with the same architecture but different initializations.

Interpretability. Using `Affinitree`, we analyze the trained DNNs individually to investigate disparities in the decision-making of the models with respect to the protected attribute gender. For each DNN, we start by extracting the learned parameters and layer structure into a universal format from NumPy for storing matrices, which can then be read in again by `Affinitree`. Next, we define preconditions $\varphi : \mathbb{R}^2 \to \mathbb{R}^3$ (Section 6.2.2) for both values of the gender attribute, i.e., $x_s = m = 0$ for male and $x_s = f = 1$ for female, as follows:

$$\big(\varphi[s \mapsto c](x)\big)_i = \begin{cases} x_i & \text{if } i < s \\ c & \text{if } i = s \\ x_{i-1} & \text{if } i > s \end{cases} \tag{8.2}$$

The other two features, age and income, are interpreted symbolically. Then, we distill each DNN $h^{(i)}$ into one `AffTree` instance $t_c^{(i)} : \mathbb{R}^2 \to \{0, 1\}$ once with the male ($c = m$) and once with the female precondition ($c = f$):

$$t_c^{(i)} = \nu\big(\texttt{argmax} \circ \delta\big(h^{(i)}\big) \circ \varphi[s \mapsto c]\big) \ .$$

As each of the three functions in the above composition is piecewise linear, their composition remains piecewise linear. Thus, the ν function can be applied to distill

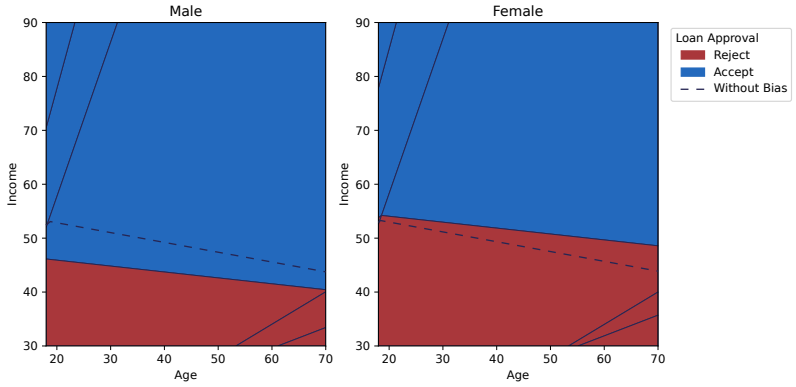


Figure 8.2: Preimage partition of one of the fifty DNNs trained on the synthetic loan approval dataset. The left plot shows the prediction of the DNN when the gender attribute is set to male and on the right when it is set to female. One can clearly observe that the decision boundaries (transition from blue to red) change depending on the value of the gender attribute.

the term into an `AffTree`. In the resulting `AffTree`, an input $\vec{x} \in \mathbb{R}^2$, where x_0 corresponds to the age and x_1 to the income, is passed sequentially through the tree following a path of predicates of the form $a_0 x_0 + a_1 x_1 \leq b$ with coefficients $a_0, a_1, b \in \mathbb{R}$. When the predicate is true, the edge with label 1 is taken, otherwise the edge with label 0. At the end of each path lies a terminal node, which in this case stores the classification result in form of either the constant function $\vec{x} \mapsto 0$ or $\vec{x} \mapsto 1$. In this way, the tree deterministically maps each input $\vec{x}$ to a corresponding prediction $h^{(i)}(x)$ of the original DNN $h^{(i)}$.

By traversing the tree $t_c^{(i)}$ in depth-first order, one can efficiently compute the path condition π for every path $\rho = (\zeta, \ldots, v)$ in the tree from the root node ζ to a terminal node v. Each pair (π, α), where α is the affine function associated with v, defines a unique linear region of the piecewise linear DNN h under the specified precondition. For classification tasks, α is always a constant function mapping inputs to one of the classes $c \in \mathscr{C}$.

The *preimage partition* of the represented PWL function of an `AffTree` can then be plotted based on these pairs each defining one linear region. For each region, the vertices of the convex polytope π are computed, and the enclosed region is shaded with a color corresponding to the class c designated by α. Such a plot is visualized in Figure 8.2, which depicts the preimage partitions for one of the fifty DNNs considered, under both preconditions (male and female).

While we currently have 50 individual trees that we can compare pairwise with each other, a global approach can be achieved by leveraging the algebraic properties of `AffTrees`. We can aggregate all 50 models $h^{(1)}, \ldots, h^{(50)}$ into a single tree, which states for each input point $\vec{x}$ how many of the models predicted a positive or negative outcome:

$$\bar{t}_s = \sum_{i=1}^{50} (x \mapsto 2x - 1) \circ t_s^{(i)} \quad .$$

Here, $x \mapsto 2x - 1$ is used to map predictions $\{0, 1\}$ to $\{-1, 1\}$. Then, $\bar{t}_s \colon \mathbb{R}^2 \to \{-50, \ldots, 50\}$ states, as desired, the number of models that predict a positive outcome

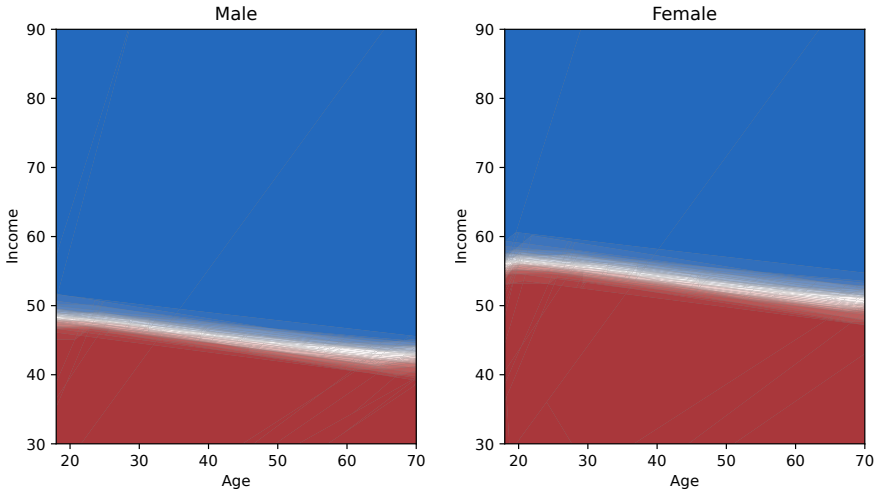


Figure 8.3: Overlay of the decision boundaries of 50 DNNs trained on the synthetic loan approval dataset. Both plots show the prediction of a DNN for all inputs with ages 18-70 and income 30-90. The plots only differ in the value of the *protected* gender attribute. On the left, gender is fixed to 'male' and on the right to 'female'. One can clearly observe that the decision boundaries (transition from blue to red) change consistently depending on the value of the gender attribute across all 50 DNNs. Concretely, the loan for a woman of age 50 with a yearly income of 50 thousand dollars would be rejected by all models while a man with the same age and income would be accepted. Both plots are generated by `Affinitree`.

minus the number of models that predict a negative outcome. As a result, the final tree illustrates where the models agree or disagree, and also how much variance they exhibit with respect to the decision boundary. In Figure 8.3, one can directly see that all DNNs clearly *replicated* the bias from the dataset (as expected). We can also see that the decision boundaries of most models agree and are similar to the theoretic boundary of the stochastic model (Equation (8.1)). Additionally, the overall variance is relatively low, as can be seen by the short white region. This shows that in this case no *algorithmic bias* was involved, i.e., the bias is not a result of the model architecture or a training artifact.

As we constructed the dataset by hand, we knew exactly what kind of bias was contained in the training samples. In real applications, judging what kind of bias is present is often not so easy. For completeness, we also trained a DNN without the bias in the dataset and then the classification boundaries were very similar for both genders (dotted line in Figure 8.2).

8.1.2 UCI Adult Dataset

The UCI Adult dataset is a popular dataset considered in many publications about algorithmic fairness [Din+21].[3] In contrast to our synthetic dataset, it consists of more features and has unbalanced classes, which introduce additional artifacts that can result in biased models. Moreover, the dataset likely includes *historic bias* or *population bias*, as the data was collected in 1994 and demographic parameters have

[3]With 375936 views (on 22.01.2025), it is one of the most visited datasets on `https://archive.ics.uci.edu/datasets`.

changed since then. `Affinitree` offers a framework that facilitates the identification and visual exploration of *individual fairness* [Dwo+12] violations. The following discussion was already presented in [AP V: [SS24]].

Dataset. The UCI adult dataset [KB96] is a prominent dataset in the field of algorithmic fairness [Din+21]. It consists of 14 attributes, out of which 6 are numeric and 8 are categorical. The aim is to predict whether an individual's income surpasses $\$50,000$ ($n_L = 2$). It is based on the 1994 US Census and therefore includes many sensitive attributes, like "Sex", "Race", and "Age".[4] The dataset has inherent systematic problems, which often lead to biased classifiers. Most notably, the dataset suffers from *class imbalance* as it contains more instances with the attribute "Male" than "Female" (sex); "White" than "Asian-Pac-Islander", "Amer-Indian-Eskimo", "Other", and "Black" (race); and "<50K" than ">50K" (target). In [Din+21], the issues of the dataset were described as:

> In 1994, the median US income was 26,000 dollars, and 50,000 dollars corresponds to the 76th quantile of the income distribution, and the 88th and 89th quantiles of the income distribution for the Black and female populations, respectively. Consequently, almost all of the Black and female instances in the dataset fall below the threshold and models trained on UCI adult tend to have substantially higher accuracies on these subpopulations. ([Din+21])

Model. For this scenario, a DNN $h\colon \mathbb{R}^8 \to \mathbb{R}^2$ consisting of four hidden layers, each containing 10 neurons with ReLU activations, was trained for 20 epochs, achieving 83.50% accuracy on the test set. Before training, each numeric feature was individually standardized to have zero mean and unit variance:

$$x_i = \frac{x_i - \mu}{\sigma} \quad .$$

All categorical attributes except "Sex" where removed to streamline the discussion. The remaining categorical attribute ("Sex") was one-hot encoded, resulting in an input size of $n_0 = 8$. Training was conducted with the Adam optimizer with a learn rate of 0.001 and a batch size of 64. Both NumPy and PyTorch were initialized with the random seed 30 and the network's weights were initialized with the Kaiming normal method. Fairness regularization in form of Kullback-Leibler projection of the true positive rate (TPR) from the `fairret` library [BDD23] was applied. TPR was chosen as it can handle class imbalanced and is computationally efficient. The updated loss used in training had the form:

$$\mathscr{L}(\theta, \vec{x}, y) = \mathscr{L}_{\text{cross-entropy}}(\theta, \vec{x}, y) + \lambda \cdot \mathbb{E}[\mathrm{KL}(f^\star(X) \mid\mid h(X))]$$

where f^* is a hypothetical model that adheres to the fairness criterion (here, equalized TPR) and $\lambda \in \mathbb{R}_{\geq 0}$ is a hyperparameter. Based on empirical tests $\lambda = 2$ was set. The fairness regularization did indeed lower the disparity for five tested fairness metrics, but also slightly reduced accuracy.

[4] These are the historic column names for the features of this dataset and might not align with today's terminology.

Table 8.1: Fairness metrics for the considered DNN over the test set with respect to the sensitive feature "Sex". A consistent disparity can be observed across all metrics. Accuracy scores are higher for females based on the class imbalance, as most female instances belong to the class "<50K".

Metric	Disparity ($\lvert\Delta\%\rvert$)	Female (%)	Male (%)
Accuracy Parity [Ber+21]	13	92	79
Demographic Parity (PR) [Dwo+12]	20	3	23
Equal Opportunity (TPR) [HPS16]	21	23	54
Predictive Parity (FPR) [Cor+17]	9	1	10
Equalized Odds (max{TPR, FPR}) [HPS16]	21	23	54

A summary of popular fairness metrics [VR18; PS23] evaluated for this DNN with respect to the sensitive attribute "Sex" is provided in Table 8.1. A consistent disparity of the fairness metrics between the male and female subpopulation can be observed. For these metrics, the following ratios from a confusion table are relevant:

Positive Rate (PR) is the proportion of all inputs predicted as positive by h among the whole population.

True Positive Rate (TPR) is the proportion of inputs correctly predicted by h as positive among those actually exceeding an income of \$50K.

False Positive Rate (FPR) is the proportion of inputs erroneously predicted by h as positive among those earning less than \$50K.

The corresponding fairness metrics than state that these rates are equal for all subpopulations, e.g., *demographic parity* demands equal PRs, and *equal opportunity* demands equal TPRs. As can be seen in Table 8.1, the DNN consistently predicts the positive outcome (defined as a yearly income exceeding 50K\$) in favor of the male demographic. The high disparity over the TPR and false positive rate (FPR) indicates that (i) the model exhibits *unexplainable discrimination* and (ii) the model appears more biased than the dataset itself, potentially indicating the introduction of *algorithmic bias* during training, despite the application of fairness regularization. Only the accuracy is higher (better) for the female subpopulation, which is likely caused by the class imbalance.

Interpretability. With `Affinitree`, we can highlight specific instances of possible bias or unfair treatment within the model's decision-making process. For applying it, let us formalize violations of individual fairness for our case, which states that similar inputs differing in the sensitive attribute sex are treated differently by the DNN h:

$$h(\vec{x}[s \mapsto m]) \neq h(\vec{x}[s \mapsto f]) \tag{8.3}$$

where $s \in \mathbb{N}$ denotes the sensitive feature, and $\vec{x}[s \mapsto c]$ sets attribute s of $\vec{x}$ to the value $c \in \mathbb{R}$. We directly extended this notion to input regions. A *region of disparity* $Q \subseteq \mathbb{R}^{n_0}$ is an inclusion-maximal set such that all points in that set

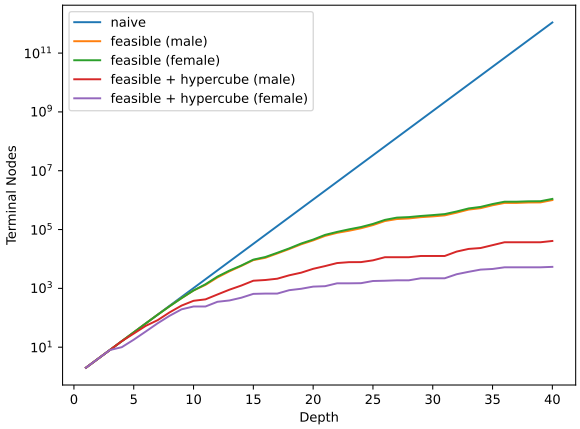


Figure 8.4: Comparison of the number of terminal nodes of the discussed `AffTrees` in Equations (8.4) and (8.5) and their theoretical worst case (naive implementation). The different optimization strategies successfully reduce the number of nodes without sacrificing correctness. Published in [AP V: [SS24]].

satisfy Equation (8.3). Now, the regions of disparity of h are exactly the connected components of the inputs that satisfy the following algebraic formula:

$$\texttt{argmax} \circ \delta(h) \circ \varphi[s \mapsto m] \neq \texttt{argmax} \circ \delta(h) \circ \varphi[s \mapsto f] \tag{8.4}$$

where $\varphi[s \mapsto c]$ is defined as in Equation (8.2). Both sides of the term are piecewise linear and can therefore be distilled into an `AffTree` instance. Since only two classes are involved here, one can compute equality (or the absence thereof) by subtracting both sides. A result of 1 indicates a male advantage, of 0 that both agree (achieving individual fairness), and of -1 a female advantage.

As discussed in Section 6.2.2, it is best to restrict the analysis of DNNs to probable inputs. Since the data has been already normalized, a simple yet efficient *convex precondition* is given by the hypercube $P = [-1.0, 1.0]^6 \times [0.0, 1.0]^2$:

$$h_c = \texttt{argmax} \circ \delta(h) \circ \mathbb{I}(P) \circ \varphi[s \mapsto c]$$

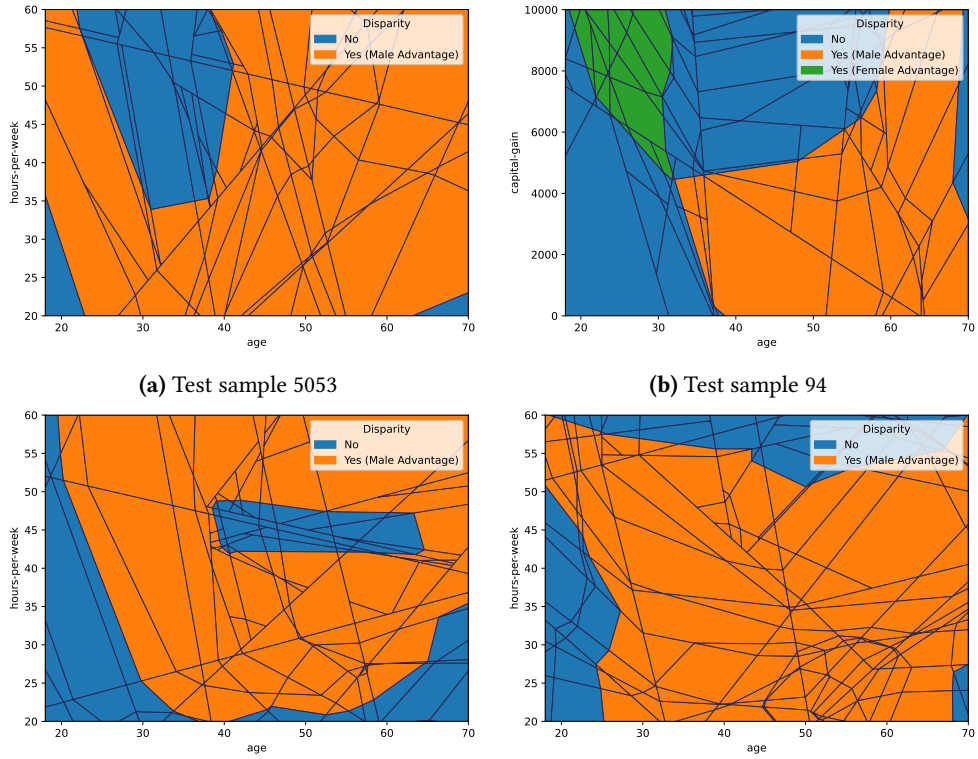
where $\mathbb{I}(X)$ is a an `Afflang` program similar to the characteristic function of polyhedral set X (Section 6.2.2):

$$\mathbb{I}(X)(\vec{x}) := \begin{cases} \vec{x} & \text{if } \vec{x} \in X \\ \lightning & \text{otherwise} \end{cases}$$

Thus, the faithful distillation can be stated as:

$$\begin{aligned} t_{\text{disparity}} &= \nu(h_m - h_f) \\ &= \nu(\texttt{argmax}) \circ \nu(\delta(h)) \circ \nu(\mathbb{I}(P)) \circ \nu(\varphi[s \mapsto m]) \\ &\quad - \nu(\texttt{argmax}) \circ \nu(\delta(h)) \circ \nu(\mathbb{I}(P)) \circ \nu(\varphi[s \mapsto f]) \end{aligned} \tag{8.5}$$

This term can be encoded and executed in `Affinitree`, producing a tree that exactly characterizes all regions of disparity.



(a) Test sample 5053

(b) Test sample 94

(c) Test sample 3074

(d) Test sample 5043

Figure 8.5: Visualization of extreme cases of disparity in two-dimensional slices aligned with coordinate axes based on concolic execution by fixing values from test samples. Inspired by [AP V: [SS24]].

This example showcases the efficacy of the distillation optimizations presented in Chapter 7. Without any optimizations, a naive implementation would branch on each of the 40 partial ReLUs, resulting in a total of $2^{40} \approx 10^{12}$ paths or $2^{41} - 1$ nodes in the tree. Instead, with optimizations the tree t_m has 81803 nodes and t_f has 10789 nodes, which correspond roughly to a reduction in size by a factor of $26.88 \cdot 10^6$ and $203.82 \cdot 10^6$, respectively. The final disparity model $t_{\text{disparity}}$ of Equation (8.5) is computed from these two trees by subtraction, i.e., $t_{\text{disparity}} = t_m - t_s$. Without optimizations the resulting tree has $|t_s| \cdot |t_m| \approx 882 \cdot 10^6$ nodes, but with optimizations only 808713 nodes, which corresponds roughly to a reduction in size by a factor of 1091. In Figure 8.4, the number of terminal nodes after distilling the first n partial ReLUs is shown for t_m and t_f with no optimizations ■, with infeasible path elimination ■■, and with infeasible path elimination and precondition ■■.

In order to produce visualizations as we previously did for the synthetic dataset (e.g., the preimage partition), we have to modify the precondition, as the input dimension is here six instead of two (already accounting that the gender attribute is fixed). One of the axes, “fnlwgt” (final weight), is not suitable for visualization. For visualization, we may select two of the remaining five attributes as axes of our plot and must fix the other four attributes with some interesting values, which also should make sense together. For example, the two attributes ‘hours-per-week’ and ‘age’ are correlated (statistically dependent), thus their values should be selected

in a way that respects the underlying distribution. One simple yet effective way to find such value pairs is by selecting training samples as source. Given a sample $\vec{z} \in \mathbb{R}^8$ and a set of two axes $I \subseteq [8]$ with $s \notin I$, $|I| = 2$, one can define a function $\psi_I \colon \mathbb{R}^8 \to \mathbb{R}^2 \to \mathbb{R}^8$ as

$$\big(\psi_I(\vec{z})(\vec{x})\big)_i := \begin{cases} x_i & \text{if } i \in I \\ z_i & \text{otherwise} \end{cases}$$

For a fixed $\vec{z}$, the function is linear and can be used for conclic execution as described in Section 6.2.1. With that, we can define an `AffTree` $t(\vec{z}, I) \colon \mathbb{R}^2 \to \{-1, 0, 1\}$ that characterizes violations of individual fairness along a two-dimensional slice I as follows:

$$\begin{aligned} h_c(\vec{z}, I) &= \texttt{argmax} \circ \delta(h) \circ \mathbb{I}(P) \circ \psi_I(\vec{z}[s \mapsto c]) \\ t(\vec{z}, I) &= \nu\big(\, h_m(\vec{z}, I) - h_f(\vec{z}, I) \,\big) \end{aligned}$$

The pairs (α, π) of terminals α with path condition π of this tree partition the input space $P \subseteq \mathbb{R}^8$ into convex regions where, depending on α, either individual fairness (Equation (8.3)) is satisfied ($\alpha = 0$), males have an advantage ($\alpha = 1$), or females have an advantage ($\alpha = -1$).

We found that roughly 10% of samples from the training set and 13% from the test set violate individual fairness according to Equation (8.3). For the 1314 samples $\vec{z}$ from the test set that violate individual fairness and all axis combinations $I \in \{\{i, j\} \mid 0 \le i < j \le 6 \wedge i, j \neq 1\}$, we distilled the `AffTree` $t(\vec{z}, I)$. Based on the resulting trees, we computed the ratio between the areas of disparity to the overall area (which is bounded by P):

$$\frac{\mu(\bigcup\{\, \pi \subseteq \mathbb{R}^2 \mid t(\vec{z}, I) \text{ has a path with condition } \pi \text{ and terminal } \alpha \in \{-1, 1\} \,\})}{\mu(P)}$$

As all polytopes involved here are two-dimensional, their area can be computed exactly by the shoelace formula. The highest ratio of 72% was found for sample number 5053 with axis 'age' and 'hours-per-week' (Figure 8.5a). Although the DNN was regularized to improve fairness, we saw a considerable disparity between the areas that favored females to the ones that favored males, the latter being considerably higher. The highest female advantage (6.85%) was found for sample number 94 with axis 'age' and 'capital-gain', which is visualized in Figure 8.5b.

8.2 Robustness Analysis

In this scenario, we use `Affinitree` to visually explore decision boundaries near adversarial examples. For that, we distill decision trees of a fixed DNN using preconditions that are built around adversarial examples. We show that, despite reasonable accuracy scores on the test set, there exist directions $\vec{v}$ in the input space where adversarials consistently occur, e.g., $h(\vec{x} + \lambda\vec{v}) \neq h(\vec{x})$ for a large range of λ. Such directions highlight systematical vulnerabilities of the DNN.

Problem Definition. Adversarial examples are a well-known phenomenon of DNNs that are often cited to highlight their chaotic nature [GSS14]. The most popular form of adversarials are defined as:

Definition 8.2.1 (Adversarial Examples). Let $h : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ be a neural network, and let $\vec{x} \in \mathbb{R}^{n_0}$ be a given point belonging to class c. An *ϵ-adversarial example* is a point $\vec{x_{\text{adv}}} \in \mathbb{R}^{n_0}$ with a maximal distance of ϵ to the original point $\|\vec{x} - \vec{x_{\text{adv}}}\|_p \leq \epsilon$, but which is *incorrectly* classified as belonging to a different class

$$h(\vec{x_{\text{adv}}}) \neq c$$

by the DNN h.

This definition works for many different kinds of models, architectures, and datasets, thanks to two key ideas: First, the ground truth for some point $\vec{x} \in \mathbb{R}^{n_0}$ is known, which is an information that cannot be extracted purely from the model h. This also implies that the given point $\vec{x}$ is well-formed and has a unique label, i.e., it is not some outlier like noise. Secondly, the adversarial example $\vec{x_{\text{adv}}} \in \mathbb{R}^{n_0}$ must lie close in the space to $\vec{x}$ with respect to some l-norm, which entails that the adversarial must share many semantic properties with the original point $\vec{x}$. With this definition, one can mechanically search for undesired semantical behavior of the model h while only considering syntactical perturbations.

It is postulated that adversarial examples are, to some degree, the result of the *curse of dimensionality* [Cha+19; Sha+18; God+23]. When the input space has a high dimensionality, the distribution of training samples is sparse relative to their enclosed volume, which in turn leaves regions in the input space where no training data exists. In these regions where training samples are absent, interpolation is by definition hard and the model must extrapolate, which is in general more error-prone for all kinds of models. Now, when for some point $\vec{x} \in \mathbb{R}^{n_0}$ such a region lies in direction $\vec{v}$, it is reasonable to assume that along the line $\vec{x} + \lambda\vec{v}$ with $\lambda \in \mathbb{R}_{>0}$ the probability of adversarials is increased. For illustrative purposes, one may think geometrically of cone-shaped objects that emit in one direction from a point $\vec{x}$ where no training sample can be found:

$$\{\, \vec{x} + \lambda\vec{u} \mid \lambda \in \mathbb{R}_{>0}, \langle \vec{u}, \vec{v} \rangle \geq \|\vec{u}\|\|\vec{v}\| \cos\theta_{\max} \,\} \quad .$$

Theoretical work in this area [TKC22] has shown that adversarial examples tend to reside in the orthogonal complement of the tangent space of the data manifold. In this scenario, we will see further evidence that adversarial examples indeed span over much larger regions.

Notably, other factors for the occurrence of adversarials have been identified as well, which are, however, less relevant for the presented scenario. Linear models are particularly vulnerable to adversarial examples, as demonstrated in [GSS14; Cub+17]. Since piecewise linear DNNs are locally linear, they inherit similar vulnerabilities. Some publications also show that the smoothness of the decision boundary, or more precisely, the Lipschitz constant of the network, plays a role in the frequency of adversarial examples [Pau+21].

When a DNN is not specifically trained for robustness, it is usually not hard to find adversarial examples. Many methods have been proposed for finding adversarial examples in DNNs [AM18]:

- Fast Gradient Sign Attack [GSS14], Fast Gradient l_2 [Miy+18]
- The Basic Iterative Method [KGB18]
- Carlini and Wagner Attacks [CW17]

For the generation of adversarial examples in this scenario, the Fast Gradient l_2 [Miy+18] attack was chosen. The attack uses the loss function $\mathscr{L}$ to guide the search to regions where the DNN performs worse:

$$\vec{x_{\text{adv}}} = \vec{x} + \epsilon \cdot \text{mask} \cdot \frac{\nabla_x \mathscr{L}(\theta, \vec{x}, y)}{\|\nabla_x \mathscr{L}(\theta, \vec{x}, y)\|_2} \tag{8.6}$$

Additionally, a binary mask was applied to fix all values that clearly belong to the background to 0.

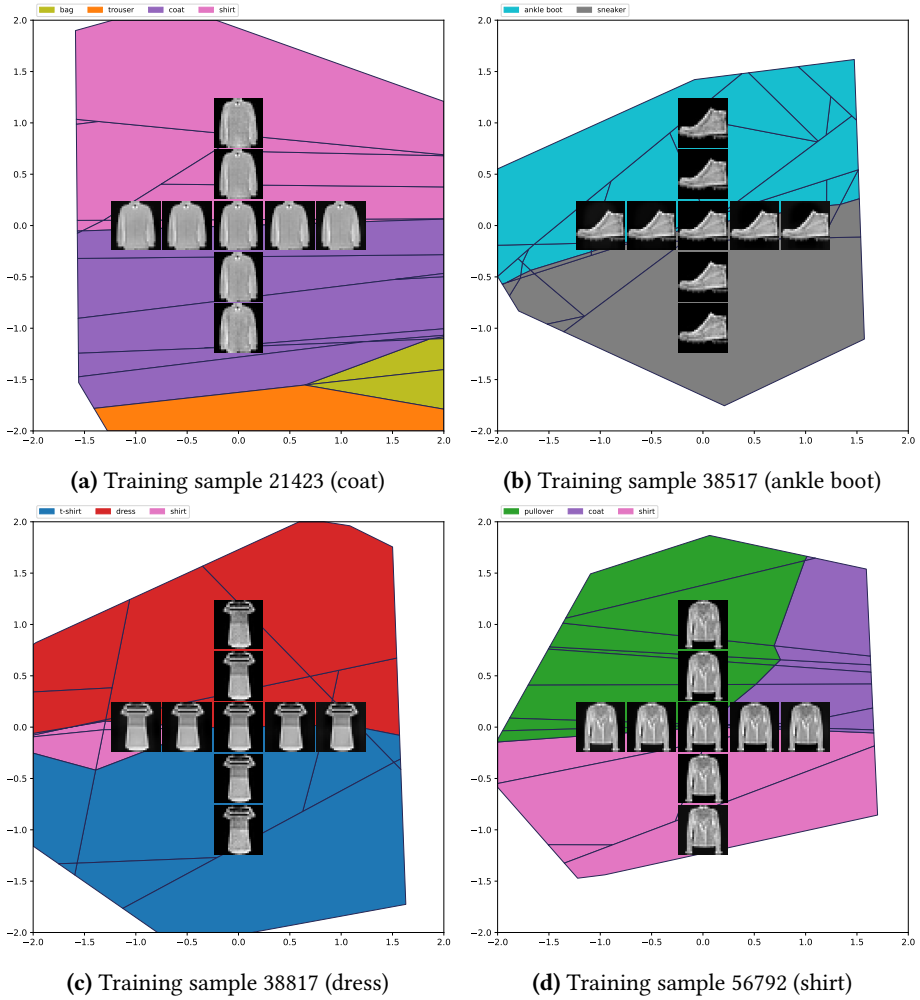
Dataset. For this example, we consider the Fashion MNIST dataset [XRV17], which consists of 28×28 pixel grayscale images ($n_0 = 784$) representing ten categories of fashion articles ($n_L = 10$):

- t-shirt
- trouser
- pullover
- dress
- coat
- sandal
- shirt
- sneaker
- bag
- ankle boot

The dataset is well-suited for the study of adversarial examples, as the perturbations made can be visually inspected to ensure that the adversarial is still a valid input. In contrast to MNIST, it is not linearly-separable and has a more complex structure.

Model. A DNN $h\colon \mathbb{R}^{784} \to \mathbb{R}^{10}$ with three hidden layers, each containing 15 neurons with ReLU activations, was trained on Fashion MNIST for 20 epochs, achieving 84.9% accuracy on the test set. Training was performed with the Adam optimizer with a learning rate of 0.001 and batch sizes of 64 and data shuffling enabled. NumPy and PyTorch were initalized with the random seed 42 and the network was initalized using the Kaiming normal method. To compute the PCA over the dataset the implementation from SciPy was used. The Fast Gradient l_2 attack method identified 111 adversarial examples from 50000 training samples when setting a small perturbation of $\epsilon = 0.01$.

Interpretability. To visualize the decision boundaries near an adversarial example $\vec{x_{\text{adv}}}$, which must be erroneous to some degree, we again slice the input space into interesting two-dimensional planes $S(\vec{x_{\text{adv}}}) \subset \mathbb{R}^{784}$ using concolic execution



(a) Training sample 21423 (coat)

(b) Training sample 38517 (ankle boot)

(c) Training sample 38817 (dress)

(d) Training sample 56792 (shirt)

Figure 8.6: Preimage partition of slices centered at an adversarial example. Vertical axis corresponds to the attack direction, horizontal axis to the first principal component.

(Section 6.2.1). From the theory of linear algebra, it is known that such a plane can be represented as

$$S(\vec{x_{\text{adv}}}) = \{\, \vec{x_{\text{adv}}} + \lambda_1 \vec{u_1} + \lambda_2 \vec{u_2} \mid \lambda_1, \lambda_2 \in \mathbb{R} \,\}$$

where $\vec{u_1}, \vec{u_2} \in \mathbb{R}^{784}$ are linearly independent vectors. When the original point $\vec{x}$ should be contained in this plane, one of those vectors must be proportional to $\vec{x} - \vec{x_{\text{adv}}}$, i.e., the attack direction. For Fast Gradient l_2, this direction is by Equation (8.6) equivalent to $\frac{\nabla_x \mathscr{L}(\theta, \vec{x}, y)}{\|\nabla_x \mathscr{L}(\theta, \vec{x}, y)\|_2}$. The other direction can be chosen freely. For a dataset with 784 dimensions it is hard to find single directions in the space that captures the behavior well. One reasonable choice is the the first principal component of the dataset, $\vec{pc_1}$, as it is the direction along which most of the variance of that dataset occurrs. Using standard linear algebra techniques, one can define a distance-preserving, linear mapping $\varphi(\vec{x_{\text{adv}}}) \colon \mathbb{R}^2 \to S(\vec{x_{\text{adv}}})$, which is the desired precondition for our distillation:

$$\varphi(\vec{x_{\text{adv}}})(\vec{y}) = Q\vec{y} + \vec{x_{\text{adv}}}$$

where $\begin{bmatrix}\vec{x} - \vec{x_{\text{adv}}} & \vec{pc_1}\end{bmatrix} = QR$ is the QR-factorization of the spanning vectors, producing an orthonormal matrix Q with the same image space.

To complete the process, a precondition (Section 6.2.2) is added to restrict inputs to plausible values. For that, we add a hypercube $[0,1]^{784}$ as a constraint. When putting the precondition after the conclic projection function, the constraints are also sliced. Here this can be seen in the form that the added constraints are all parallel to the axes, but the boundary of the slice will instead be a non-axis-aligned convex polytope. With that we can now define an `AffTree` $t(\vec{x_{\text{adv}}}) \colon \mathbb{R}^2 \to \{0, \ldots, 9\}$ that corresponds to the network's classification in the plane $S(\vec{x_{\text{adv}}})$ as:

$$t(\vec{x_{\text{adv}}}) := v\big(\texttt{argmax} \circ \delta(h) \circ \mathbb{I}([0,1]^{784}) \circ \varphi(\vec{x_{\text{adv}}})\big)$$

Another benefit of the concolic execution is a faster execution time, as more paths become infeasible, i.e., every path ρ where the implied polytope $\pi \subseteq \mathbb{R}^{n_0}$ does not intersect the two-dimensional plane $\pi \cap S(\vec{x_{\text{adv}}}) = \emptyset$.

Following the same steps as in the previous two scenarios, we create a plot of the two-dimensional slice $S(\vec{x_{\text{adv}}}) \cap [0,1]^{784}$ centered at the adversarial example (Figure 8.6). When looking at the reference images (the two images to the north, east, south, and west of the central image), one can convince oneself that all perturbations are indeed subtle to imperceptible. So we may note that all input images are not degenerate edge cases (e.g., out-of-distribution data or random noise), but actually depict valid clothing items. Nevertheless, the dress depicted in Figure 8.6c is also seen as a t-shirt ■ and shirt ■ by the model. The ankle boot depicted in Figure 8.6b is also seen as a sneaker ■. The coat in Figure 8.6a is also seen as t-shirt ■, bag ■, and trouser ■. And finally, minimal perturbations over the shirt in Figure 8.6d result in predictions of pullovers ■ and coats ■. While some of these mistakes are understandable (on a first glance the dress can for example be mistaken as a t-shirt), other confusions seem implausible (the coat is clearly not a bag or trouser). As stated at the beginning, the plots also show that the attack direction points to a region where many adversarial examples can be found.

8.3 Final Remarks

Two practical applications were presented that demonstrated how the theory of the first part can be used in explainability settings. The computation of two-dimensional slices with the concolic mode and infeasible path elimination combined rigorous analysis with comprehensible visuals. While being a good start, future research should provide additional methods for effectively utilizing the explanatory information provided by `AffTrees`.

CHAPTER 9

Related Work

This chapter lists related work to the topic of this thesis, i.e., to explain deep neural networks through faithful surrogate models. The field of explainable AI (XAI) research is diverse, spanning many different concepts and paradigms. The survey by Arrieta et al. [Arr+20] provides a comprehensive overview of XAI methods and categorizes them into several taxonomies (Figures 6 and 11 are especially relevant). Their general classification follows the same dimensions as the one already used in Chapter 1.

There are *ante-hoc* [Spe22] or *transparent model* [Arr+20] or *intrinsic* [AB18] methods that work with ML models that are inherently understandable or transparent, such as logistic or linear regression, decision trees, rule-based systems, or k-nearest neighbors. Other authors consider only a subset of them as explainable [Gui+18; Lip18; Mol25]. Moreover, these systems are not applicable to all tasks. In contrast, *post-hoc* explanations do not alter the model, but instead explain its decisions or the complete model *after* training has finished. In this way, the model's performance is not influenced.

A popular post-hoc, local, model agnostic XAI technique is LIME (Local Interpretable Model-Agnostic Explanations) [RSG16]. In this approach, a decision is explained by sampling many small perturbations of the original input and training a linear model over these. In this way, a local linear approximation of the original model is created, from which explanations can be generated more easily.

SHAP (SHapley Additive exPlanations) [LL17] is another popular, model-agnostic, local, post-hoc explainability method. It computes feature relevance scores by applying the concept of Shapley values from game theory.

Another local, post-hoc technique that is mostly model-agnostic is called Layer-wise Relevance Propagation (LRP) [Bac+15; Mon+19], which propagates relevance scores backwards through the model. For that, the model is decomposed into layers for which custom propagation rules are defined. Then going back from the prediction, at each layer the importance of every input (e.g., neuron) is computed until the input is reached. In the case of images, the relevance score is given for each input pixel and can be overlaid with the original input.

Especially in the context of image classification, heat maps and saliency maps are an intuitive approach to highlight feature importance. These are again local, post-hoc methods that are usually applied to DNNs. For DL systems, a common approach is to infer relevance from the gradient $\frac{\partial h}{\partial x}$ of the input image x, such as gradient explanations [SVZ13; Erh+09; Bae+10], gradient $\odot$ input [Shr+16], integrated gradient [STY17], GradCAM [Sel+17], and SmoothGrad [Smi+17].

A post-hoc approach specific to DNNs is Activation Maximization [NYC16], which computes the input (usually an image) that results in the strongest activation of a neuron. When applied to a neuron representing a class (like "grocery store"), the method produces images that can be interpreted as representatives of the class according to the DNN. In this way, the method highlights global properties of the model.

Recently, it was shown that post-hoc methods often fail to capture the intricate structure of DNNs, leading users to form false beliefs about the system [BKS17; Rud19; MI22; INM19; LB20; Dim+20]. For example, in [Ade+18] it was shown that certain proposed saliency maps are indifferent to changes in the parameter of a DNN. In [Sla+20], it was shown that models can be altered to change LIME and SHAP explanations while maintaining their accuracy and other performance metrics.

9.1 Theoretical Analysis of Linear Regions

Closely related to this work are methods that analyze PWL DNNs by decomposing them into their linear regions. Such approaches can be found in theoretical publications that prove formal properties of DNNs and in works on explainability that provide information about the internal logic of DNNs.

The PWL nature of ReLU neural networks was first studied in [PMB13; Mon+14]. These works proved the first upper bounds on the number of linear regions that ReLU neural networks exhibit. Subsequent research [Rag+17; Mon17; Aro+18; STR18; CGR22; Wan22; GEU24] derived tighter upper bounds and also provided lower bounds for the worst case and also considered new activation functions and architectures. Beyond worst-case behavior, the analysis of [HR19b; HR19a] showed that, on average, the number of linear regions observed over a one-dimensional line in the input space only grows linearly and not exponentially with the number of neurons. The properties of linear regions, such as the radius of the largest contained sphere, are studied in [ZW20]. The authors show that training with dropout or batch normalization considerably influences these properties, even when the classification accuracy does not change. While polytopes can be relatively complex shapes, the polytopes that make up the linear regions of trained DNNs are comparatively simple [Fan+23]. Concretely, the average number of faces of these polytopes is bounded by the input dimension and not by the depth of the network. In [Hin21], the authors examine how linear regions are characterized by the state of all ReLUs in a neural network (also known as *activation patterns* or *ReLU configurations*), essentially providing a global decomposition.

9.2 Neural Network Verification

Neural network verification is the task of proving or disproving the formula

$$\vec{x} \in \mathscr{C} \implies \mathscr{P}(h(\vec{x}))$$

where h is a DNN, $\mathscr{C}$ a (convex) precondition, and $\mathscr{P}$ a (piecewise) linear property. In this field, the branch-and-bound paradigm is often applied [Bun+20]. To improve efficiency, abstract interpretation domains, such as boxes, zonotopes, polyhedra, or star sets [Geh+18; Tra+19], can be used for bounding subproblems. When the result is inconclusive, the subproblem is refined by branching. In the case of PWL DNNs, branching can be naturally performed along the region boundaries of the activation functions. For example, ReLU can be split into the regions $(-\infty, 0]$ and $(0, \infty)$, on which it is linear and no longer contributes to the error during bounding. In the worst case, branching occurs for all activation functions of the DNN, which resembles the splitting done in an `AffTree`. However, the actual tree structure is usually not directly stored in neural network verifiers. Popular tools that follow this schema are nnenum [Bak21], α, β-CROWN [Shi+24], ERAN [Li+20], and Marabou [Kat+19] based on its integration of DeepPoly [Sin+19]. For an overview of their strengths and weaknesses, see their scoring at the annual VNN COMP [Bri+23; Bri+24]. The bounding procedure improves performance over a complete decomposition, although bounding after every branch is also inefficient. Thus, heuristics guide when and where to branch in the network, and how many activation functions are branched at the same time [Bak21]. As the running time of the verification of neural networks often depends on the number of their linear regions, regularization can be used during training to reduce the number of regions while (hopefully) not diminishing overall performance [BM22].

9.3 Decomposition for Explainability

In [Chu+18], the authors present an approach that decomposes neural networks along their activation patterns into their linear components. They specifically study the ReLU activation. In [Sud+20], ReLU DNNs are decomposed into their linear regions for interpretability and diagnostics. The authors introduce linear profile plots for analyzing the linear functions of these regions. In [Ayt22], additionally to FNNs, now also CNNs and RNNs are decomposed into their linear components. The result is stored in a semantically-equivalent tree structure for explainability purposes. Redundant predicates are removed from the trees. In [LJ20], the authors show the equivalence of oblique decision trees (ones were the predicates are linear inequalities over input features) and ReLU neural networks. They use this connection to train oblique decision trees with gradient descent, opening them to gradient-based optimization techniques such as DropConnect. Also, they find the representation in form of neural networks to be exponentially more efficient than a traditional tree representation. In [NKA20], FNNs with ReLU activation are translated into decision trees to increase interpretability. They also present an improved version of the C-Net algorithm to extract decision rules from trained DNNs. The authors of [BB20] apply max-affine spline operators from approximation theory to represent

FNNs, CNNs, and RNNs. In the case of neural controllers, rule or tree extraction is especially interesting [LV23]. The PWL structure of DNNs can also be used for reversing engineering [RK20]. From the way the bounding hyperplanes of the underlying function bent one can deduce how deep they occur within the network, allowing to (re)construct a DNN from a PWL function.

A visual analysis for CNNs is presented in [Jia+20], where multiple visualizations are provided based on the fully connected classifier at the end of CNNs. These visualizations are computed based on a surrogate decision tree that is distilled based on rule extraction methods. Its inputs, which are the result of previous convolution and pooling layers, are converted to human concepts based on network dissection methods.

9.4 Tool Support

Gopinath et. al. [Gop+18] present a translation of neural networks into imperative programs, to which one can apply classic symbolic and concolic execution. They present their method on a CNN, where they use concolic execution to analyze the effect of changing individual pixels in the input.

The tool DeepConcolic [Sun+18b; Sun+18a] uses symbolic and concrete execution to derive interesting inputs $\mathscr{T}$ with respect to a formal metric, called coverage metric. Examples include Lipschitz Continuity, Neuron Coverage, Sign Sign Coverage, and Neuron Boundary Coverage. The set of points is iteratively increased based on heuristics and symbolic execution.

The tool NeuroSPF [Usm+21] translates a DNN (FNN or CNN) into an equivalent Java program, where the primitives, such as convolution, are hardcoded with nested for loops, while the weights can be loaded. The Java program is then be analyzed with the Symbolic Path Finder (SPF), which supports both concolic and symbolic execution. To demonstrate the tool, the authors fix all but a central pixel in an MNIST image and show how NeuroSPF finds adversarial examples by modifying this pixel.

The tool SplineCam [Hum+23] is specialized for the visualization of DNNs (FNN or CNN) by using a graph-based domain for representing the preimage partition. Instead for storing the faces (i.e., hyperplanes) of the polytopes of the input partition, they store the vertices. Therefore, the approach scales exponentially with the input dimension of the network. However, when analyzing the network for two-dimensional slices of the input space, the method is fast, making it well suited for visualizations.

CHAPTER 10

Conclusion and Outlook

This chapter summarizes the results of this thesis in Section 10.1. Finally, Section 10.2 provides an outlook into future research for the presented techniques.

10.1 Conclusion

In this thesis, a faithful and modular approach for the distillation of deep neural networks into decision trees was presented. To achieve this, a sequence of intermediate languages was defined, each optimized for certain aspects of the pipeline. At the start of this sequence stands `Afflang`, which bundles many different DL primitives into a single, minimal, and imperative language. Through explicit constructions, it was shown how popular PWL operators of the ONNX framework can be encoded in `Afflang`. In this way, activation functions, such as ReLU, Leaky ReLU, Abs, and Hard Sigmoid, were encoded in `Afflang`, as well as other primitives, such as the argmax function, concatenation, convolution, max pooling, padding, and residual connections. Implementing additional PWL primitives is straightforward in `Afflang`. It was also briefly mentioned how non-PWL functions can be approximated by PWL functions, as they are universal function approximators. A concise analysis of `Afflang` laid out its formal properties, like its expressiveness, continuity, and type correctness.

Building on this, a calculus of SOS rules was defined that implements symbolic execution for `Afflang` based on lifting vector and tensor operations to operations over affine functions. The result of applying symbolic execution is a CFG that is in this specific case a finite tree. By refining this finite model of the complete semantics of the `Afflang` program—and therefore also of DNNs that can be encoded in `Afflang`—one obtains a concise model that builds the foundation for further analysis.

Concolic execution and preconditions can both be applied to `Afflang` by using affine functions and leveraging function composition and infeasible path elimination.

The resulting tree, called `AffTree`, is equipped with a suit of algebraic operations, such as function composition, addition, and equality, by selectively applying the theory of ADDs to the decision tree. In this way, `AffTrees` form an efficient and modular model to represent the semantics of PWL DNNs. Applying a technique

from program analysis, called infeasible path elimination, ensures that redundancies are eliminated in `AffTrees`.

In the resulting framework, a lookup table is created from DL primitives to `AffTrees`, where the effect of the intermediate language `Afflang` is hardcoded. In this way, at runtime one does no longer need to translate the network multiple times, but has instead a direct mapping. Through the decomposition into essential building blocks, the compositional and algebraic properties of the framework can be fully utilized, and through the built-in optimizations, the process can identify and prune redundancies early on. This framework is implemented in the tool `Affinitree`, which is available for Python and Rust.

For the purposes of explainability, `AffTrees` are a FISM that provides explanatory information for trained DNNs. In the experiments, it was shown how `AffTrees` can be employed as a post-hoc explainability method to either explain individual outcomes or the complete model in contexts like fairness or adversarial robustness.

Although the method works well for small networks, scaling to larger networks is naturally more challenging. The application of formal and rigorous methods is always computationally more involved than approximate solutions, and like normal symbolic execution, this approach also suffers from the *state-space-explosion* problem. In the case of DNNs, it was proven that their verification is NP-complete [Kat+17], highlighting the scalability limits of this approach. On the one hand, on average, the number of linear regions that trained DNNs exhibit is far below the worst case, providing an opportunity for optimization. On the other hand, even with an efficient data structure and infeasible path elimination, the process of identifying and pruning infeasible activation patterns requires solving many LPs, which can be expensive when the network gets larger.

Overall, this thesis provides a crucial contribution for the endeavor of "opening the black box" in the context of DNNs. The initial goal of providing the complete semantic function of otherwise opaque DNNs in an accessible and efficient data structure was achieved. The semantic functions of otherwise opaque DNNs can be distilled into faithful surrogate models, allowing further analysis in a traceable way. The efficiency and modularity of the approach provides a foundational framework that invites future researchers to build their own analysis on top of it.

10.2 Future Work

The presented framework opens up many new possibilities for the explanation of DNNs. In the case of fairness, the algebraic properties were used to evaluate the behavior of the model under different values of protected attributes. Going further, it would be interesting to see how other formal concepts can be applied to `AffTrees` for explainability. Preliminary experiments showed already that notions like the Lipschitz constant can be applied to `AffTrees`. When $L(\cdot)$ is the Lipschitz constant of a continuous function, then the following identity provides a simple upper bound:

$$L(f \circ g) \leq L(f) \cdot L(g)$$

This technique can be directly applied to the distillation of `AffTrees`, which also uses function composition. In this way, the upper bound can be refined as needed.

Another such formal notion is based on violations of injectivity. In [Mon17], a crucial observation was made that the layers of DNNs successively fold their input space such that it becomes easier to separate differently labeled samples. With the sequential composition of `AffTrees`, it is possible to compute exactly which inputs are mapped to the same output by a layer, essentially violating injectivity. Computing these allows a rigorous definition of what inputs the DNN identifies as equivalent, or more precisely, as indistinguishable. Their computation is already possible, but additional research is needed to filter and visualize them in a comprehensible manner.

It would also be interesting to apply logic explanations, such as abductive and inflated explanations, in the context of `AffTrees`. In [MSS24a; MSS24b], we applied logic explanations to Random Forest (RF) and boosted tree models. There, the forests are also distilled into a single ADD or tree model that is semantically equivalent to the original ensemble. Based in this distilled model, logic explanations can be computed using graph algorithms and interval propagation. In principle, the same methods can be applied to `AffTrees`, however, as the predicates are more complex (not axis-aligned), the method must be adapted. In this way, aspects of the full `AffTree` could be reduced into a comprehensible but sound logic explanation.

In similar (branch-and-bound) verification approaches, the use of abstract interpretation and CEGAR-like refinement loops plays an essential role in their current performance [Bak21]. Thinking this further, possible future work might discover abstract interpretation domains that are suitable in the context of explainability. It could be that for certain desiderata domains exists that are fully abstract, i.e., being as precise as necessary for checking the desiderata satisfaction but not anymore than that. Discovering such domains would increase scalability considerably.

Future research might explore how these explanatory information can be processed in a way that they increase understanding in stakeholders. Specifically for stakeholders with little prior knowledge, forms of abstraction and visualization are required to increase their accessibility. For a complete framework that includes explanations tailored to the needs of different stakeholders, domain experts need to be part of the solution. For finding reductions and abstractions from the mathematical nature of `AffTrees` and their formal properties, researchers from psychology and philosophy can help.

List of Figures

Acronyms

ADD algebraic decision diagram. 9, 28, 83, 93, 99, 123, 125

AI artificial intelligence. 1–6, 10, 104

BNF Backus–Naur Form. 35, 41

BSP binary space partitioning. 9

CFG control flow graph. 11, 33, 73, 84–86, 93, 123

CNN convolutional neural network. 11, 23–25, 27, 32, 35, 53, 63, 72, 101, 121, 122

DAG directed acyclic graph. 9, 28

DL deep learning. 2, 3, 8, 10, 11, 22, 23, 34, 35, 42, 53, 54, 63, 68, 73, 120, 123, 124

DNN deep neural network. iii, 2, 3, 5–11, 23, 32–34, 36, 41, 44, 45, 53, 54, 60–62, 71, 73, 74, 81, 103–111, 113–115, 120–128

FISM faithful yet interpretable surrogate model. 8, 11, 33, 124

FNN feed forward neural network. 11, 24, 32, 41, 53, 54, 56, 57, 61–63, 121, 122, 127

FPR false positive rate. 110

LLM large language model. 1

LP linear program. 93–97, 124

LSTM long short-term memory. 71

ML machine learning. 2, 4, 5, 8, 23, 24, 81, 82, 119

ONNX Open Neural Network Exchange. 11, 32, 35, 41, 53, 71, 72, 123, 126

PCA Principal Component Analysis. 80, 115

PR positive rate. 104, 110

PWL piecewise linear. 7–9, 11, 21, 22, 31, 33, 34, 36, 39, 40, 43–47, 49, 50, 53–55, 57–60, 72–74, 76, 83, 85–88, 91, 92, 99, 120–123, 126, 127

RF Random Forest. 5, 125

RNN recurrent neural network. 11, 27, 28, 32, 35, 53, 70, 121, 122

SOS structural operational semantics. 35, 37–43, 74, 77, 84, 123, 127

SVM Support Vector Machine. 5

TPR true positive rate. 109, 110

XAI explainable AI. 4–6, 8, 119, 126

Bibliography

[AB18] Amina Adadi and Mohammed Berrada. „Peeking inside the black-box: a survey on explainable artificial intelligence (XAI)“. In: *IEEE access* 6 (2018), pp. 52138–52160.

[Ade+18] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian J. Goodfellow, Moritz Hardt, and Been Kim. „Sanity Checks for Saliency Maps“. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. Vol. 31. New York, NY, USA: Curran Associates, Inc., 2018, pp. 9525–9536. URL: https://proceedings.neurips.cc/paper/2018/hash/294a8ed24b1ad22ec2e7efea049b8737-Abstract.html.

[AM18] Naveed Akhtar and Ajmal Mian. „Threat of adversarial attacks on deep learning in computer vision: A survey“. In: *Ieee Access* 6 (2018), pp. 14410–14430.

[Ang+16] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. *Machine Bias: There's software used across the country to predict future criminals. And it's biased against blacks.* May 2016. URL: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.

[Aro+18] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit 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. URL: https://openreview.net/forum?id=B1J%5C_rgWRW.

[Arr+20] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. „Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI“. In: *Information fusion* 58 (2020), pp. 82–115.

[Asi50] Isaac Asimov. *I, Robot*. Vol. 1. Gnome Press, 1950.

[Axl97] Sheldon Axler. *Linear algebra done right*. Springer Science & Business Media, 1997.

[Ayt22] Caglar Aytekin. „Neural Networks are Decision Trees“. In: *arXiv preprint arXiv:2210.05189* (2022).

[Bac+15] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. „On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation“. In: *PloS one* 10.7 (2015), e0130140.

[Bae+10] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert Müller. „How to explain individual classification decisions“. In: *The Journal of Machine Learning Research* 11 (2010), pp. 1803–1831.

[Bak21] Stanley 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*. Springer. 2021, pp. 19–36.

[BB20] Randall Balestriero and Richard G Baraniuk. „Mad max: Affine spline insights into deep learning“. In: *Proceedings of the IEEE* 109.5 (2020), pp. 704–727.

[BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. „Representation Learning: A Review and New Perspectives“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (2013), pp. 1798–1828. DOI: 10.1109/TPAMI.2013.50.

[BDD23] Maarten Buyl, MaryBeth Defrance, and Tijl De Bie. „fairret: a Framework for Differentiable Fairness Regularization Terms“. In: *arXiv preprint arXiv:2310.17256* (2023).

[Ber+21] Richard Berk, Hoda Heidari, Shahin Jabbari, Michael Kearns, and Aaron Roth. „Fairness in criminal justice risk assessments: The state of the art“. In: *Sociological Methods & Research* 50.1 (2021), pp. 3–44.

[BKS17] Kevin Baum, Maximilian A Köhl, and Eva Schmidt. „Two challenges for CI trustworthiness and how to address them“. In: *Proceedings of the 1st Workshop on Explainable Computational Intelligence (XCI 2017)*. 2017.

[BKS95] Sven G. Bartels, Ludwig Kuntz, and Stefan Scholtes. „Continuous selections of linear functions and nonsmooth critical point theory“. In: *Nonlinear Analysis: Theory, Methods & Applications* 24.3 (1995), pp. 385–407. ISSN: 0362-546X. DOI: https://doi.org/10.1016/0362-546X(95)91645-6.

[BM22] Benedikt Böing and Emmanuel Müller. „On Training and Verifying Robust Autoencoders“. In: *2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)*. 2022, pp. 1–10. DOI: 10.1109/DSAA54385.2022.10032334.

[Bor23] Georg Borges. „Liability for AI Systems Under Current and Future Law: An overview of the key changes envisioned by the proposal of an EU-directive on liability for AI“. In: *Computer Law Review International* 24.1 (2023), pp. 1–8.

[Bri+23] Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T Johnson, and Changliu Liu. „First three years of the international verification of neural networks competition (VNN-COMP)“. In: *International Journal on Software Tools for Technology Transfer* 25.3 (2023), pp. 329–339.

[Bri+24] Christopher Brix, Stanley Bak, Taylor T Johnson, and Haoze Wu. „The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results“. In: *arXiv preprint arXiv:2412.19985* (2024).

[Bro+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. „Language Models are Few-Shot Learners“. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. Vol. 33. New York, NY, USA: Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[Bro83] Arne Brondsted. *An Introduction to Convex Polytopes*. First. Springer, New York, NY, 1983. DOI: https://doi.org/10.1007/978-1-4612-1148-8.

[Bry86] Bryant. „Graph-Based Algorithms for Boolean Function Manipulation“. In: *IEEE Transactions on Computers* 35.8 (Aug. 1986), pp. 677–691. ISSN: 1557-9956. DOI: 10.1109/TC.1986.1676819.

[BS16] Solon Barocas and Andrew D Selbst. „Big data's disparate impact“. In: *Calif. L. Rev.* 104 (2016), p. 671.

[Buc05] Bruce G Buchanan. „A (very) brief history of artificial intelligence“. In: *Ai Magazine* 26.4 (2005), pp. 53–53.

[Bun+20] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. „Branch and bound for piecewise linear neural network verification“. In: *Journal of Machine Learning Research* 21.42 (2020), pp. 1–39.

[Bur16] Jenna Burrell. „How the machine 'thinks': Understanding opacity in machine learning algorithms“. In: *Big data & society* 3.1 (2016).

[Cad+08] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. „EXE: Automatically generating inputs of death“. In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), pp. 1–38.

[CGR22] Kuan-Lin Chen, Harinath Garudadri, and Bhaskar D Rao. „Improved bounds on neural complexity for representing piecewise linear functions“. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 7167–7180.

[Cha+19] Nandish Chattopadhyay, Anupam Chattopadhyay, Sourav Sen Gupta, and Michael Kasper. „Curse of dimensionality in adversarial examples“. In: *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2019, pp. 1–8.

[CHH02] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. „Deep blue“. In: *Artificial intelligence* 134.1-2 (2002), pp. 57–83.

[Chu+18] Lingyang Chu, Xia Hu, Juhua Hu, Lanjun Wang, and Jian 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*. 2018, pp. 1244–1253.

[Cor+17] Sam Corbett-Davies, Emma Pierson, Avi Feller, Sharad Goel, and Aziz Huq. „Algorithmic decision making and the cost of fairness“. In: *Proceedings of the 23rd acm sigkdd international conference on knowledge discovery and data mining*. 2017, pp. 797–806.

[CSS23] Barnaby Crook, Maximilian Schlüter, and Timo Speith. „Revisiting the Performance-Explainability Trade-Off in Explainable Artificial Intelligence (XAI)“. In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. Sept. 2023, pp. 316–324. DOI: 10.1109/REW57809.2023.00060.

[Cub+17] Ekin D Cubuk, Barret Zoph, Samuel S Schoenholz, and Quoc V Le. „Intriguing properties of adversarial examples“. In: *arXiv preprint arXiv:1711.02846* (2017).

[Cur+58] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*. Vol. 1. North-Holland Amsterdam, 1958.

[CW17] Nicholas Carlini and David Wagner. „Towards evaluating the robustness of neural networks“. In: *2017 ieee symposium on security and privacy (sp)*. Ieee. 2017, pp. 39–57.

[Cyb89] George Cybenko. „Approximation by superpositions of a sigmoidal function“. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[Dim+20] Botty Dimanov, Umang Bhatt, Mateja Jamnik, and Adrian Weller. „You shouldn't trust me: Learning models which conceal unfairness from multiple explanation methods.“ In: (2020).

[Din+21] Frances Ding, Moritz Hardt, John Miller, and Ludwig Schmidt. „Retiring adult: New datasets for fair machine learning“. In: *Advances in neural information processing systems* 34 (2021), pp. 6478–6490.

[Dre18] Quantic Dream. *Detroit: Become Human*. PlayStation 4, Microsoft Windows. 2018. URL: https://www.quanticdream.com/en/detroit-become-human.

[Dwo+12] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. „Fairness through awareness“. In: *Proceedings of the 3rd innovations in theoretical computer science conference*. 2012, pp. 214–226.

[Erh+09] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. „Visualizing higher-layer features of a deep network“. In: *University of Montreal* 1341.3 (2009), p. 1.

[Fan+23] Feng-Lei Fan, Wei Huang, Xiangru Zhong, Lecheng Ruan, Tieyong Zeng, Huan Xiong, and Fei Wang. „Deep relu networks have surprisingly simple polytopes“. In: *arXiv preprint arXiv:2305.09145* (2023).

[Far+12] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. „Learning hierarchical features for scene labeling“. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2012), pp. 1915–1929.

[FC18] Jonathan Frankle and Michael Carbin. „The lottery ticket hypothesis: Finding sparse, trainable neural networks“. In: *arXiv preprint arXiv:1803.03635* (2018).

[Flo+18] Luciano Floridi, Josh Cowls, Monica Beltrametti, Raja Chatila, Patrice Chazerand, Virginia Dignum, Christoph Luetge, Robert Madelin, Ugo Pagallo, Francesca Rossi, et al. „AI4People—an ethical framework for a good AI society: opportunities, risks, principles, and recommendations“. In: *Minds and machines* 28 (2018), pp. 689–707.

[FT21] Alessandro Facchini and Alberto Termine. „Towards a Taxonomy for the Opacity of AI Systems“. In: *Conference on Philosophy and Theory of Artificial Intelligence*. Springer. 2021, pp. 73–89.

[FWC85] Randall Frakes, Bill Wisher, and James Cameron. *The Terminator*. 1985.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Geh+18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. „Ai2: Safety and robustness certification of neural networks with abstract interpretation“. In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 3–18.

[GEU24] Alexis Goujon, Arian Etemadi, and Michael Unser. „On the number of regions of piecewise linear neural networks“. In: *Journal of Computational and Applied Mathematics* 441 (2024), p. 115667.

[GF17] Bryce Goodman and Seth Flaxman. „European Union regulations on algorithmic decision-making and a “right to explanation”“. In: *AI magazine* 38.3 (2017), pp. 50–57.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. „DART: Directed automated random testing“. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[God+23] Charles Godfrey, Henry Kvinge, Elise Bishoff, Myles Mckay, Davis Brown, Tim Doster, and Eleanor Byler. „How many dimensions are required to find an adversarial example?“ In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 2353–2360.

[Goo+13] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. „Multi-digit number recognition from street view imagery using deep convolutional neural networks“. In: *arXiv preprint arXiv:1312.6082* (2013).

[Gop+18] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S Pasareanu, and Sarfraz Khurshid. „Symbolic execution for deep neural networks“. In: *arXiv preprint arXiv:1807.10439* (2018).

[GS21] Frederik Gossen and Bernhard Steffen. „Algebraic aggregation of random forests: towards explainability and rapid evaluation“. In: *International Journal on Software Tools for Technology Transfer* (2021). ISSN: 1433-2787. DOI: 10.1007/s10009-021-00635-x.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. „Explaining and harnessing adversarial examples“. In: *arXiv preprint arXiv:1412.6572* (2014).

[Gui+18] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. „A Survey of Methods for Explaining Black Box Models“. In: *ACM Comput. Surv.* 51.5 (Aug. 2018). ISSN: 0360-0300. DOI: 10.1145/3236009.

[GZB94] Valentin V Gorokhovik, Oleg I Zorko, and G Birkhoff. „Piecewise affine functions and polyhedral sets“. In: *Optimization* 31.3 (1994), pp. 209–221.

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[Hin+12] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. „Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups“. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[Hin21] Peter Hinz. „Using activation histograms to bound the number of affine regions in ReLU feed-forward neural networks“. In: *ArXiv* abs/2103.17174 (2021).

[Hin86] Geoffrey E Hinton. „Learning distributed representations of concepts“. In: *Proceedings of the Annual Meeting of the Cognitive Science Society*. Vol. 8. 1986.

[HLS13] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. John Wiley & Sons, 2013.

[HPS16] Moritz Hardt, Eric Price, and Nati Srebro. „Equality of opportunity in supervised learning“. In: *Advances in neural information processing systems* 29 (2016).

[HR19a] Boris Hanin and David Rolnick. „Complexity of Linear Regions in Deep Networks“. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, June 2019, pp. 2596–2604. URL: https://proceedings.mlr.press/v97/hanin19a.html.

[HR19b] Boris Hanin and David Rolnick. „Deep relu networks have surprisingly few activation patterns“. In: *Advances in neural information processing systems* 32 (2019).

[HRZ17] Martin Henk, Jürgen Richter-Gebert, and Günter M Ziegler. „Basic properties of convex polytopes“. In: *Handbook of discrete and computational geometry*. Chapman and Hall/CRC, 2017, pp. 383–413.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. „Multilayer feedforward networks are universal approximators“. In: *Neural networks* 2.5 (1989), pp. 359–366.

[Hul+23] Adam Hulman, Ole Lindgård Dollerup, Jesper Friis Mortensen, Matthew E Fenech, Kasper Norman, Henrik Støvring, and Troels Krarup Hansen. „ChatGPT-versus human-generated answers to frequently asked questions about diabetes: A Turing test-inspired survey among employees of a Danish diabetes center“. In: *Plos one* 18.8 (2023), e0290773.

[Hum+23] Ahmed Imtiaz Humayun, Randall Balestriero, Guha Balakrishnan, and Richard G Baraniuk. „Splinecam: Exact visualization and characterization of deep network geometry and decision boundaries“. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 3789–3798.

[Hüt10] Hans Hüttel. *Transitions and trees: an introduction to structural operational semantics*. Cambridge University Press, 2010.

[INM19] Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. „On validating, repairing and refining heuristic ML explanations“. In: *arXiv preprint arXiv:1907.02509* (2019).

[Iof15] Sergey Ioffe. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *arXiv preprint arXiv:1502.03167* (2015).

[Jia+20] Shichao Jia, Peiwen Lin, Zeyu Li, Jiawan Zhang, and Shixia Liu. „Visualizing surrogate decision trees of convolutional neural networks“. In: *Journal of Visualization* 23 (2020), pp. 141–156.

[Jud20] Thomas W Judson. *Abstract algebra: theory and applications*. 2020.

[Kat+17] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. „Reluplex: An efficient SMT solver for verifying deep neural networks“. In: *International conference on computer aided verification*. Springer. 2017, pp. 97–117.

[Kat+19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. „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*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer. Springer, 2019, pp. 443–452. DOI: 10.1007/978-3-030-25540-4_26.

[KB96] Ronny Kohavi and Barry Becker. „Uci adult data set“. In: *UCI Meachine Learning Repository* 5 (1996).

[KC19] Puneet Kohli and Anjali Chadha. „Enabling pedestrian safety using computer vision techniques: A case study of the 2018 uber inc. self-driving car crash“. In: *Future of information and communication conference*. Springer. 2019, pp. 261–279.

[KGB18] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio. „Adversarial examples in the physical world“. In: *Artificial intelligence safety and security*. Chapman and Hall/CRC, 2018, pp. 99–112.

[Kin76] James C King. „Symbolic execution and program testing“. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[Kli20] Marc-Uwe Kling. *QualityLand 2.0: Kikis Geheimnis*. Berlin: Ullstein, 2020. ISBN: 978-3-550-20102-8.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems* 25 (2012).

[KVB88] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. „Design of an image edge detection filter using the Sobel operator“. In: *IEEE Journal of solid-state circuits* 23.2 (1988), pp. 358–367.

[Lan+21] Markus Langer, Daniel Oster, Timo Speith, Holger Hermanns, Lena Kästner, Eva Schmidt, Andreas Sesing, and Kevin Baum. „What do we want from Explainable Artificial Intelligence (XAI)?–A stakeholder perspective on XAI and a conceptual model guiding interdisciplinary XAI research“. In: *Artificial Intelligence* 296 (2021), p. 103473.

[LB20] Himabindu Lakkaraju and Osbert Bastani. „“How do I fool you?” Manipulating User Trust via Misleading Black Box Explanations“. In: *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. 2020, pp. 79–85.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. „Deep learning“. In: *nature* 521.7553 (2015), pp. 436–444.

[LeC+98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Les+93] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. „Multilayer feedforward networks with a nonpolynomial activation function can approximate any function“. In: *Neural networks* 6.6 (1993), pp. 861–867.

[Li+20] Renjue Li, Jianlin Li, Cheng-Chao Huang, Pengfei Yang, Xiaowei Huang, Lijun Zhang, Bai Xue, and Holger Hermanns. „Prodeep: a platform for robustness verification of deep neural networks“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1630–1634.

[Lip18] Zachary C Lipton. „The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.“ In: *Queue* 16.3 (2018), pp. 31–57.

[LJ20] Guang-He Lee and Tommi S. Jaakkola. „Oblique Decision Trees from Derivatives of ReLU Networks“. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=Bke8UR4FPB.

[LL17] Scott M Lundberg and Su-In Lee. „A unified approach to interpreting model predictions“. In: *Advances in neural information processing systems* 30 (2017).

[Loh+22] Michael Lohaus, Matthäus Kleindessner, Krishnaram Kenthapadi, Francesco Locatello, and Chris Russell. „Are two heads the same as one? Identifying disparate treatment in fair neural networks“. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16548–16562.

[Lov42] Ada Lovelace. „Notes upon LF Menabrea's sketch of the analytical engine invented by Charles Babbage“. In: *Bibliothèque Universelle de Genève* 82 (1842), pp. 245–295.

[LV23] Torben Logemann and Eric MSP Veith. „NN2EQCDT: Equivalent Transformation of Feed-Forward Neural Networks as DRL Policies into Compressed Decision Trees“. In: *vol* 15 (2023), pp. 94–100.

[Mag22] Robert Magnus. „Metric Spaces“. In: *Metric Spaces*. Springer, 2022, pp. 1–27.

[Meh+21] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. „A survey on bias and fairness in machine learning“. In: *ACM computing surveys (CSUR)* 54.6 (2021), pp. 1–35.

[Mei+24] Qiaozhu Mei, Yutong Xie, Walter Yuan, and Matthew O Jackson. „A Turing test of whether AI chatbots are behaviorally similar to humans“. In: *Proceedings of the National Academy of Sciences* 121.9 (2024), e2313925121.

[MI22] Joao Marques-Silva and Alexey Ignatiev. „Delivering Trustworthy AI through formal XAI". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 11. 2022, pp. 12342–12350.

[Miy+18] Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, and Shin Ishii. „Virtual adversarial training: a regularization method for supervised and semi-supervised learning". In: *IEEE transactions on pattern analysis and machine intelligence* 41.8 (2018), pp. 1979–1993.

[Mol25] Christoph Molnar. *Interpretable machine learning. A Guide for Making Black Box Models Explainable*. 3rd ed. 2025. ISBN: 978-3-911578-03-5. URL: https://christophm.github.io/interpretable-ml-book.

[Mon+14] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. „On the number of linear regions of deep neural networks". In: *Advances in neural information processing systems* 27 (2014).

[Mon+19] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. „Layer-wise relevance propagation: an overview". In: *Explainable AI: interpreting, explaining and visualizing deep learning* (2019), pp. 193–209.

[Mon17] Guido Montúfar. „Notes on the number of linear regions of deep neural networks". In: (2017).

[MR79] Etienne Morel and Claude Renvoise. „Global optimization by suppression of partial redundancies". In: *Communications of the ACM* 22.2 (1979), pp. 96–103.

[MSS24a] Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. „Computing Inflated Explanations for Boosted Trees: A Compilation-Based Approach". In: *The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*. Ed. by Mike Hinchey and Bernhard Steffen. Cham: Springer Nature Switzerland, Oct. 2024, pp. 183–201. ISBN: 978-3-031-73887-6. DOI: 10.1007/978-3-031-73887-6_14.

[MSS24b] Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. „Voting-Based Shortcuts through Random Forests for Obtaining Explainable Models". In: *Real Time and Such: Essays Dedicated to Wang Yi to Celebrate His Scientific Career*. Ed. by Susanne Graf, Paul Pettersson, and Bernhard Steffen. Cham: Springer Nature Switzerland, Oct. 2024, pp. 135–153. ISBN: 978-3-031-73751-0. DOI: 10.1007/978-3-031-73751-0_11.

[MSS25] Alnis Murtovi, Maximilian Schlüter, and Bernhard Steffen. „An Efficient Compilation-based Approach to Explaining Random Forests Through Decision Trees". In: *Proceedings of the 17th International Conference on Agents and Artificial Intelligence, ICAART 2025*. Vol. 2. SCITEPRESS, 2025, pp. 484–495. DOI: 10.5220/0013188600003890.

[Mur+23] Alnis Murtovi, Alexander Bainczyk, Gerrit Nolte, Maximilian Schlüter, and Bernhard Steffen. „Forest GUMP: a tool for verification and explanation“. In: *International Journal on Software Tools for Technology Transfer* (May 2023). ISSN: 1433-2787. DOI: 10.1007/s10009-023-00702-5.

[NKA20] Tung D Nguyen, Kathryn E Kasmarik, and Hussein A Abbass. „An exact transformation from deep neural networks to multi-class multivariate decision trees“. In: *arXiv preprint arXiv:2003.04675* (2020).

[Nol+23] Gerrit Nolte, Maximilian Schlüter, Alnis Murtovi, and Bernhard Steffen. „The Power of Typed Affine Decision Structures: A Case Study“. In: *International Journal on Software Tools for Technology Transfer* (Apr. 2023). ISSN: 1433-2787. DOI: 10.1007/s10009-023-00701-6.

[NS16] Sven Nyholm and Jilles Smids. „The ethics of accident-algorithms for self-driving cars: An applied trolley problem?“ In: *Ethical theory and moral practice* 19.5 (2016), pp. 1275–1289.

[NSM23] Oded Nov, Nina Singh, and Devin Mann. „Putting ChatGPT's medical advice to the (Turing) test: survey study“. In: *JMIR Medical Education* 9 (2023), e46939.

[NYC16] Anh Nguyen, Jason Yosinski, and Jeff Clune. „Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks“. In: *arXiv preprint arXiv:1602.03616* (2016).

[Olt+19] Alexandra Olteanu, Carlos Castillo, Fernando Diaz, and Emre Kıcıman. „Social data: Biases, methodological pitfalls, and ethical boundaries“. In: *Frontiers in big data* 2 (2019), p. 13.

[Ouy+22] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. „Training language models to follow instructions with human feedback, 2022“. In: *URL https://arxiv. org/abs/2203.02155* 13 (2022), p. 1.

[Ovc00] Sergei Ovchinnikov. „Max-min representation of piecewise linear functions“. In: *arXiv preprint math/0009026* (2000).

[Ovc02] Sergei Ovchinnikov. „Max-Min Representation of Piecewise Linear Functions“. In: *Beiträge zur Algebra und Geometrie / Contributions to Algebra and Geometry* 43.1 (2002), pp. 297–302.

[Oxf] Oxford University Press. *artificial intelligence. Oxford English Dictionary*. Accessed January 27, 2025. URL: https://www.oed.com/dictionary/artificial-intelligence_n.

[Par+22] Darsh Parekh, Nishi Poddar, Aakash Rajpurkar, Manisha Chahal, Neeraj Kumar, Gyanendra Prasad Joshi, and Woong Cho. „A review on autonomous vehicles: Progress, methods and challenges“. In: *Electronics* 11.14 (2022), p. 2162.

[Pau+21] Patricia Pauli, Anne Koch, Julian Berberich, Paul Kohler, and Frank Allgöwer. „Training robust neural networks using Lipschitz bounds“. In: *IEEE Control Systems Letters* 6 (2021), pp. 121–126.

[Plo04] Gordon D. Plotkin. „A structural approach to operational semantics“. In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139.

[PMB13] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. „On the number of response regions of deep feed forward networks with piece-wise linear activations“. In: *arXiv preprint arXiv:1312.6098* (2013).

[PS23] Dana Pessach and Erez Shmueli. „Algorithmic fairness“. In: *Machine Learning for Data Science Handbook: Data Mining and Knowledge Discovery Handbook*. Springer, 2023, pp. 867–886.

[Rag+17] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. „On the expressive power of deep neural networks“. In: *international conference on machine learning*. PMLR. 2017, pp. 2847–2854.

[RK20] David Rolnick and Konrad Kording. „Reverse-engineering deep relu networks“. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 8178–8187.

[Ros58] Frank Rosenblatt. „The perceptron: a probabilistic model for information storage and organization in the brain.“ In: *Psychological review* 65.6 (1958), p. 386.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos 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*. 2016, pp. 1135–1144.

[Rud19] Cynthia Rudin. „Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead“. In: *Nature machine intelligence* 1.5 (2019), pp. 206–215.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. „Swish: a self-gated activation function“. In: *arXiv preprint arXiv:1710.05941* 7.1 (2017), p. 5.

[Sch+23] Maximilian Schlüter, Gerrit Nolte, Alnis Murtovi, and Bernhard Steffen. „Towards rigorous understanding of neural networks via semantics-preserving transformations“. In: *International Journal on Software Tools for Technology Transfer* (May 2023). ISSN: 1433-2787. DOI: 10.1007/s10009-023-00700-7.

[Sel+17] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. „Grad-cam: Visual explanations from deep networks via gradient-based localization“. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 618–626.

[Sha+18] Ali Shafahi, W Ronny Huang, Christoph Studer, Soheil Feizi, and Tom Goldstein. „Are adversarial examples inevitable?“ In: *arXiv preprint arXiv:1809.02104* (2018).

[Shi+24] Zhouxing Shi, Qirui Jin, Zico Kolter, Suman Jana, Cho-Jui Hsieh, and Huan Zhang. „Neural Network Verification with Branch-and-Bound for General Nonlinearities“. In: *arXiv preprint arXiv:2405.21063* (2024).

[Shr+16] Avanti Shrikumar, Peyton Greenside, Anna Shcherbina, and Anshul Kundaje. „Not just a black box: Learning important features through propagating activation differences“. In: *arXiv preprint arXiv:1605.01713* (2016).

[Sil+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. „A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play“. In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404.

[Sin+19] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. „An abstract domain for certifying neural networks“. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30.

[Sla+20] Dylan Slack, Sophie Hilgard, Emily Jia, Sameer Singh, and Himabindu Lakkaraju. „Fooling lime and shap: Adversarial attacks on post hoc explanation methods“. In: *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. 2020, pp. 180–186.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. „CUTE: A concolic unit testing engine for C“. In: *ACM SIGSOFT software engineering notes* 30.5 (2005), pp. 263–272.

[Smi+17] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. „Smoothgrad: removing noise by adding noise“. In: *arXiv preprint arXiv:1706.03825* (2017).

[SN23] Maximilian Schlüter and Gerrit Nolte. „Introduction to Symbolic Execution of Neural Networks-Towards Faithful and Explainable Surrogate Models“. In: *Electronic Communications of the EASST* 82 (Oct. 2023). DOI: 10.14279/tuj.eceasst.82.1225.

[Spe22] Timo Speith. „A review of taxonomies of explainable artificial intelligence (XAI) methods“. In: *Proceedings of the 2022 ACM conference on fairness, accountability, and transparency*. 2022, pp. 2239–2250.

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. „Dropout: a simple way to prevent neural networks from overfitting“. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[SS24] Maximilian Schlüter and Bernhard Steffen. „Affinitree: A Compositional Framework for Formal Analysis and Explanation of Deep Neural Networks“. In: *Tests and Proofs*. Ed. by Marieke Huisman and Falk Howar. Cham: Springer Nature Switzerland, Sept. 2024, pp. 148–167. ISBN: 978-3-031-72044-4. DOI: 10.1007/978-3-031-72044-4_8.

[Ste96] Bernhard Steffen. „Property-oriented expansion“. In: *Static Analysis: Third International Symposium, SAS'96 Aachen, Germany, September 24–26, 1996 Proceedings 3*. Springer. 1996, pp. 22–41.

[STR18] Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. „Bounding and counting linear regions of deep neural networks“. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4558–4566.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. „Axiomatic attribution for deep networks“. In: *International conference on machine learning*. PMLR. 2017, pp. 3319–3328.

[Sud+20] Agus Sudjianto, William Knauth, Rahul Singh, Zebin Yang, and Aijun Zhang. „Unwrapping the black box of deep ReLU networks: interpretability, diagnostics, and simplification“. In: *arXiv preprint arXiv:2011.04041* (2020).

[Sun+18a] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. „Testing deep neural networks“. In: *arXiv preprint arXiv:1803.04792* (2018).

[Sun+18b] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. „Concolic testing for deep neural networks“. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 109–119.

[Sut14] I Sutskever. „Sequence to Sequence Learning with Neural Networks“. In: *arXiv preprint arXiv:1409.3215* (2014).

[Sut19] Richard Sutton. „The bitter lesson“. In: *Incomplete Ideas (blog)* 13.1 (2019), p. 38.

[SVS24] Barbara Steffen, Moshe Vardi, and Bernhard Steffen. *Let's Talk AI: Interdisciplinarity Is a Must*. Vol. 15000. LNCS. To appear. Springer Nature, 2024.

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. „Deep inside convolutional networks: Visualising image classification models and saliency maps“. In: *arXiv preprint arXiv:1312.6034* (2013).

[SZ15] Gerard Sierksma and Yori Zwols. *Linear and integer optimization: theory and practice*. CRC Press, 2015.

[TKC22] Hajime Tasaki, Yuji Kaneko, and Jinhui Chao. „Curse of co-Dimensionality: Explaining Adversarial Examples by Embedding Geometry of Data Manifold“. In: *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE. 2022, pp. 2364–2370.

[TM99] JM Tarela and MV Martinez. „Region configurations for realizability of lattice piecewise-linear models“. In: *Mathematical and Computer Modelling* 30.11-12 (1999), pp. 17–27.

[Tra+19] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T Johnson. „Star-based reachability analysis of deep neural networks“. In: *International symposium on formal methods*. Springer. 2019, pp. 670–686.

[Tra+20] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T Johnson. „NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems“. In: *International Conference on Computer Aided Verification*. Springer. 2020, pp. 3–17.

[Tur50] Alan M Turing. „Computing Machinery and Intelligence“. In: *Mind* 59.246 (1950).

[Usm+21] Muhammad Usman, Yannic Noller, Corina S Păsăreanu, Youcheng Sun, and Divya 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)*. IEEE. 2021, pp. 25–28.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. „Attention is all you need“. In: *Advances in neural information processing systems* 30 (2017).

[Vla14] David C Vladeck. „Machines without principals: liability rules and artificial intelligence“. In: *Wash. L. Rev.* 89 (2014), p. 117.

[VR18] Sahil Verma and Julia Rubin. „Fairness definitions explained“. In: *Proceedings of the international workshop on software fairness*. 2018, pp. 1–7.

[Wan+20] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. „Linformer: Self-attention with linear complexity“. In: *arXiv preprint arXiv:2006.04768* (2020).

[Wan22] Yuan Wang. „Estimation and Comparison of Linear Regions for ReLU Networks.“ In: *IJCAI*. 2022, pp. 3544–3550.

[Wei76] Joseph Weizenbaum. „Computer power and human reason: From judgment to calculation“. In: *San Francisco* (1976).

[Wil63] R. H. Wilkinson. „A Method of Generating Functions of Several Variables Using Analog Diode Logic“. In: *IEEE Transactions on Electronic Computers* EC-12.2 (1963), pp. 112–129. DOI: 10.1109/PGEC.1963.263420.

[WMR17] Sandra Wachter, Brent Mittelstadt, and Chris Russell. „Counterfactual explanations without opening the black box: Automated decisions and the GDPR“. In: *Harv. JL & Tech.* 31 (2017), p. 841.

[WMR21] Sandra Wachter, Brent Mittelstadt, and Chris Russell. „Why fairness cannot be automated: Bridging the gap between EU non-discrimination law and AI“. In: *Computer Law & Security Review* 41 (2021), p. 105567.

[WZZ22] Xiaomeng Wang, Yishi Zhang, and Ruilin Zhu. „A brief review on algorithmic fairness“. In: *Management System Engineering* 1.1 (2022), p. 7.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[Xu+16] Jun Xu, Ton JJ van den Boom, Bart De Schutter, and Shuning Wang. „Irredundant lattice representations of continuous piecewise affine functions“. In: *Automatica* 70 (2016), pp. 109–120.

[Zas81] Thomas Zaslavsky. „The geometry of root systems and signed graphs“. In: *The American Mathematical Monthly* 88.2 (1981), pp. 88–105.

[Zed21] Carlos Zednik. „Solving the black box problem: A normative framework for explainable artificial intelligence“. In: *Philosophy & technology* 34.2 (2021), pp. 265–288.

[ZS19] Zhi Quan Zhou and Liqun Sun. „Metamorphic testing of driverless cars“. In: *Communications of the ACM* 62.3 (2019), pp. 61–67.

[ZW20] Xiao Zhang and Dongrui 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. URL: https://openreview.net/forum?id=SkeFl1HKwr.