Universität des Saarlandes

Master's Thesis

in partial fulfillment of the requirements for the degree of
M. Sc. Language Science and Technology

An Isabelle Formalization of the Expressiveness of Deep Learning

---

Alexander Bentkamp

Supervisors: Prof. Dr. Dietrich Klakow and Dr. Jasmin Blanchette

November 2016

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration

I hereby confirm that the thesis presented here is my own work, with all assistance acknowledged.

Lübeck, den 14.11.2016

Alexander Bentkamp

Abstract

Deep learning has had a profound impact on computer science in recent years, with applications to search engines, image recognition and language processing, bioinformatics, and more. Recently, Cohen et al. provided theoretical evidence for the superiority of deep learning over shallow learning. I formalized their mathematical proof using the proof assistant Isabelle/HOL. This formalization simplifies and generalizes the original proof, while working around the limitations of the Isabelle type system. To support the formalization, I developed reusable libraries of formalized mathematics, including results about the matrix rank, the Lebesgue measure, and multivariate polynomials, as well as a library for tensor analysis.

Contents

Acknowledgment

I would like to express my sincere gratitude to my supervisors Prof. Dr. Dietrich Klakow and Dr. Jasmin Blanchette for giving me the opportunity to combine their respective research areas machine learning and interactive theorem proving in this master thesis. With their excellent supervision, I have learned a lot about the process of scientific research and writing.

I am highly grateful to Dr. Johannes Hölzl for his support to understand the formalization of the Lebesgue measure, for reviewing my thesis, and for helping me to publish the formalization in the *Archive of Formal Proofs*.

Dr. Martin Lotz had plenty of patience explaining me his theorems on algebraic varieties. I am grateful because his advice which fundamentally improved the mathematical results of this thesis.

Dr. Ondřej Kunčar provided a preliminary version of his implementation of local typedefs from his dissertation project and helped me to apply it to my formalization.

Moreover, I would like to thank Dr. Florian Haftmann, Fabian Immler, Dr. Andreas Lochbihler, Alexander Maletzky, Dr. Walther Neuper, and Dr. René Thiemann for their helpful advice.

1. Introduction

Machine learning enables computers to perform tasks that seem impossible to teach them programmatically. Now there are unbeatable artificial intelligences playing Go, self-driving cars, and practical speech recognition systems. In particular deep learning algorithms have enabled this breakthrough. However, on the theoretical side only little is known about the reasons why these deep learning algorithms work so well. Recently, Cohen et al. [11] presented a theoretical approach using tensor theory that can explain the power of one particular deep learning algorithm called convolutional arithmetic circuits.

In this master's thesis I formalized their results in the proof assistant Isabelle/HOL. A proof assistant (also called interactive theorem prover) is an interactive software tool with a graphical user interface for the development of computer-checked formal proofs. Traditional proofs on paper normally omit smaller proof steps to increase readability. In contrast a formal proof contains every single application of interference rules, and states exactly which axioms, assumptions or previously deduced statements the rules are applied to. These formal proofs are difficult to grasp by humans, but computers can easily verify the correctness of these proofs. But the development of formal proofs can be arbitrarily tedious even if a traditional proof on paper already exists. Proof assistants facilitate this task by providing smaller proof steps automatically while letting the user contribute the larger proof steps and ideas.

Isabelle/HOL is a natural choice for this task because its strength lies in the high level of automation it provides. Moreover, it has an active community and includes a large library of formalized mathematical theories including measure theory, linear algebra, and polynomial algebra.

This work pursues several objectives: On the one hand, it is a case study of applying proof assistants to a field where they have been barely used before, namely machine learning. It shows that modern proof assistants such as Isabelle can be used in various fields to verify the correctness of results, but also supporting researchers to find new results. The generalization and the simplifications I found demonstrate how formal proof development can also benefit the research outside the world of formal proofs.

On the other hand, such as most formalizations, this project lead to the development of generally useful libraries that can be used in future formalizations about possibly completely different topics. Most prominently, I developed a library for tensors and their properties including the tensor product, tensor rank, and matricization. Moreover, I extended several libraries: I added a the matrix

rank and its properties to Thiemann and Yamada's matrix library [29], adapted the definition of the Lebesgue measure by Hölzl and Himmelmann [17] to my purposes, and extended Lochbihler and Haftmann's polynomial library [16] by various lemmas, including the theorem that zero sets of multivariate polynomials $\neq 0$ are Lebesgue null sets. These are topics that are independent of the domain of machine learning, and they can be build upon in future purely mathematical formalization projects. In addition to these main motivations, a personal interest is to develop my expertise with a proof assistant in a larger project.

This thesis is divided into three chapters: Chapter 2 discusses the mathematical background and the theory of deep learning. Chapter 3 introduces Isabelle/HOL, and Chapter 4 explains how the theory of deep learning is formalized.

This analysis of deep learning focuses on one particular type of networks called convolutional arithmetic circuits, which can be realized as shallow, deep or truncated networks (Sections 2.1 and 2.2). The characteristic of the deep network model is that it has more layers than the shallow network, while this does not necessarily imply more network nodes. The truncated network is a generalization of the two models, which can realize a network of any number of layers. Section 2.3 introduces the mathematical background knowledge needed to understand the mathematical theorems.

Section 2.4 explains the theorems of network capacity which essentially state that deep networks are more expressive than shallow ones. The deep networks are more powerful in expressing arbitrary functions, assuming that there is a perfect training algorithm to obtain the correct network weights. More precisely, the functions that can be expressed by the shallow model form a Lebesgue null set in the space of functions that can be expressed by the deep model. More generally, even adding a single layer in the truncated model has this effect. This is the result obtained by Cohen et al. [11] which I discuss in Section 2.5. To formalize the results the proofs by Cohen et al. are restructured (Section 2.6 and 2.7). The differences to the original proof are presented in Section 2.8. In addition to making the formalization easier the restructuring also leads to a useful generalization of the result (Section 2.9).

Chapter 3 introduces Isabelle/HOL: Its architecture is designed to ensure trustworthiness of the verified proofs (Section 3.1). Isabelle users contribute to an online library of formalizations called *Archive of Formal Proofs* (Section 3.2). Isabelle/HOL is based on the generic proof assistant Isabelle (Section 3.2), extended with a formalism of higher-order logic (HOL) (Section 3.4). Sections 3.5 to 3.7 describe the specification language of Isabelle/HOL, followed by the proof language in Section 3.8 and the general workflow in Section 3.9.

Chapter 4 presents the Isabelle/HOL libraries relevant for this formalization and how I extended them (Section 4.2, 4.4 and 4.5). Moreover, I present my newly developed tensor library (Section 4.3). Section 4.6 then explains how the deep learning networks and their properties are formalized.

2. The Expressiveness of Deep Learning

Machine learning algorithms are designed to make decisions or predictions without being explicitly programmed. This thesis focuses on *supervised* learning. These algorithms learn from a set of sample data, that specify input and desired output for each data set. This process is called *training*. The algorithms generalize this training data, allowing them to imitate the learned output on previously unseen input data.

A typical application of machine learning is image recognition, i.e., assigning a given image to one of several categories, depending on what the image depicts. For training, a supervised learning algorithm needs a set of labeled pictures, and builds a model of the features that cause an image to belong to a category. Afterwards, the algorithm can be used to label unseen pictures.

Deep learning algorithms use graph-like structures with multiple processing layers to represent the abstraction of the data. Empirically, it is known that more layers increase performance. Cohen et al. present a theoretic approach to explain this effect a deep learning architecture called *convolutional arithmetic circuits*. These circuits combine the ideas of two other architectures called *sum-product networks* and *convolutional neural networks*.

2.1. Sum-Product Networks

Sum-product networks (SPNs) are a deep learning architecture, also known as arithmetic circuits. SPNs consist of a rooted directed acyclic graph with input variables as leaf nodes and two types of interior nodes: sum nodes and product nodes. The incoming edges of sum nodes are labeled with real weights, which have to be learned during training. Possible training algorithms are expectation maximization or gradient descent [25].

The network is evaluated by assigning real numbers to each input variable, calculating the product at product nodes, and the weighted sum at sum nodes. The value at the root is the output of the network.

Figure 2.1 shows an example of an SPN. It has four input nodes x_1, x_2, x_3, x_4. Applied to the domain of image recognition, these values represent the pixel data of the image. The structure of the network has to be crafted manually. For example, it is reasonable to connect adjacent pixels early in the network. The

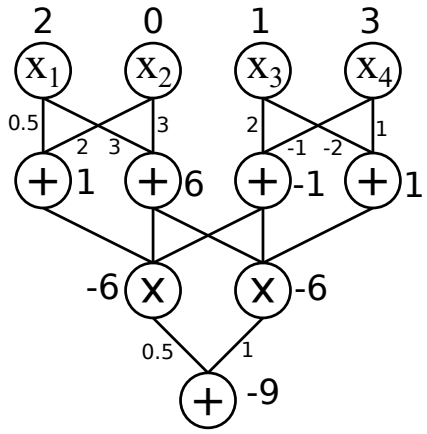


Figure 2.1.: The evaluation of an SPN

edges leading to sum nodes are labeled with weights, which are learned during training. The figure shows an evaluation of the network on the data $(2, 0, 1, 3)$, which represents a possibly unseen picture. Evaluating the weighted sums and products, the output value -9 is obtained. This value has to be interpreted in the same way that the training data is coded. For example, negative values could mean that the algorithm categorizes this data as a landscape picture, whereas positive values encode animal pictures.

2.2. Convolutional Arithmetic Circuits

Convolutional neural networks (ConvNets) are another deep learning architecture. ConvNets are structured as a stack of layers, such as convolutional, pooling and rectified linear unit layers. To evaluate a ConvNet for a data set, the convolutional layers apply a special system of weighted sums, whose weights have to be learned in training. Pooling layers are used to reduce the amount of data, usually by combining groups of incoming values to their average or their maximum value. Rectified linear unit (ReLU) layers apply the function $x \mapsto \max(0, x)$ to each incoming value.

Cohen et al. transfer the structure of ConvNets to SPNs, creating convolutional arithmetic circuits. They use 1×1 convolutions, which amount to multiplying a matrix containing weights to the incoming values. Instead of maximum or average pooling, they use multiplication, such that the resulting network is an SPN. They do not use ReLU layers to preserve the SPN nature of the network. Instead, a layer of non-linear function application is inserted before the SPN, called representational layer.

Although the analysis presented in this thesis is not limited to these, three models of convolutional arithmetic circuits are studied closely in this thesis:

- The deep network model which has the largest amount of layers possible using binary branching
- The shallow network model which has the smallest amount of layers possible
- The truncated network model which is a generalization of the first two, parameterized by the desired number of layers L_c. For $L_c = \log_2 N$ and $L_c = 1$ it is identical to the deep and the shallow network, respectively.

We further assume that the number of inputs N is a power of 2, which simplifies the proofs, although the analysis is not principally limited to these cases. The number of output nodes is denoted as Y.

The structure of these networks is illustrated in Figure 2.2 for networks with eight input nodes ($N = 8$). The networks take a vector of real numbers as input. In the figures each of the values is represented by a gray square. First, the representational layer (black arrows) is applied, which consists of non-linear functions f_θ such as gaussians or sigmoids. There are M different non-linear functions $f_{\theta_1}, \ldots, f_{\theta_M}$, such that each input value x_i yields a vector of M different values $f_{\theta_1}(x_i), \ldots, f_{\theta_M}(x_i)$ (the gray bars).

What follows is the SPN. In the language of SPNs we have alternating sum and product layers, which would be called 1×1 convolution and pooling layers in the language of ConvNets, respectively. They calculate the following:

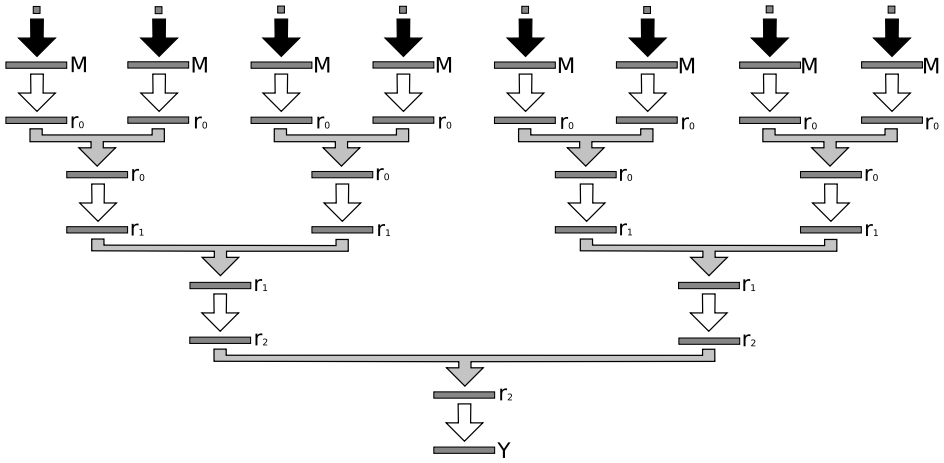
The convolutional layers (white arrows) multiply a weight matrix to the incoming vector. The size of the matrices depends on parameters $r_0, \ldots, r_{L-1}$. In layer l, the matrix has a size of $r_l \times r_{l-1}$, where $r_{-1} = M$ and $r_L = Y$. In the shallow model, we denote $Z = r_0$. The values of these matrices are denoted as $a_\alpha^{l,j,\gamma}$ where $\alpha = 1, \ldots, r_{l-1}$ is the column index, $\gamma = 1, \ldots, r_l$ is the row index, l denotes the number of the layer and j denotes the position in that layer. Expressed as a formula, the convolution operation performs the following mapping:

$$(v_\alpha)_{\alpha=1,\ldots,r_{l-1}} \mapsto \left(\sum_{\alpha=1}^{r_{l-1}} a_\alpha^{l,j,\gamma} v_\alpha \right)_{\gamma=1,\ldots,r_l}$$

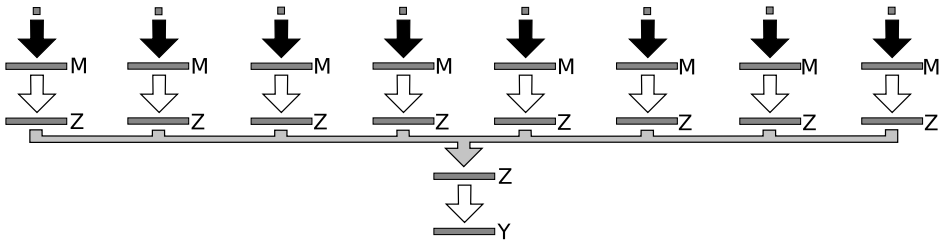
The entries of these matrices are the weights of the network, which must be learned during training. In this discussion we allow the weights in the branches of the network to be different. In practice it can be useful to have the branches share the weights, i.e., the weight matrices must be identical in each branch of the same layer. Then $a_\alpha^{l,j,\gamma}$ is independent of j, which results in a reduction of the number of weights. The proofs for the case of shared weights are analogous to the ones presented in this thesis.

The pooling layers (gray arrows) multiply the incoming vectors componentwise and do not contain any weights. So, if the pooling operation has J incoming vectors, it performs the following mapping:

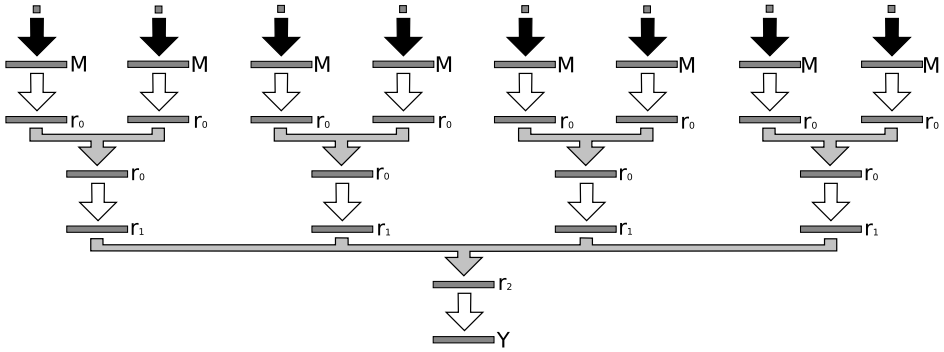
$$(v_\alpha^1)_{\alpha=1,\ldots,r_l}, \ldots, (v_\alpha^J)_{\alpha=1,\ldots,r_l} \mapsto (v_\alpha^1 \cdot \ldots \cdot v_\alpha^J)_{\alpha=1,\ldots,r_l}$$



(a) The deep network



(b) The shallow network



(c) The truncated network for $L_c = 2$

Figure 2.2.: The network models for $N = 8$ input nodes. The representational layer applies non-linear functions (black arrows). The convolutional layers multiply by a weight matrix (white arrows). The pooling layers multiply componentwise (gray arrows).

The deep network model always merges two branches at a time in a pooling layer, while the shallow network model merges all branches in the first pooling layer. The truncated network model starts merging two branches at a time and finally merges all branches at some layer L_c. The deep and the shallow networks are special cases of this model with $L_c = \log_2 N$ and $L_c = 1$, respectively.

2.3. Mathematical Background

In this section I give a brief introduction to the Lebesgue measure, and a more in-depth discussion to tensors. The explanations presume a basic understanding of linear algebra.

2.3.1. Lebesgue Measure

The Lebesgue measure is a mathematical description of the intuitive concept of length, surface or volume. It extends this concept from simple geometrical shapes to a large amount of subsets of $\mathbb{R}^n$, including all closed and open sets. It is provably impossible to have a measure that measures all subsets of $\mathbb{R}^n$ while maintaining intuitively reasonable properties. The sets that the Lebesgue measure can assign a volume are called *measurable*. The volume that is assigned to a measurable set can be a real number ≥ 0 or ∞. A set of Lebesgue measure 0 is also called *null set*. If a property holds for all points in $\mathbb{R}^n$ except for a null set, we say the property holds *almost everywhere*.

The following lemma is of special significance for the proofs in this thesis [9]:

Lemma 2.3.1. *If $p \not\equiv 0$ is a polynomial with d variables, the set of points in $\mathbb{R}^d$ where it vanishes is of Lebesgue measure zero.*

2.3.2. Tensors

The easiest way to understand tensors is to see them as multidimensional arrays. Vectors and matrices are special cases of tensors, but in general tensors are allowed to identify their entries by more than one or two indices. Each index corresponds to a *mode* of the tensor. For matrices these modes are "row" and "column". The number of modes is the *order* of the tensor. The number of values an index can take in a particular mode is the *dimension* in that mode. So a real tensor $\mathcal{A}$ of order N of dimension M_i in mode i contains values $\mathcal{A}_{d_1,\dots,d_N} \in \mathbb{R}$ for $d_i \in \{1,\dots,M_i\}$. The space of all these tensors is written as $\mathbb{R}^{M_1 \times \cdots \times M_N}$.

We define a product of tensors, which generalizes the outer vector product:

$$\begin{aligned} \otimes : \mathbb{R}^{M_1 \times \cdots \times M_{N_1}} \times \mathbb{R}^{M'_1 \times \cdots \times M'_{N_2}} &\to \mathbb{R}^{M_1 \times \cdots \times M_{N_1} \times M'_1 \times \cdots \times M'_{N_2}} \\ (\mathcal{A} \otimes \mathcal{B})_{d_1,\dots,d_{N_1},d'_1,\dots,d'_{N_2}} &= \mathcal{A}_{d_1,\dots,d_{N_1}} \cdot \mathcal{B}_{d'_1,\dots,d'_{N_2}} \end{aligned}$$

Analogously to matrices, we can define a rank on tensors, which is called *CP-rank*:

Definition 2.3.1. The CP-rank of a tensor $\mathcal{A}$ of order N is defined as the minimal number Z of terms that are needed to write $\mathcal{A}$ as a linear combination of tensor products of vectors $\mathbf{a}_{z,1}, \ldots, \mathbf{a}_{z,N}$:

$$\mathcal{A} = \sum_{z=1}^{Z} \lambda_z \cdot \mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N} \text{ with } \lambda_z \in \mathbb{R} \text{ and } \mathbf{a}_{z,i} \in \mathbb{R}^{M_i}$$

By rearranging the entries of a tensor, the values can also be written into a matrix, where each matrix entry corresponds to a tensor entry. This process is called *matricization*. To make it compatible with the tensor product, we define it as follows:

Definition 2.3.2. Let $\mathcal{A}$ be a tensor of even order $2N$. Let M_i be the dimension in mode $2i-1$ and M_i' the dimension in mode $2i$. The matricization $[\mathcal{A}] \in \mathbb{R}^{\prod M_i \times \prod M_i'}$ is defined as follows: The entry $\mathcal{A}_{d_1,d_1',\ldots,d_N,d_N'}$ is written into the matrix $[\mathcal{A}]$ at row $d_1 + M_1 \cdot (d_2 + M_2 \cdot (\cdots + M_{N-1} \cdot d_N))$ and column $d_1' + M_1' \cdot \left(d_2' + M_2' \cdot \left(\cdots + M_{N-1}' \cdot d_N'\right)\right)$.

This means that the even and the odd indices operate as digits in a mixed radix numeral system to specify the column and the row in the matricization, respectively. If all modes have equal dimension M, the indices are digits in a M-adic representation of the column and row indices.

The CP-rank of a tensor is related to the rank of its matrix in the following way:

Lemma 2.3.2. *Let $\mathcal{A}$ be a tensor of even order. Then*

$$\operatorname{rank}[\mathcal{A}] \leq \text{CP-rank}\, \mathcal{A}$$

A proof of this is given in the original paper by Cohen et al. [11, p. 21].

An alternative way to look at tensors is as multilinear maps. A map $f : \mathbb{R}^{M_1} \times \cdots \times \mathbb{R}^{M_N} \to \mathbb{R}$ is *multilinear* if for each i the mapping $\boldsymbol{x}_i \mapsto f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ is linear (all variables but x_i are fixed). Such a mapping is completely determined if its values on basis vectors are specified:

$$\begin{aligned} f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) &= f(x_{1,1}\boldsymbol{e}_1 + \cdots + x_{1,M_1}\boldsymbol{e}_{M_1}, \ldots, x_{N,1}\boldsymbol{e}_1 + \cdots + x_{N,M_N}\boldsymbol{e}_{M_N}) \\ &= \sum_{i_1,\ldots,i_N} x_{1,i_1} \cdot \cdots \cdot x_{N,i_N} \cdot f(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{i_N}) \end{aligned}$$

Therefore, to determine such a multilinear mapping we need $M_1 \times \cdots \times M_N$ real values that define the values for $f(\boldsymbol{e}_{i_1}, \ldots, \boldsymbol{e}_{i_N})$. The tensor that contains these values is identified with that multilinear mapping.

2.4. Theorems of Network Capacity

This analysis of the networks discusses their expressiveness. A function f can be *expressed* by a network if there exists a weight configuration such that the input x produces the output $f(x)$ for all possible inputs x. The expressiveness of a network is its ability to express functions with arbitrary weight configurations. Therefore this analysis does not discuss the training algorithm, i.e., how the desired weight configuration can be obtained given some training values of the function. I will call the functions that can be expressed with some given deep network or some given shallow network *deep network functions* and *shallow network functions*, respectively.

Since the representational layers of all three network models are the same (fixing the values of N and M), I focus on the lower part of the networks without the representational layer and I call it the *deep, shallow* or *truncated SPN*, respectively. Accordingly I call the functions expressed by the SPNs alone *deep, shallow* or *truncated SPN functions*.

The central question is whether and to what extend the deep SPN is more powerful than the shallow SPN. Both SPN models are known to be *universal*, meaning they can realize any multilinear function if an arbitrary number of nodes is allowed, i.e., if there is no limit to the values of r_l and Z, respectively. The models including the representational layer are also universal in the sense that they can approximate any L^2-function for $M \to \infty$.

When comparing the deep and the shallow network model, Cohen et al. fix the number of nodes in the deep network model by fixing the parameters N, M, r_0, $\ldots$, r_{L-1}, Y, and limit the amount of nodes in the shallow network by requiring $Z < r^{N/2}$ where $r := \min(r_0, M)$. The number of nodes in the shallow SPN is $(N+1) \cdot Z + Y$. Effectively, the shallow network is only allowed to use less than exponentially many nodes w.r.t. the number of inputs. They show that these "small" shallow networks can only express a small fraction of the functions that a deep network can express. More precisely they prove the following theorem, which is a slightly modified version of Theorem 1 from the original work by Cohen et al. [11]. Note that the representing tensors mentioned in the original theorem are isomorphic to the SPN functions used here.

Theorem 2.4.1 (Fundamental theorem of network capacity)**.** *Consider the deep network model with parameters M, r_l and N and define $r := \min(r_0, M)$. The weight space of the deep network is the space of all possible values for the weights $a_\alpha^{l,j,\gamma}$. In this weight space, let S be the set of weight configurations that represent a deep SPN function which can also be expressed by the shallow SPN model with $Z < r^{N/2}$. Then S is a set of measure zero with respect to the Lebesgue measure.*

This theorem can be generalized to the following theorem about truncated networks, which is a slightly modified version of Theorem 2 from Cohen et al. [11]:

Theorem 2.4.2 (Generalized theorem of network capacity)**.** *Consider two trun-*

cated SPNs, one with depth L_1 and parameters $r_l^{(1)}$, the other with depth L_2 and parameters $r_l^{(2)}$. Let $L_2 < L_1$ and define $r := \min\{M, r_0^{(1)}, \ldots, r_{L_2-1}^{(1)}\}$. In the space of all possible weight configurations for the L_1-network, let S be the set of weight configurations that represent a L_1-SPN function which can also be expressed by the L_2-SPN model with $r_{L_2-1}^{(2)} < r^{N/2^{L_2}}$. Then S is a set of Lebesgue measure zero.

Theorem 2.4.1 is a special case of this theorem, setting $L_1 = \log_2 N$ and $L_2 = 1$. Both of these theorems can be expanded quite easily to the entire networks including the representational layer. Moreover, it can be shown that S is closed, which means that any deep SPN function with parameters outside S cannot be approximated arbitrarily well by the shallow SPN model. Since these corollaries are of less interest for this work, see the original work [11] for details.

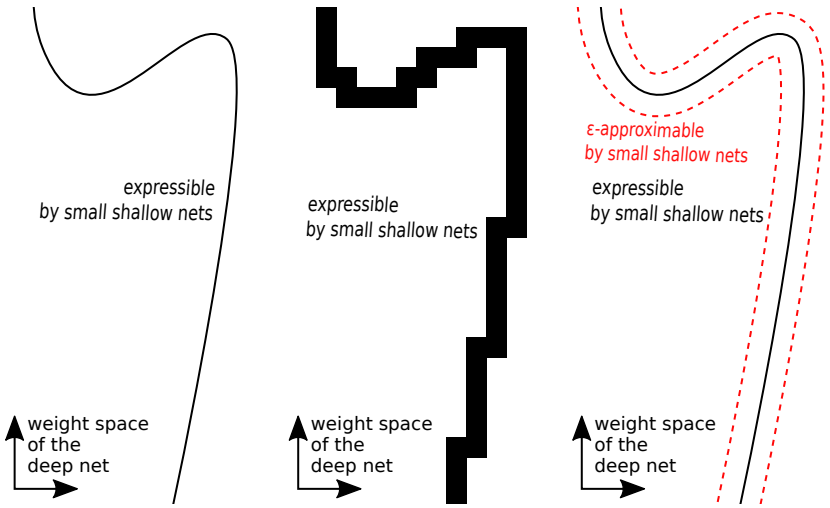
2.5. Discussion of the Original Result

2.5.1. Null Sets and Approximation

Although Cohen et al. provided a detailed insight into the mathematical theory behind deep learning, this result should be studied carefully. What exactly does it mean that S is of measure zero (also called a *null set*)? Cohen et al. refer to the probabilistic interpretation of null sets, which states the following: If one draws a random point from $\mathbb{R}^n$ using some continuous distribution, with probability 1 the resulting point will lie outside of a given null set. The event that the point lies inside that null set has probability 0, but is still possible (provided that the null set is not the empty set). This distinction between events of probability 0 and impossible events makes sense theoretically, but whether it applies in the real world is debatable.

In a corollary Cohen et al. also prove that S is closed. This means that any point outside S cannot be approximated arbitrarily well using points from S. This excludes for example sets such as $\mathbb{Q} \subset \mathbb{R}$, which is a null set, but not closed. Nonetheless, there are large subsets of $\mathbb{R}^n$ that are closed and null sets. For example consider the set $\{(x_1, \ldots, x_n) \mid x_1 \text{ is a multiple of } \varepsilon\} \subset \mathbb{R}^n$ for some fixed $\varepsilon > 0$, which is a set of hyperplanes with distance ε to each other. The set is a null set and closed. Any point outside the set can be approximated, not arbitrarily well, but up to ε. Mathematically this is a difference, but in practice there is no difference if ε is small.

In implementations of the deep learning algorithm, there will be a limit to how exact the calculations can be performed and a limit to what values the network weights can have. Therefore the weight space is a finite set, since the computer can only store a finite number of different values. With respect to the Lebesgue measure the entire weight space and all its subsets are then a closed null set. For these reasons, it would be desirable to find more precise ways to measure the size



(a) An illustration of the set S as a line in the 2-dimensional space.

(b) The set S how it would look like in an implementation.

(c) An approximation of the discretized set using the ϵ-neighborhood of S.

Figure 2.3.

of the set S.

One way to do this is to use a uniform discrete measure on some subset of $\mathbb{R}^n$. If the line in Figure 2.3a represents the set S, using a discrete measure would correspond to measuring a set as illustrated in Figure 2.3b. This set is closer to the discrete weight space in an implementation, in this illustration using fixed point arithmetic. Since this discrete set is cumbersome to handle mathematically, an alternative is illustrated in Figure 2.3c. The ε-neighborhood of S is a good approximation of the set in Figure 2.3b, and it is much easier to describe it mathematically.

2.5.2. ReLU Networks

Unfortunately, the convolutional arithmetic circuits are easy to analyze, but little used. They are equivalent to SimNets which have been developed by Cohen et al., the same research group that also found this tensor approach to analyze them.

However, Cohen et al. claim that these networks are simply in an early stage of development and have the potential to outperform the popular ConvNets with rectified linear unit (ReLU) activation. SimNets have been demonstrated to perform as well as these state of the art networks, even outperform them when computational resources are limited [10].

Moreover, the tensor analysis of convolutional arithmetic circuits can be connected to the ConvNets with ReLU activation[12]. Cohen et al. provide a transformation of convolutional arithmetic circuits into ConvNets with ReLU activation, which allows to deduce properties of the latter from the tensor analysis described

here. Unlike the convolutional arithmetic circuits, ReLU ConvNets with average pooling are not universal, i.e., even with an arbitrary number of nodes arbitrary functions cannot be expressed. Moreover, ReLU ConvNets do not show complete depth efficiency, i.e., the analogue of the set S for those networks has a Lebesgue measure greater zero. This leads Cohen et al. believe that convolutional arithmetic circuits could become a leading approach for deep learning, once suitable training algorithms have been developed.

2.6. The Restructured Proof of the Fundamental Theorem of Network Capacity

2.6.1. Proof Outline

The proof of Theorem 2.4.1 by Cohen et al. is a single, monolithic induction over the deep network structure that contains matrix theory, tensor theory, measure theory and polynomials. This proof strategy is not appropriate for a formalization, since inductions are generally complicated enough already and this approach does not allow a separation of the different mathematical theories involved. I restructured the proof to obtain a more modular version, which is presented here.

The restructured proof follows the following strategy:

Step I. We describe the behavior of an SPN function at an output node y by a tensor $\mathcal{A}^y$ that depends on the network weights. We focus on an arbitrary output node y of the deep network. If the shallow network cannot represent the output of this node, it cannot represent the entire output either.

Step II. We show that the CP-rank of a tensor representing an SPN function indicates how many nodes the shallow model needs to represent this function.

Step III. We construct a multivariate polynomial p, mapping the deep network weights w to a real number $p(w)$.

Step IV. We show that if $p(w) \neq 0$, the tensor $\mathcal{A}^y$ representing the network with weights w has a high CP-rank. More precisely, the CP-rank is then exponential in the number of inputs.

Step V. Show that p is not the zero polynomial and hence its zero set is a Lebesgue null set by Lemma 2.3.1.

By steps IV and V, $\mathcal{A}^y$ has an exponential CP-rank almost everywhere. By step II, the shallow network therefore needs exponentially many nodes to represent the deep SPN functions almost everywhere, which proves Theorem 2.4.1.

2.6.2. Tensors and Sum-Product Networks

The SPNs described before define a multilinear mapping from their input vectors to their output vectors: The convolutional layers contain a multiplication by a matrix, which is a linear mapping. The pooling layers contain a componentwise multiplication, which is linear if all but one of the incoming vectors are fixed. Therefore, each output value of the network can be represented by a tensor $\mathcal{A}^y$, which is step I in the proof outline. The tensor's entries $\mathcal{A}^y_{d_1,\dots d_N}$ contain the output value of the network if the input vectors are the basis vectors $\boldsymbol{e}_{d_1}, \dots, \boldsymbol{e}_{d_N}$.

The representing tensor can be computed inductively through the network, where convolutional layers introduce weighted sums of tensors and pooling layers introduce tensor products. For the deep network this results in the following equations:

$$\begin{aligned}
\psi^{0,j,\gamma} &= \mathbf{e}_\gamma \\
\phi^{0,j,\gamma} &= \sum_{\alpha=1}^{M} a_\alpha^{0,j,\gamma}\psi^{0,j,\alpha} = (a_1^{0,j,\gamma}, \dots a_M^{0,j,\gamma}) \\
\psi^{1,j,\gamma} &= \phi^{0,2j-1,\gamma} \otimes \phi^{0,2j,\gamma} \\
\phi^{1,j,\gamma} &= \sum_{\alpha=1}^{r_0} a_\alpha^{1,j,\gamma}\psi^{1,j,\alpha} \\
&\dots \\
\phi^{L-1,j,\gamma} &= \sum_{\alpha=1}^{r_{L-2}} a_\alpha^{L-1,j,\gamma}\psi^{L-1,j,\alpha} \\
\psi^{L,1,\gamma} &= \phi^{L-1,1,\gamma} \otimes \phi^{L-1,2,\gamma} \\
\mathcal{A}^y &= \phi^{L,1,y} = \sum_{\alpha=1}^{r_{L-1}} a_\alpha^{L,1,y}\psi^{L,1,\alpha}
\end{aligned}$$

For the shallow network the same principles apply and yield an equation that is close to the definition of the CP-rank, which will be useful in the proof below:

$$\mathcal{A}^y = \sum_{\alpha=1}^{Z} a_\alpha^{1,1,y} \cdot \mathbf{a}^{1,\gamma} \otimes \dots \otimes \mathbf{a}^{N,\gamma} \text{ where } \mathbf{a}^{j,\gamma} = (a_1^{0,j,\gamma}, \dots, a_M^{0,j,\gamma})$$

This equation shows that for some shallow network with a fixed parameter Z, all tensors representing this network have a CP-rank of at most Z, which is step II of the proof outline:

Lemma 2.6.1. *Let $\mathcal{A}$ be a tensor. If a shallow SPN with parameter Z is represented by this tensor, then*

$$Z \geq \text{CP-rank}(\mathcal{A})$$

So when the CP-rank of the tensor representing the deep network is exponential, the number of nodes needed by the shallow network is exponential.

The same applies to the truncated SPN and results in these equations:

$$\begin{aligned}
\psi^{0,j,\gamma} &= \mathbf{e}_\gamma \\
\phi^{0,j,\gamma} &= \sum_{\alpha=1}^{M} a_\alpha^{0,j,\gamma}\psi^{0,j,\alpha} = (a_1^{0,j,\gamma}, \dots a_M^{0,j,\gamma}) \\
\psi^{1,j,\gamma} &= \phi^{0,2j-1,\gamma} \otimes \phi^{0,2j,\gamma} \\
\phi^{1,j,\gamma} &= \sum_{\alpha=1}^{r_0} a_\alpha^{1,j,\gamma}\psi^{1,j,\alpha} \\
&\dots \\
\phi^{L_c-1,j,\gamma} &= \sum_{\alpha=1}^{r_{L_c-2}} a_\alpha^{L_c-1,j,\gamma}\psi^{L_c-1,j,\alpha} \\
\psi^{L_c,1,\gamma} &= \bigotimes_{j=1}^{N/2^{L_c-1}} \phi^{L_c-1,j,\gamma} \\
\mathcal{A}^y &= \phi^{L_c,1,y} = \sum_{\alpha=1}^{r_{L_c-1}} a_\alpha^{L_c,1,y}\psi^{L_c,1,\alpha}
\end{aligned}$$

2.6.3. The Restructured Proof

As described in the proof outline (Section 2.6.1), we want to define a multivariate polynomial p (Step III), which maps the network weights w of the deep network to a real number $p(w)$ and has the following properties:

- If $p(w) \neq 0$, then the tensor $\mathcal{A}^y$ representing the deep network with weights w has a high CP-rank. More precisely, CP-rank$(\mathcal{A}^y) \geq r^{N/2}$. (Step IV)
- The polynomial p is not the zero polynomial. (Step V)

By Lemma 2.3.2, the CP-rank of a tensor is greater than or equal to the rank of its matricization. Therefore, it suffices to show a high rank of $[\mathcal{A}^y]$ instead of a high CP-rank of $\mathcal{A}^y$. As a first step to defining a polynomial p with the desired properties, we show this rank to be high for one specific weight configuration:

Lemma 2.6.2. *There exists a deep SPN weight configuration such that*

$$\text{rank}[\mathcal{A}^y] = r^{N/2}$$

.

Proof. We prove by induction over the deep network structure that there exists a weight configuration such that

$$(\phi^{l,j,\gamma})_{d_1,\dots,d_{2^l}} = \begin{cases} 1 & \text{if } d_i < r \text{ for all } i \\ & \text{and } (d_1, d_3, \dots, d_{2^l-1}) = (d_2, d_4, \dots, d_{2^l}) \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

for all $l \geq 1$ and for all j and γ.

We start with the base case $l = 1$: In the first convolutional layer, we choose matrices that contain 1 on their main diagonal, and zero elsewhere. (If the matrix is square, it would be the identity matrix.) Then we obtain after the first convolutional layer:

$$\begin{aligned} \phi^{0,j} &= (\mathbf{e}_1, \ldots, \mathbf{e}_M, 0, \ldots, 0) && \text{if } M < r_0, \text{ or} \\ \phi^{0,j} &= (\mathbf{e}_1, \ldots, \mathbf{e}_{r_0}) && \text{if } r_0 \leq M \end{aligned}$$

For tensors of order 1, the tensor product is identical with the outer vector product. Therefore, after the first pooling layer, we obtain

$$\begin{aligned} \psi^{1,j} &= (\mathbf{e}_1\mathbf{e}_1^t, \ldots, \mathbf{e}_M\mathbf{e}_M^t, 0, \ldots, 0) && \text{if } M < r_0, \text{ or} \\ \psi^{1,j} &= (\mathbf{e}_1\mathbf{e}_1^t, \ldots, \mathbf{e}_{r_0}\mathbf{e}_{r_0}^t) && \text{if } r_0 \leq M \end{aligned}$$

In the second convolutional layer, we choose matrices that have ones everywhere. As a result we obtain

$$\phi^{1,j,\gamma} = \mathbf{e}_1\mathbf{e}_1^t + \cdots + \mathbf{e}_r\mathbf{e}_r^t \text{ for all } \gamma, \text{ where } r = \min(r_0, M)$$

This tensor fulfills equation 2.1. This ends the base case of our induction.

For the induction step we assume that

$$(\phi^{l-1,j,\gamma})_{d_1,\ldots,d_{2^{l-1}}} = \begin{cases} 1 & \text{if } d_i < r \text{ for all } i \\ & \text{and } (d_1, d_3, \ldots, d_{2^{l-1}-1}) = (d_2, d_4, \ldots, d_{2^{l-1}}) \\ 0 & \text{otherwise} \end{cases}$$

for all j and for all γ. After the following pooling layer we obtain $\psi^{l,j,\gamma} = \phi^{l-1,2j-1,\gamma} \otimes \phi^{l-1,2j,\gamma}$, i.e.,

$$(\psi^{l,j,\gamma})_{d_1,\ldots,d_{2^l}} = \begin{cases} 1 & \text{if } d_i < r \text{ for all } i \\ & \text{and } (d_1, d_3, \ldots, d_{2^l-1}) = (d_2, d_4, \ldots, d_{2^l}) \\ 0 & \text{otherwise} \end{cases}$$

In the following convolutional layer we choose matrices that contain only ones in their first column, and zeros in the other columns. Therefore we obtain $\phi^{l,j,\gamma} = \psi^{l,j,1}$ for all j and all γ. So $\phi^{l,j,\gamma}$ fulfills equation 2.1 and this concludes the induction step.

Since $\phi^{L,j,y} = \mathcal{A}^y$, we have shown that

$$\mathcal{A}^y_{d_1,\ldots,d_N} = \begin{cases} 1 & \text{if } d_i < r \text{ for all } i \\ & \text{and } (d_1, d_3, \ldots, d_{N-1}) = (d_2, d_4, \ldots, d_N) \\ 0 & \text{otherwise} \end{cases}$$

This is equivalent to a diagonal matrix $[\mathcal{A}^y]$ that has a 1 on the diagonal position

k if k has a M-adic representation that contains only digits lower than r, and 0 otherwise. This matrix has dimension $M^{N/2} \times M^{N/2}$ and therefore there are $r^{N/2}$ different M-adic representations that contain only digits lower than r. So $[\mathcal{A}^y]$ is a diagonal matrix with $r^{N/2}$ non-zero entries and hence $\text{rank}[\mathcal{A}^y] = r^{N/2}$ for this weight configuration. □

A well known lemma from matrix theory connects the rank of a matrix to its square submatrices with non-zero determinant. A *submatrix* is obtained from a matrix by deleting any rows and/or columns. The determinants of square submatrices are also called *minors*. The *size* of the minor is the size of the submatrix that it corresponds to.

Lemma 2.6.3. *The rank of a matrix is equal to the size of its largest non-zero minor.*

We define p as the mapping from the network weights to one of the $r^{N/2} \times r^{N/2}$ minors of $[\mathcal{A}^y]$. Independently of the minor we choose, by Lemma 2.6.3, $\text{rank}[\mathcal{A}^y] \geq r^{N/2}$ if $p(w) \neq 0$. By Lemma 2.3.2, it follows that p fulfills the first desired property that $\text{CP-rank}(\mathcal{A}^y) \geq r^{N/2}$ if $p(w) \neq 0$, which completes step IV.

Lemma 2.6.2 states that there is a weight configuration w where $\text{rank}[\mathcal{A}^y] = r^{N/2}$. By Lemma 2.6.3, this implies that there exists a non-zero $r^{N/2} \times r^{N/2}$ minor of $[\mathcal{A}^y]$ for this weight configuration w. By choosing one of these minors for the definition of p, we can ensure the second desired property that p is not the zero polynomial, which completes step V.

It is not obvious that the mapping from the network weights to this $r^{N/2} \times r^{N/2}$ minor of $[\mathcal{A}^y]$ is indeed a polynomial, though:

Lemma 2.6.4. *Any mapping from the deep network weights to one of the minors of $[\mathcal{A}^y]$ can be represented as a polynomial.*

Proof. First, we show by induction over the SPN structure that the entries of $[\mathcal{A}^y]$ are polynomials if we consider the weights as variables: The inputs of the SPN (i.e. after the representational layer) are constant with respect to the weights and therefore polynomial. The convolutional layers compute a multiplication by a weight matrix, so only multiplication and addition operations are involved, which map polynomials to polynomials. The pooling layers involve multiplications only, so polynomials are mapped to polynomials.

Finally, the resulting tensor $\mathcal{A}^y$ has polynomial entries, and therefore the entries of $[\mathcal{A}^y]$ are polynomial. Calculating a minor amounts to picking some of these polynomial entries and calculating their determinant. The Leibniz formula of the determinant involves only products and sums. Therefore the minors of $[\mathcal{A}^y]$ are polynomial as well. □

We can now prove Theorem 2.4.1:

Proof of Theorem 2.4.1. Let S be the set of weight configurations that represent a deep SPN function which can also be expressed by the shallow SPN model with $Z < r^{N/2}$. We must show that S is a null set.

Let $\mathcal{A}^y$ be the representing tensor of the deep SPN for some weight configuration w. By the discussion above, there exists a non-zero polynomial p with the property that whenever $p(w) \neq 0$, then CP-rank$(\mathcal{A}^y) \geq r^{N/2}$.

Let $S' = \{w \mid p(w) = 0\}$. Then we have CP-rank$(\mathcal{A}^y) \geq r^{N/2}$ except on S'. We consider a shallow SPN with parameter Z that can express $\mathcal{A}^y$ as well. By Lemma 2.6.1 we obtain $Z \geq$ CP-rank$(\mathcal{A}^y) \geq r^{N/2}$ except on S'. Therefore we have $S \subseteq S'$. Given that $p \not\equiv 0$, S' is a null set by Lemma 2.3.1, which proves that S is a null set as well. □

2.7. Analogous Restructuring for the Generalized Theorem of Network Capacity

The proof of Theorem 2.4.2 can be restructured in the same way and is only slightly more complicated. As stated in the theorem, we compare two truncated networks, one with depth L_1 and parameters $r_l^{(1)}$, the other with depth L_2 and parameters $r_l^{(2)}$. We assume that $L_2 < L_1$. The proof works mostly analogously, with the L_1 network taking over the role of the deep network and the L_2 network taking over the role of the shallow network.

2.7.1. The "Squeezing Operator" φ_q

As in the original proof, we need to introduce the "squeezing" operator φ_q, which maps a tensor of higher order to a tensor of lower order. It is similar to the matricization, in that it only rearranges the tensor entries while preserving their values.

Definition 2.7.1. Let $q \in \mathbb{N}$ and let $\mathcal{A}$ be a tensor of order $c \cdot q$ (for some $c \in \mathbb{N}$) and dimension M_i in mode i. Then $\varphi_q(\mathcal{A})$ is a tensor of order c where the entry $\mathcal{A}_{d_1,\ldots,d_c}$ is written into $\varphi_q(\mathcal{A})$ at position

$$
\begin{aligned}
&d_1 + M_1 \cdot (d_2 + M_2 \cdot (\cdots + M_{q-1} \cdot d_q),\\
&d_{q+1} + M_{q+1} \cdot (d_{q+2} + M_{q+2} \cdot (\cdots + M_{2q-1} \cdot d_{2q}),\\
&\ldots,\\
&d_{cq-q+1} + M_{cq-q+1} \cdot (d_{cq-q+2} + M_{cq-q+2} \cdot (\cdots + M_{cq-1} \cdot d_{cq})
\end{aligned}
$$

In other words, blocks of q modes each are squeezed into one mode, using the mixed radix numeral system of base M_i. This operator is compatible with tensor

addition and multiplication in the following way:

$$\begin{aligned}
&\varphi_c(\mathcal{A} + \mathcal{B}) = \varphi_c(\mathcal{A}) + \varphi_c(\mathcal{B}) \\
&\varphi_c(\mathcal{A} \otimes \mathcal{B}) = \varphi_c(\mathcal{A}) \otimes \varphi_c(\mathcal{B}) \qquad \text{for tensors } \mathcal{A} \text{ and } \mathcal{B} \text{ of order divisible by } q
\end{aligned}$$

We need the squeezing operator only for $q = 2^{L_2-1}$ and abbreviate

$$\varphi := \varphi_{2^{L_2-1}}$$

2.7.2. CP-rank of Truncated SPN Tensors

Analogously to Lemma 2.6.1, we can also reason about the tensor that is produced by the truncated SPN of depth L_2. However, we cannot estimate the CP-rank of this tensor directly, but the CP-rank of its "squeezed" version. Recall from Section 2.6.2 that the final steps in constructing a truncated SPN tensor of depth L_2 are:

$$\begin{aligned}
\psi^{L_2,1,\gamma} &= \bigotimes_{j=1}^{N/2^{L_2-1}} \phi^{L_2-1,j,\gamma} \\
\mathcal{A}^y &= \sum_{\alpha=1}^{r_{L_2-1}} a_\alpha^{L_2,1,y} \psi^{L_2,1,\alpha}
\end{aligned}$$

Now we apply the "squeezing" operator φ and use its compatibility with tensor addition and multiplication to obtain

$$\varphi(\mathcal{A}^y) = \sum_{\alpha=1}^{r_{L_2-1}} a_\alpha^{L_2,1,y} \bigotimes_{j=1}^{N/2^{L_2-1}} \varphi(\phi^{L_2-1,j,\gamma})$$

Since $\phi^{L_2-1,j,\gamma}$ is a tensor of order 2^{L_2-1}, its "squeezed" version $\varphi(\phi^{L_2-1,j,\gamma})$ is of order 1, i.e., a vector. By definition of the CP-rank (Definition 2.3.1), this shows that the "squeezed" version of any truncated SPN tensor of depth L_2 has a maximal CP-rank of r_{L_2-1}:

Lemma 2.7.1. *Let $\mathcal{A}$ be a tensor. If a truncated SPN with depth L_2 is represented by this tensor, then*

$$r_{L_2-1} \geq \text{CP-rank}(\varphi(\mathcal{A}))$$

2.7.3. The Restructured Proof

As in the proof of Theorem 2.4.1, we construct a polynomial p. This polynomial maps the weights of the L_1-SPN to a real number. However, since we can only estimate the CP-rank of the "squeezed" tensor, we need p to have the following properties:

- If $p(w) \neq 0$, then the tensor $\mathcal{A}^y$ representing the L_1-SPN fulfills the inequality CP-rank$(\varphi(\mathcal{A}^y)) \geq r^{N/2^{L_2}}$.

- The polynomial p is not the zero polynomial.

By Lemma 2.3.2, it suffices to show a high rank of $[\varphi(\mathcal{A}^y)]$ instead of a high CP-rank of $\varphi(\mathcal{A}^y)$. As a first step, we show this rank to be high for one specific weight configuration:

Lemma 2.7.2. *Let $\mathcal{A}^y$ be a tensor representing the L_1-SPN. Then there exists a weight configuration of the L_1-SPN such that* rank$[\varphi(\mathcal{A}^y)] = r^{N/2^{L_2}}$ *where* $r := \min\{M, r_0^{(1)}, \ldots, r_{L_2-1}^{(1)}\}$.

Proof. **Step 1:** First, we prove by induction over the first L_2 layers of the L_1-network structure that there exists a weight configuration such that

$$(\phi^{l,j,\gamma})_{d_1,\ldots,d_{2^l}} = \begin{cases} 1 & \text{if } \gamma \leq r \text{ and } (d_1, \ldots, d_{2^l}) = (\gamma, \ldots, \gamma) \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

for all $0 \leq l \leq L_2 - 1$ and for all j and γ. Note that $\phi^{l,j,\gamma}$ and $\psi^{l,j,\gamma}$ here refer to the corresponding tensors in the L_1-SPN.

We start with the base case $l = 0$: In the first convolutional layer, we choose matrices that contain an $r \times r$ identity matrix in their upper left corner and 0 elsewhere. Then we obtain after the first convolutional layer:

$$\phi^{0,j} = (\mathbf{e}_1, \ldots, \mathbf{e}_r, 0, \ldots, 0)$$

This fulfills equation 2.2 for $l = 0$.

For the induction step we assume that

$$(\phi^{l-1,j,\gamma})_{d_1,\ldots,d_{2^{l-1}}} = \begin{cases} 1 & \text{if } \gamma \leq r \text{ and } (d_1, \ldots, d_{2^{l-1}}) = (\gamma, \ldots, \gamma) \\ 0 & \text{otherwise} \end{cases}$$

for all j and for all γ. After the following pooling layer we obtain $\psi^{l,j,\gamma} = \phi^{l-1,2j-1,\gamma} \otimes \phi^{l-1,2j,\gamma}$, i.e.,

$$(\psi^{l,,j,\gamma})_{d_1,\ldots,d_{2^l}} = \begin{cases} 1 & \text{if } \gamma \leq r \text{ and } (d_1, \ldots, d_{2^l}) = (\gamma, \ldots, \gamma) \\ 0 & \text{otherwise} \end{cases}$$

In the following convolutional layer we choose again matrices that contain an $r \times r$ identity matrix in their upper left corner and 0 elsewhere. Since $\psi^{l,j,\gamma}$ is a zero tensor for $\gamma > r$ anyway we obtain $\phi^{l,j,\gamma} = \psi^{l,j,\gamma}$ for all j and all γ. So $\phi^{l,j,\gamma}$ fulfills equation 2.1 and this concludes the induction. In particular we have for

$l = L_2 - 1$:

$$(\phi^{L_2-1,j,\gamma})_{d_1,\dots,d_{2^{L_2-1}}} = \begin{cases} 1 & \text{if } \gamma \leq r \text{ and } (d_1,\dots,d_{2^{L_2-1}}) = (\gamma,\dots,\gamma) \\ 0 & \text{otherwise} \end{cases}$$

Therefore $\varphi(\phi^{L_2-1,j,\gamma}) = \mathbf{0}$ for $\gamma > r$ and $\varphi(\phi^{L_2-1,j,\gamma}) = \mathbf{e}_{i_\gamma}$ for $\gamma \leq r$ where i_γ is the 2^{L_2-1}-digit number with all digits of value γ in the numeral system of base M. In particular $i_1 < i_2 < \cdots < i_r$, i.e., the i_γ are all different.

Step 2: We prove by induction over the following layers of the L_1-SPN structure that there exists a weight configuration such that

$$\varphi(\phi^{l,j,\gamma})_{d_1,\dots,d_{2^{l-L_2+1}}} = \begin{cases} 1 & \text{if } d_i \in \{i_\gamma\}_{\gamma=1,\dots,r} \text{ for all } i \\ & \text{and } (d_1, d_3, \dots, d_{2^{l-L_2+1}-1}) \\ & = (d_2, d_4, \dots, d_{2^{l-L_2+1}}) \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

for $l = L_2, \dots, L_1 - 1$ and for all j and γ.

We start with the base case $l = L_2$: From Step 1 we know that

$$\begin{aligned} \varphi(\phi^{L_2-1,j,\gamma}) &= \mathbf{e}_{i_\gamma} && \text{for } \gamma \leq r \\ \varphi(\phi^{L_2-1,j,\gamma}) &= \mathbf{0} && \text{for } \gamma > r \end{aligned}$$

For tensors of order 1, the tensor product is identical with the outer vector product. Therefore, after next pooling layer, we obtain

$$\begin{aligned} \varphi(\psi^{L_2,j,\gamma}) &= \mathbf{e}_{i_\gamma}\mathbf{e}_{i_\gamma}^t && \text{for } \gamma \leq r \\ \varphi(\psi^{L_2,j,\gamma}) &= \mathbf{0} && \text{for } \gamma > r \end{aligned}$$

In the following convolutional layer, we choose matrices that have ones everywhere. Since the φ-operator is compatible with addition we then obtain

$$\varphi(\phi^{L_2,j,\gamma}) = \mathbf{e}_{i_1}\mathbf{e}_{i_1}^t + \cdots + \mathbf{e}_{i_r}\mathbf{e}_{i_r}^t \text{ for all } \gamma$$

This tensor fulfills equation 2.3. This ends the base case of our induction.

For the induction step we assume that

$$\varphi(\phi^{l-1,j,\gamma})_{d_1,\dots,d_{2^{l-L_2}}} = \begin{cases} 1 & \text{if } d_i \in \{i_\gamma\}_{\gamma=1,\dots,r} \text{ for all } i \\ & \text{and } (d_1, d_3, \dots, d_{2^{l-L_2}-1}) \\ & = (d_2, d_4, \dots, d_{2^{l-L_2}}) \\ 0 & \text{otherwise} \end{cases}$$

for all j and for all γ. After the following pooling layer we obtain $\psi^{l,j,\gamma} = \phi^{l-1,2j-1,\gamma} \otimes \phi^{l-1,2j,\gamma}$. Since $l \geq L_2$ and therefore the order of $\phi^{l-1,2j-1,\gamma}$ and $\phi^{l-1,2j,\gamma}$ is a multiple of 2^{L_2-1}, this implies $\varphi(\psi^{l,j,\gamma}) = \varphi(\phi^{l-1,2j-1,\gamma}) \otimes \varphi(\phi^{l-1,2j,\gamma})$.

Hence:

$$\varphi(\psi^{l,j,\gamma})_{d_1,\dots,d_{2^{l-L_2+1}}} = \begin{cases} 1 & \text{if } d_i \in \{i_\gamma\}_{\gamma=1,\dots,r} \text{ for all } i \\ & \text{and } (d_1, d_3, \dots, d_{2^{l-L_2+1}-1}) \\ & = (d_2, d_4, \dots, d_{2^{l-L_2+1}}) \\ 0 & \text{otherwise} \end{cases}$$

In the following convolutional layer we choose matrices that contain only ones in their first column, and zeros in the other columns. Therefore we obtain $\phi^{l,j,\gamma} = \psi^{l,j,1}$ for all j and all γ. So $\phi^{l,j,\gamma}$ fulfills equation 2.3 and this concludes the induction step.

Step 3: In the last pooling layer we have

$$\psi^{L_1,1,\gamma} = \bigotimes_{j=1}^{N/2^{L_1-1}} \phi^{L_1-1,j,\gamma}, \qquad \text{i.e., } \varphi(\psi^{L_1,1,\gamma}) = \bigotimes_{j=1}^{N/2^{L_1-1}} \varphi(\phi^{L_1-1,j,\gamma})$$

It follows that

$$\varphi(\psi^{L_1,j,\gamma})_{d_1,\dots,d_{N/2^{L_1-1}}} = \begin{cases} 1 & \text{if } d_i \in \{i_\gamma\}_{\gamma=1,\dots,r} \text{ for all } i \\ & \text{and } (d_1, d_3, \dots, d_{N/2^{L_1-1}-1}) \\ & = (d_2, d_4, \dots, d_{N/2^{L_1-1}}) \\ 0 & \text{otherwise} \end{cases}$$

In the last convolutional layer we then use that matrix again that contains only ones in its first column, and zeros in the other columns. Then $\mathcal{A}^y = \phi^{L_1,1,\gamma} = \psi^{L_1,1,1}$, i.e.,

$$\varphi(\mathcal{A}^y)_{d_1,\dots,d_{N/2^{L_1-1}}} = \begin{cases} 1 & \text{if } d_i \in \{i_\gamma\}_{\gamma=1,\dots,r} \text{ for all } i \\ & \text{and } (d_1, d_3, \dots, d_{N/2^{L_1-1}-1}) \\ & = (d_2, d_4, \dots, d_{N/2^{L_1-1}}) \\ 0 & \text{otherwise} \end{cases}$$

This means that $[\varphi(\mathcal{A}^y)]$ is a diagonal matrix that has a 1 on the diagonal position k if k has only digits from $\{i_\gamma\}_{\gamma=1,\dots,r}$ in the numeral system of base $M^{2^{L_2-1}}$, and 0 otherwise. This matrix has dimension $M^{N/2} \times M^{N/2}$, so in that numeral system the matrix indices can be expressed using $N/2^{L_2-1} = N/2^{L_2}$ digits. Therefore there are $r^{N/2^{L_2}}$ different representations that contain only digits from $\{i_\gamma\}_{\gamma=1,\dots,r}$. So $[\varphi(\mathcal{A}^y)]$ is a diagonal matrix with $r^{N/2^{L_2}}$ non-zero entries and hence $\operatorname{rank}[\varphi(\mathcal{A}^y)] = r^{N/2^{L_2}}$ for this weight configuration.

□

For the same reasons as discussed in the proof of Lemma 2.6.4, the minors of $[\varphi(\mathcal{A}^y)]$ can be considered as polynomials with the weights as variables. With Lemma 2.6.3 it follows from Lemma 2.7.2 that there exists a weight configuration

such that one of the $r^{N/2^{L_2}} \times r^{N/2^{L_2}}$ minors of $[\varphi(\mathcal{A}^y)]$ is not zero. Let p be the polynomial that represents one of these minors. This polynomial p cannot be the zero polynomial, as there exists a weight configuration, where it does not vanish. On the other hand, whenever $p(w) \neq 0$ for some weights w, then $\text{rank}[\varphi(\mathcal{A}^y)] \geq r^{N/2^{L_2}}$ by Lemma 2.6.3, and therefore $\text{CP-rank}(\varphi(\mathcal{A}^y)) \geq r^{N/2^{L_2}}$ Lemma 2.3.2.

This lets us now prove Theorem 2.4.2:

Proof of Theorem 2.4.2. Let S be the set of weight configurations that represent a L_1-SPN function which can also be expressed by the L_2-SPN model with $r^{(2)}_{L_2-1} < r^{N/2^{L_2}}$.

Let $\mathcal{A}^y$ be the representing tensor of the L_1-SPN for some weight configuration w, and p a polynomial with the properties described above. Let $S' = \{w \mid p(w) = 0\}$. Then we have $\text{rank}[\varphi(\mathcal{A}^y)] \geq r^{N/2^{L_2}}$ except on S'. Then we apply Lemma 2.7.1 to obtain $r^{(2)}_{L_2-1} \geq \text{CP-rank}(\varphi(\mathcal{A}^y)) \geq \text{rank}[\varphi(\mathcal{A}^y)] \geq r^{N/2^{L_2}}$ except on S'. Therefore we have $S \subseteq S'$. Given that $p \not\equiv 0$, by Lemma 2.3.1, S' is a null set, which proves that S is a null set as well. □

2.8. Comparison with the Original Proof

2.8.1. Proof Structure

Unlike the original proof, the above version is much easier to formalize, for both the fundamental and the generalized theorem of network capacity (Theorem 2.4.1 and Theorem 2.4.2). The reasons are the same for both theorems; I will discuss the fundamental theorem (Theorem 2.4.1) here as an example.

The original proof applies one monolithic induction to a large part of the proof. This induction not only proves the existence of a weight configuration with $\text{rank}[\mathcal{A}^y] \geq r^{N/2}$ (as in Lemma 2.6.2), but it also proves that this inequality holds almost everywhere. As a consequence the induction is simultaneously concerned with tensors, matrices, ranks, polynomials and the Lebesgue measure.

The above version is more modular and can therefore be split in smaller proofs more easily. The monolithic induction is split into two smaller inductions, namely Lemma 2.6.2 (involving only tensors) and Lemma 2.6.4 (stating that the minors of $[\mathcal{A}^y]$ are polynomials). The application of Lemma 2.6.3 and Lemma 2.3.1 in the end can be completely separated from the deep network induction.

Moreover, this restructured proof avoids some lemmas that are used in the original proof but are not yet formalized in Isabelle/HOL. For example, the matrix rank must only be computed for that one specific weight configuration here. To compute the rank of other weight configurations, the original proof uses the Kronecker product (the matrix analogue of the tensor product) and its property to multiply the rank.

2.8.2. Unformalized Parts

There are some statements in the original work that I did not formalize due to lack of time. Only the fundamental theorem of network capacity in the case of non-shared weights is formalized, i.e., the weight matrices in each branch may be different. In the original paper the non-shared case is discussed in the proof, and a note explains how the proof can be adapted to the shared case. Unfortunately, this is not easy to transfer to a formalization, because it would require to generalize all definitions and proofs such that they subsume both the shared case and the non-shared case.

Furthermore, I completely ignore the representational layer in my formalization, because the transfer of the theorems of network capacity to the network including the representational layer can be done independently as described in the original work by Cohen et al.

2.9. Generalization Obtained from the Restructuring

2.9.1. Algebraic Varieties

The restructured proofs as formulated above shows the same results as in the original work of Cohen et al. But the restructuring allows for an easy generalization, of both the fundamental and the generalized theorem of network capacity. I will discuss the latter as an example here. Looking at the end of the proof again, we observe that S' is not only a null set, but the zero set of a polynomial $p \not\equiv 0$, which is a stronger property. Moreover we know exactly how p is constructed (by induction over the L_1 network). For example we can determine the degree of p depending on the parameters of the network. This allows us to derive more properties of the set S' and hence for $S \subseteq S'$.

The zero sets of polynomials and their properties are well studied: An entire mathematical area called *algebraic geometry* is dedicated to these sets. In the language of algebraic geometry the zero sets of polynomials are called *algebraic varieties*. More generally, an algebraic variety is a set of common zeros of a set of polynomials:

Definition 2.9.1. A set $V \subseteq \mathbb{R}^n$ is a (real) *algebraic variety* if there exists a set P of polynomials such that

$$V = \{x \in \mathbb{R}^n \mid p(x) = 0 \text{ for all } p \in P\}.$$

2.9.2. Tubular Neighborhood Theorems

Although being the zero set of a polynomial $\not\equiv 0$ is mathematically stronger than being a null set, the difference is subtle. The following results from algebraic geometry [8, 23] are helpful:

Theorem 2.9.1. *Let $W \subset S^m$ be a real algebraic variety defined by homogeneous polynomials of degree at most $D \geq 1$ such that $W \neq S^m$. Then we have for $0 < \epsilon$:*

$$\frac{\mathrm{vol}_m T_{\mathbb{P}}(W, \epsilon)}{\mathcal{O}_m} \leq 2 \sum_{k=1}^{m-1} \frac{m}{k} (2D)^k (1+\epsilon)^{m-k} \epsilon^k + \frac{m\mathcal{O}_m}{\mathcal{O}_{m-1}} (2D)^m \epsilon^m$$

where $\mathcal{O}_m := \mathrm{vol}_m(S^m)$ denotes the m-dimensional volume of the sphere S^m and $T_{\mathbb{P}}(W, \epsilon)$ is the tubular ϵ-neighborhood of W using the projective distance.

Theorem 2.9.2. *Let V be the zero set of homogeneous multivariate polynomials $f_1, \ldots, f_s$ in $\mathbb{R}^n$ of degree at most D. Assume V is a complete intersection of dimension $m = n - s$. Let x be uniformly distributed in a ball $B^n(0, \sigma)$ of radius σ around the origin 0. Then:*

$$\mathbf{P}\{\mathrm{dist}(x, V) \leq \epsilon\} \leq 2 \sum_{i=0}^{m} \binom{n}{s+i} \left(\frac{2D\epsilon}{\sigma}\right)^{s+i}$$

These theorems need some further explanation about what they mean and how they can be applied to the convolutional arithmetic circuits. I will discuss them step by step, starting with Theorem 2.9.1. We consider a subset W of the unit sphere $S^m \subset \mathbb{R}^{m+1}$. We will see later that the inequality can be extended to the entire $\mathbb{R}^{m+1}$, which corresponds to the weight space of the L_1-SPN (i.e $\mathbb{R}^{m+1} = \mathbb{R}^n$).

W being a real algebraic variety means that it is the zero set of a set of polynomials, i.e., the set where all of these polynomials vanish. In the proof of Theorem 2.4.2 we used the polynomial p to define S', which will be the only defining polynomial for W. Although p does have zeros outside of S^m, these are not relevant for Theorem 2.9.1, which completely ignores the surrounding $\mathbb{R}^{m+1}$. So we use $W := S' \cap S^m$.

Moreover, Theorem 2.9.1 requires p to be *homogeneous*, i.e., all terms of the polynomial must have the same degree. This is true for p, because of its construction: The inputs are constant with respect to the weights, so they are all homogeneous polynomials. In a convolutional layer they are multiplied by a weight and added up, so the degree of each term is increased by 1, which results again in homogeneous polynomials. In a pooling layer two (or more) homogeneous polynomials are multiplied, which doubles (or multiplies) the degrees of each term, still resulting in homogeneous polynomials. So p is homogeneous.

Being homogeneous implies one more useful property: The zero set S' of p is invariant under multiplication by any real number λ. If $x \in \mathbb{R}^n$ is a zero of p, then $0 = \lambda^d \cdot p(x) = p(\lambda \cdot x)$ where d is the degree of p, because d is the degree of each term of p as well. So we can describe S' as

$$S' = \{\lambda \cdot x \mid x \in W \text{ and } \lambda \in \mathbb{R}\}. \tag{2.4}$$

In particular, $W \neq S^p$. Otherwise S' would be the entire $\mathbb{R}^n$.

Since all conditions are fulfilled, the theorem gives us an upper bound to the volume of the tubular neighborhood $T_{\mathbb{P}}(W, \epsilon)$ of W. Because of equation 2.4, this bound can be transferred to the set

$$\{x \mid \text{dist}(x, S') \leq \epsilon \cdot |x|\}.$$

This set has infinite volume, but the ratio of the volume of W to the volume of S^m is the same as the ratio of this set to the surrounding space if restricted to a ball $B^{m+1}(0, \sigma)$ ball of arbitrary size σ. This set $\{x \mid \text{dist}(x, S') \leq \epsilon \cdot |x|\}$ is similar to a tubular ϵ-neighborhood, but the "tube" becomes larger proportionally to the distance from the origin. This "tube" becoming larger proportionally to the distance from the origin is a fairly accurate approximation of the set S' in the discrete weight space of a computer with floating point arithmetic. Floating point numbers have the property that they are proportionally less precise the larger they are.

Before we calculate the bound in Section 2.9.3, take a look at Theorem 2.9.2. This theorem is similar to the first one, but it applies to a subset V of the entire space $\mathbb{R}^n$. We set $s = 1$ and $f_1 = p$, so $V = S'$. A main difference is also that V is assumed to be a *complete intersection*. I will omit an explanation what this means, because S' is definitely not a complete intersection. From personal correspondence with the author Martin Lotz though, I know that in the case $s = 1$ this condition can be avoided. This could be proved using a similar trick as used in the proof of Theorem 2.9.1, which does not work as nicely for more than one polynomial.

Then Theorem 2.9.2 (or rather a version of this theorem that requires $s = 1$ but no complete intersection) gives us an upper bound for $\mathbf{P}\{\text{dist}(x, V) \leq \epsilon\}$, when x is uniformly distributed in $B^n(0, \sigma)$. This corresponds to the volume of the tubular ε-neighborhood of $V = S'$ intersected with $B^n(0, \sigma)$, which is a good approximation of the set S' in the discrete weight space of a computer with fixed point arithmetic as illustrated in Figures 2.3b and 2.3c.

2.9.3. Calculation of the Bounds

To calculate the bounds we must answer two questions first: What is the degree of p and what is a reasonable value for ε?

As discussed above degree can be calculated by induction over the network. We take the truncated network models and obtain the results for the shallow and the deep network model as special cases. The inputs have degree 0 when interpreted as polynomials of the network weights. Each convolutional layer increases the degree by one, and each pooling layer with a two-branching doubles the degree. Before the L_1th pooling layer that merges all branches there are L_1 convolutional layers and $L_1 - 1$ pooling layers. The polynomial representing the network up to that

layer therefore has a degree of

$$\underbrace{(\ldots((0+1)\cdot 2+1)\cdot 2+\ldots)\cdot 2+1}_{L_1 \text{ ones and } L_1-1 \text{ twos}} = 2^{L_1-1}+2^{L_1-2}+\cdots+2^0 = 2^{L_1}-1$$

Then the L_1th pooling layer multiplies the degree by the number of branches that it merges, which is $N/2^{L_1-1}$. Finally the last convolutional layer increases the degree by 1 again. So any polynomial that represents one of the entries of $\mathcal{A}^y$ has a degree of

$$(2^{L_1}-1)\cdot N/2^{L_1-1}+1 = 2N - N/2^{L_1-1}+1$$

The calculation of the $r^{N/2^{L_2}} \times r^{N/2^{L_2}}$ minors further raises the degree of the polynomials to the power of $r^{N/2^{L_2}}$. This results in a degree for p of

$$D = (2N - N/2^{L_1-1}+1)^{r^{N/2^{L_2}}}$$

This degree is minimal for $L_1 = \log_2 N$ and $L_2 = \log_2 N - 1$ where it is equal to $D = (2N-1)^{r^2}$. According to the original work, realistic values are $N = 65,536$ and $r = 100$, which yield a degree of

$$D \simeq 2^{170,000}$$

What is a reasonable value for ε? Since it is more realistic, I discuss the case of floating point numbers first. A widely used format is the double-precision format, which occupies 8 bytes (64 bits). It uses 1 bit for the sign, 11 bits for the exponent and 52 bits for the fraction. The fraction part stores the digits of the number, while the exponent part determines where to set the binary point (the analogous of the decimal point). This way of storing numbers leads to high precision for smaller numbers and less precision for larger numbers. More precisely for some $x \in \mathbb{N}$, numbers between -2^x and 2^x can be stored with a precision of at least 2^{x-52}, since the fraction contains 52 digits.

Theorem 2.9.1 is a statement about points on the unit sphere, whose coordinates can only take values between -1 and 1. Therefore a reasonable value would be $\epsilon = 2^{-52}$.

If we allow 8 bytes for the fixed point values as well, the ratio between the highest possible number and the precision is 2^{64}. So for Theorem 2.9.2 we can set $\epsilon/\sigma = 2^{-64}$.

These calculations show that the degree of p is extremely high, while reasonable values for ϵ are relatively small. Moreover both Theorem 2.9.1 and 2.9.2 are useful only if the right-hand side is smaller than 1. Otherwise the statements are trivial. If we want the right-hand sides to be smaller than 1, we need at least $d < 1/\epsilon$ in Theorem 2.9.1 and $D < \sigma/\epsilon$ in Theorem 2.9.2, which is completely unrealistic given the calculations above.

This result lets me conjecture that the shallow network investigated here is more

expressive than assumed. The set S' is a null set but it might be still very densely packed such that it is large from a practical perspective. Unfortunately the entire analysis is build upon many inequalities, which might be too generous. Therefore a mathematical result estimating the size of S' with a lower bound seems to require a completely different approach.

3. Isabelle/HOL: A Proof Assistant for Higher-Order Logic

Isabelle is a generic *proof assistant*, which is an interactive software tool with a graphical user interface for the development of computer-checked formal proofs. Isabelle is generic in that it supports different formalisms, such as first-order logic (FOL), higher-order logic (HOL), and Zermelo-Fraenkel set theory (ZF). These formalisms are based on a built-in metalogic, which is based on an intuitionistic fragment of Church's simple type theory. On top of the metalogic, HOL introduces a more elaborate variant of Church's simple type theory, including the usual connectives and quantifiers. A list of Isabelle symbols can be found in Appendix A.

Generally, proof assistants have a modeling language to describe the algorithms to be studied, a property language to state theorems about these algorithms, and a proof language to explain why the theorems hold. For Isabelle, the modeling language and the property language are almost identical. For the purpose of this thesis, I do not differentiate the two and summarize them as Isabelle's metalogic (Section 3.3), extended by the HOL formalism (Section 3.4), whereas Isabelle's proof language is presented separately (Section 3.8).

3.1. Isabelle's Architecture

Isabelle's architecture follows the ideas of the theorem prover LCF in implementing a small inference kernel that ensures the correctness of proofs. This architecture is designed to minimize the risk of accepting incorrect proofs. Trusting Isabelle amounts to trusting its inference kernel, but also trusting the compiler and runtime system of Standard ML, the programming language in which the kernel is written, the operating system, and the hardware. Moreover, care is needed to ensure that a formalization proves what it is supposed to prove, because the specification of the mathematical statement can contain mistakes.

The inference kernel specifies Isabelle's metalogic, which is based on a fragment of Church's simple type theory (1940), which is also referred to as higher-order logic. The metalogic contains a polymorphic type system, including a type `prop` for truth values. Unlike first-order logic, where formulas and terms are distinguished, formulas in higher-order logic are just terms of type `prop`. Likewise, what is called a predicate in first-order logic, is just a function. Functions can be arguments for other functions and it is permitted to quantify over them.

HOL is the most widely used instance of Isabelle, which extends the metalogic to a variant of Church's simple type theory by introducing more quantifiers and connectives, as well as introducing additional axioms such as the axiom of choice and the axiom of function extensionality.

3.2. The Archive of Formal Proofs

The *Archive of Formal Proofs* (AFP) [20] is an online library of Isabelle formalizations contributed by Isabelle users. It is organized in the way of a scientific journal maintained by the Isabelle developers, meaning that submissions are refereed and published as articles. An AFP article contains a collection of Isabelle *theories*, i.e., files with definitions, lemmas and proofs. As of 2016, the AFP collected more than 300 articles about diverse topics from computer science, logic and mathematics.

3.3. Isabelle's Metalogic

All Isabelle formalisms are based on its metalogic, which introduces types and terms in the style of a simply typed λ-calculus as described by Church in 1940.

3.3.1. Types

Types are either type constants, type variables, or type constructors:

- Type constants represent simple types such as `nat` for the natural numbers, or `real` for the real numbers.
- Type variables are placeholders for arbitrary types. For better readability, I use the letters α, β, γ for type variables in this thesis instead of the Isabelle syntax `'a`, `'b`, `'c`.
- Type constructors build types depending on other types, for example the type constructor `list` represents lists such as lists of natural numbers `nat list`, or lists of real numbers `real list`. Type constructors with more than one argument use parentheses around the arguments, e.g., `(`α`,`β`) fun`. Type constructors are usually written in postfix notation and they associate to the left, e.g. `nat list list` the same as `(nat list) list`, representing lists of lists of natural numbers.

The type constructor `(`α`,`β`) fun`, which is normally written as $\alpha \Rightarrow \beta$, represents functions from α to β. Functions in Isabelle are total, i.e., they are defined on all values of the type α.

All functions in Isabelle have a single argument, but nesting the type constructor emulates function spaces of functions with two or more arguments, e.g., `nat` $\Rightarrow$ `nat` $\Rightarrow$ `real`, which is the same as `nat` $\Rightarrow$ `(nat` $\Rightarrow$ `real)`. A function of type `nat` $\Rightarrow$ `nat` $\Rightarrow$ `real` takes an argument of type `nat`, and returns a function of type `nat` $\Rightarrow$

`real`, which in turn takes an argument of type `nat`, and returns a real number. This is a principle known as *currying*.

3.3.2. Type Classes

Types can be organized in type classes. Type classes are defined by constants that contained typed must provide, and properties that contained types must fulfill. A type fulfills these requirements can be made an instance of that type class by specifying the constants and proving that the properties hold.

An example of a type class is the class `finite`. It requires no constants, and the defining property of that class is that the type's universe (i.e., the set of all values of this type) is finite. The boolean type `bool` can be registered as an instance of the class `finite` because it has a universe of only two values (`True` and `False`). The fact that `bool`'s universe is finite must be proved to instantiate it, though.

Type variables can also be restricted to a certain type class using the double colon syntax. The types that this type variable can be instantiated with are constrained to that type class. E.g., α`::finite` can be instantiated by types belonging to `finite`.

3.3.3. Terms

Terms are either variables, constants, function applications, or λ-abstractions:

- Variables (e.g., `x`) represent an arbitrary value of a type. Isabelle distinguishes between schematic and non-schematic variables. Non-schematic variables represent fixed, but unknown values, whereas schematic variables can be instantiated with arbitrary terms. When stating a theorem and proving it, variables are usually fixed. After the proof, the theorem's variables are treated as schematics such that other proofs can instantiate them arbitrarily. Syntactically, schematic variables are marked by a question mark, e.g. `?x`.

- Constants (e.g., `0`, `sin`, `op<`) represent a specific value of a type. In particular, variables and constants can also represent functions.

- Function application is written without parentheses surrounding or commas separating the arguments, i.e., `f x y` for a function `f` and arguments `x` and `y`. In fact, functions are always unary: Applying a function to multiple arguments is represented by a sequence of unary function application, a principle known as *currying*. A function `f` mapping two arguments of type α and β to a value of type γ is of type $\alpha \Rightarrow \beta \Rightarrow \gamma$, which the same as $\alpha \Rightarrow$ `(`$\beta \Rightarrow \gamma$`)`. Function application associates to the left, i.e., `f x y` is the same as `(f x) y`. Therefore, the first (unary) application in the term `f x y` invokes `f` on `x`, yielding a value `f x` of type $\beta \Rightarrow \gamma$. The second application invokes `f x` on `y`, yielding a value of type γ.

Using syntactic sugar, some functions can be written as infix operators (e.g., `x + y` instead of `plus x y`).

- A λ-abstraction builds a function from a term. E.g., if `g` is of type $\alpha \Rightarrow \alpha \Rightarrow \beta$, then λ`x. g x x` is of type $\alpha \Rightarrow \beta$.

A term can be marked to have a certain type using a double colon, e.g., `x::nat` denotes a variable `x` that represents a natural number. For terms that are not annotated the type will be inferred from context using a variant of Hindley-Milner's type inference algorithm.

3.4. The HOL Object Logic

The HOL formalism extends the Isabelle metalogic to a more elaborate version of higher-order logic, introducing additional axioms, the usual connectives and quantifiers, and basic types.

3.4.1. Logical Connectives and Quantifiers

Isabelle's metalogic introduces a restricted collection of connectives and quantifiers. It uses unusual syntax for these logical symbols to leave the usual mathematical syntax open to the extending formalisms such as HOL. The universal quantifier is $\bigwedge$, the implication is $\Longrightarrow$, and equality is $\equiv$. These connectives and quantifiers operate on the truth values of type `prop`.

The implication $\Longrightarrow$ associates to the right such that multiple premises $\mathtt{P}_1$, ..., $\mathtt{P}_n$ of a conclusion `Q` can be written as $\mathtt{P}_1 \Longrightarrow \ldots \Longrightarrow \mathtt{P}_n \Longrightarrow \mathtt{Q}$.

HOL introduces another type `bool` with values `True` and `False`. A constant `Trueprop` maps these values to values of type `prop`. The constant `Trueprop` is inserted automatically by Isabelle's parser and it is usually hidden from the user. Therefore, I will not write it explicitly in my thesis either.

HOL defines connectives and quantifiers operating on the type `bool`. The most important connectives are "not" $\neg$, "and" $\wedge$, "or" $\vee$, "implies" $\longrightarrow$ and "equivalent" $\longleftrightarrow$. The existential and universal quantifier are written as $\exists$`x.` and $\forall$`x.`, respectively, followed by the expression that is quantified over.

The difference between $\bigwedge$ and $\forall$, $\equiv$ and `=`, as well as $\Longrightarrow$ and $\longrightarrow$ is largely technical, caused by the difference between metalogic's type `prop` and the HOL type `bool`. For this thesis, the two sets of symbols can safely be thought as being equivalent.

3.4.2. Numeral Types

HOL supports frequently used numeral types such as `nat` for natural numbers, `int` for integers and `real` for real numbers.

A natural number in Isabelle is either `0` or `Suc n` where `n` is a natural number (`Suc` standing for 'successor'). Hence, the sequence of natural numbers is

```
0, Suc 0, Suc (Suc 0), Suc (Suc (Suc 0)), ...
```

To simplify this construction for the user, it is possible to write `0`, `1`, `2`, `3`, ... instead.

3.4.3. Pairs

Given two types α and β, one can construct the Cartesian product of the two, written as $\alpha \times \beta$. The values of this type are pairs of two values, where the first one is of type α and the second one is of type β. The pair of two values `a` and `b` is written as `(a,b)`. The same syntax can be used for triples `(a,b,c)` and larger tuples.

The components of a pair can be extracted using the functions `fst` ("first") and `snd` ("second"), e.g., `fst (a,b) = a` and `snd (a,b) = b`.

3.4.4. Lists

Lists in Isabelle/HOL are ordered, finite collections of values. All of these values must have the same type α, the list type is then called α `list`. Lists are equivalent to what is called an array (of variable length) in many programming languages.

The most simple list is the empty list, which is notated `[]`. Longer lists can be constructed using the operator `#`, which prepends an element to an existing list. If for example `xs` is a list and `x` is a new element, then `x # xs` is the list `xs` headed by `x`. Accordingly, the list with elements 1, 2, 3 would be represented as `1 # (2 # (3 # []))`.

Some important functions that operate on lists are `hd`, `tl`, `last`, `butlast`, `!`, `take` and `drop`. The function `hd` ("head") returns the first element of a list. The function `tl` ("tail") returns the remaining list without the first element. Similarly, `last` returns the last element, and `butlast` returns the list without the last element. The operator `!` returns the $(n+1)$st element of a list. The function `take n` will return the first n elements of a list, while `drop n` will return the list without the first `n` elements. For example if `xs` is the list 1, 2, 3, then

```
hd xs = 1
tl xs = 2 # (3 # [])
last xs = 3
butlast xs = 1 # (2 # [])
xs ! 1 = 2
take 2 xs = 1 # (2 # [])
drop 2 xs = 3 # []
```

3.4.5. Sets

The type α `set` denotes sets of elements from type α. Sets are often described using set comprehensions, e.g., `{x. P x}` is the set of all `x` for which `P x` is true.

Instead of `P x` more complex expressions are possible, whereas `x` must be a simple variable in this syntax. The empty set is written as `{}`.

The infix operator $\in$ tests whether a value is contained in a set, i.e. the expression `a` $\in$ `{x. P x}` is equivalent to `P a`. Set comprehension and the $\in$-operator map the types α `set` and $\alpha \Rightarrow$ `bool` isomorphically to each other.

Sometimes it is useful to have more complex terms in the front part of a set comprehension. For these cases there is the syntax `{f x | x. P x}`. For example the set of all squared prime numbers is `{x * x | x. prime x}`. If there is no side condition, one can use the constant `True`, e.g., the set of all square numbers is `{x * x | x::nat. True}`.

3.5. Outer and Inner Syntax

Isabelle distinguishes between two syntactic levels: the inner and outer syntax. All of the above, i.e., types and terms, including formulas, are inner syntax. The inner syntax is marked by enclosing them in quotation marks `"`. If a piece of inner syntax only consists of a single identifier, the quotation marks can be omitted, i.e. instead of `"x"`, `"0"` and `"nat"`, we can write `x`, `0` and `nat`.

The definitional principles and the proof language explained in the following chapters use the outer syntax, and all expressions of terms and types with more than a single identifier are enclosed in quotation marks.

3.6. Type and Constant Definitions

Isabelle/HOL provides various ways to introduce types and constants conveniently. It is possible although not recommended to introduce them by axiomatization. Axioms are usually avoided because they can easily contradict each other, i.e., lead to inconsistent specifications. In this section, I focus on ways to introduce types and constants more safely.

3.6.1. Typedef

The command **typedef** is a way to introduce types. It creates types from non-empty subsets of the universes of other types. The following definition introduces a type for unordered pairs. In contrast to ordered pairs, the elements (a, b) and (b, a) are identified.

typedef α `unordered_pair = "{A::`α` set. card A` $\leq$ `2` $\wedge$ `A` $\neq${}}"`

This creates a type `unordered_pair` which is parametrized by a type α. Each value of this type corresponds to a non-empty set with at most two elements, which is the usual mathematical definition of unordered pairs.

3.6.2. Inductive Datatypes

Another way to introduce types is the command **datatype**. It creates an algebraic datatype freely generated by the specified constructors. The constructors may be parametrized, even by values of the type currently being defined. This leads to a recursive nature of the values, which can be considered as finite directed graphs. The introduced types follow the motto "No junk, no confusion":

- No junk: There are no values in the model of the datatype that do not correspond to a term built from a finite number of applications of the constructors.
- No confusion: Two different constructor terms (terms consisting only of constructors) are always interpreted as two distinct values.

The following code defines binary trees that store values of type α in their leaf nodes:

```
datatype α tree = Leaf α | Branch "α tree" "α tree"
```

The name of the type is `tree`, and its constructors are `Leaf` and `Branch`. The simplest tree is `Leaf a`, where `a` is of type α. More complex trees can be build up using the newly introduced keyword `Branch`, which requires two arguments of type `tree`. An example for a `nat tree` is `Branch (Leaf 3) (Branch (Leaf 5) (Leaf 2))`.

Incidentally lists are also defined as an inductive datatype.

3.6.3. Plain Definitions

The commands **definition** and **abbreviation** can introduce shorter names for longer expressions. The following code defines a predicate for prime numbers:

```
definition prime :: "nat ⇒ bool" where
"prime p = (1 < p ∧ (∀m. m dvd p ⟶ m = 1 ∨ m = p))"
```

Here, `dvd` stands for 'divides'.

The command **abbreviation** works similarly on the surface but is only syntactic sugar. The command **definition** on the other hand is a disciplined form of axiom and introduces a new symbol internally. However, at the level of abstraction of this thesis, we can safely ignore this difference.

3.6.4. Recursive Function Definitions

The command **definition** can only be used for non-recursive definitions. In some cases it is desirable to invoke the function under definition on the right-hand side. For this purpose we can use **fun**.

The following function `sum` adds all numbers in a list of reals:

```
fun sum :: "real list ⇒ real" where
"sum [] = 0" |
"sum (x # xs) = x + sum xs"
```

This definition distinguishes two cases (separated by a vertical bar |): The sum of an empty list is 0. For a non-empty list we can assume that it has a first element `x` and the rest of the list `xs`. Invoking `sum` recursively we get the sum over the rest of the list and add the first element to get the sum over the entire list.

The commands **definition** and **fun** follow the definition principles of typed functional programming languages like ML.

3.6.5. Inductive Predicates

The command **inductive** introduces a predicate by an enumeration of introduction rules. Given these rules, Isabelle generates a least fixed point definition for this predicate.

The following declaration defines a predicate `even`, which is `True` for even numbers and `False` for odd numbers:

```
inductive even :: "nat ⇒ bool" where
zero: "even 0" |
step: "even n ⟹ even (n + 2)"
```

The resulting predicate `even` is the predicate which is `True` on the smallest set possible without violation the rules. In this way, **inductive** behaves like the logic programming language Prolog, which considers a statement false if it cannot be derived from the given rules (‘negation as failure‘).

Besides the introduction rules, an inductive predicate declaration also generates induction, case distinction and simplification rules.

3.7. Locales

A locale is a module that encapsulates a set of definitions, lemmas and theorems, which by default have a global scope. Locales are also useful to introduce shared side conditions to several theorems or lemmas, without repeating them in each theorem statement.

Group theory for example introduces a locale that fixes a group operator and a neutral element that must fulfill certain assumptions, namely the group axioms. In this way, these assumptions do not have to be repeated for every lemma. This locale can be introduced as follows:

```
locale group =
 fixes zero :: α ("0")
  and plus :: "α ⇒ α ⇒ α" (infixl "+" 65)
  and uminus :: "α ⇒ α" ("- _" [81] 80)
 assumes add_assoc: "(a + b) + c = a + (b + c)"
```

```
  and add_0_left: "0 + a = a"
  and left_minus: "- a + a = 0"
begin
 ...
end
```

Moreover, locales allow the assumptions and fixed variables to be instantiated elsewhere, e.g., the real numbers form a group and all group lemmas apply for them.

3.8. Proof Language

The statement of a lemma in Isabelle creates a *proof state*, a collection of statements that must be proved to show that the lemma holds. Isabelle provides various *tactics*, which are procedures that transform proof goals into zero or more new subgoals, ensuring that the original goal is a true statement if the new subgoals can be discharged. When tactic applications transformed the proof state into having no more subgoals, the proof is complete.

There are two ways to write proofs in Isabelle: **apply** scripts and Isar proofs. An **apply** script describes the proof backwards, starting with the proof goal, and applying tactics until the no more proof goals are left. The **apply** syntax only states the involved tactics and lemmas explicitly, but not the subgoals after each step.

In contrast, Isar proofs describe a proof in a forward and more structured way, from the assumptions to the proof goal. Isar is based on the natural deduction calculus, which is designed to bring formal proofs closer to how proofs are written traditionally.

3.8.1. Stating Lemmas

Lemmas can be stated using the commands **lemma** and **theorem**, which are technically equivalent but **theorem** marks facts of higher significance for human reader.

The commands are followed by an optional label for later reference and the lemma statement, e.g.,

```
lemma exists_equal: "∃y. x = y"
```

All free variables are implicitly universally quantified, i.e. the above abbreviates

```
lemma exists_equal: "⋀x. ∃y. x = y"
```

Alternatively, lemma statements can be divided in three sections as follows:

```
lemma prod_geq_0:
 fixes m::nat and n::nat
 assumes "0 < m * n"
 shows "0 < m"
```

The command **fixes** introduces types of variables. The command **assumes** states assumptions and **shows** states the conclusion. The sections **fixes** and **assumes** are optional. Multiple statements in a section can be concatenated by **and**.

These two variants to state lemmas are completely exchangeable: The above lemma statement is equivalent to

```
lemma prod_geq_0: "0 < (m::nat) * n ⟹ 0 < m"
```

3.8.2. Apply Scripts

The following proof shows a property of the function `rev`, which reverses the order of a list, in **apply** style:

```
lemma rev_rev: "rev (rev xs) = xs"
apply (induction xs)
apply auto
done
```

The lemma states that reversing a list `xs` twice will recover the original list. The command **lemma** assigns the name `rev_rev` to the lemma for later reference. Moreover, it creates a proof state with a single proof goal, which is identical to the lemma statement.

The first **apply** command invokes the tactic `induction`, which will use the standard list induction when applied to a list. This tactic transforms the goal `rev (rev xs) = xs` into two subgoals:

```
1. rev (rev []) = []
2. ⋀a xs. rev (rev xs) = xs ⟹ rev (rev (a # xs)) = a # xs
```

The first subgoal is the base case of the induction which states the property for the empty list. The second subgoal is the induction step. It states that for some `a` and some `xs` that fulfills the property, the list `a # xs` fulfills the property as well.

The second **apply** command invokes the tactic `auto`, which resolves both subgoals. The command **done** marks the end of the proof.

3.8.3. Isar Proofs

In Isar proofs, intermediate formulas on the way are stated explicitly, which makes Isar proofs easier to read and understand. Most of my formalization is written in Isar.

The structure of an Isar proof resembles the structure of the proof goal. A goal of the form $\bigwedge x_1 \ldots x_k.\ A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$ can be discharged using the following proof structure:

```
proof -
 fix x1 ... xk
 assume A1
```

```
  ⋮
  assume A_n
  have l_1: P_1 using ... by ...
  ⋮
  have l_n: P_n using ... by ...
  show B using ... by ...
qed
```

where $\texttt{P}_1, \ldots, \texttt{P}_n$ are intermediate properties, which are optionally assigned labels $\texttt{l}_1, \ldots, \texttt{l}_n$ for referencing the property.

Isar proofs are surrounded by the keywords **proof** and **qed**. The keyword **proof** can be optionally followed by a tactic, which is applied to the proof goal initially. A minus symbol (`-`) signifies no tactic application. Omitting the minus symbol applies a default tactic, which is chosen automatically depending on the proof goal. Variables and assumptions are introduced by **fix** and **assume**.

Intermediate formulas are introduced by the keyword **have**, whereas the last formula, which completes the proof goal, is introduced by **show**. The keywords **have** and **show** introduce proof goals and need to be followed by instructions how to discharge those. These instructions can either be a nested **proof** ... **qed** block or a proof method, which is a combination of one or more tactics such as `metis`, `auto` and `induction`.

A proof method is introduced by the keyword **by**. It is optionally preceded by a `using` command, which introduces facts (i.e. other lemmas or properties) as assumptions to the proof goal. For example, if a property `P` can be proved with the tactic `metis` using another property labeled `l`, we can write

```
have P using l by metis
```

Some tactics such as `metis` can take facts as arguments such that we can equivalently write

```
have P by (metis l)
```

The keywords **have** and **show** may be preceded by **then** to indicate that the previous property should be used in the proof search as well. If immediately preceded by the property `l`, we can abbreviate the above by

```
then have P by metis
```

3.8.4. An Example Isar Proof

The following proof shows that the tail of a list is one element shorter than the original list. Recall that the tail `tl xs` of a list `xs` is the list `xs` without its first element.

HOL is a logic of total functions, i.e. functions need to be defined on all arguments. A function value can be left unspecified, but it is often convenient to specify concrete default values.

For the special case of an empty list we have the default value `tl [] = []`. At first sight, it seems as if `length (tl xs) = length xs - 1` does not hold for the empty list. But this is an equation of type `nat`, and there is no `-1` in the type of natural numbers. Therefore calculations of type `nat` that would result in a negative value are assigned `0` instead, for example `0 - 1 = 0`. This might seem odd, but often it results in nice properties without inconvenient side conditions as in the following lemma:

```
1 lemma "length (tl xs) = length xs - 1"
2 proof (cases xs)
3  assume "xs = []"
4  then have "tl xs = []" by (metis List.list.sel(2))
5  then show "length (tl xs) = length xs - 1"
6   by (metis diff_0_eq_0 list.size(3) `xs = []`)
7 next
8  fix a as
9  assume "xs = a # as"
10  have "length as + 1 = length (a # as)"
11   by (metis One_nat_def list.size(4))
12   then have "length (tl xs) + 1 = length xs"
13   by (metis list.sel(3) `xs = a # as`)
14  then show "length (tl xs) = length xs - 1"
15    by (metis add_implies_diff)
16 qed
```

Following the informal proof above, we must distinguish two cases in the formal proof, too. This is done by applying the tactic `cases` on `xs`, which can be done directly after the keyword **proof**. This will split the proof goal into two subgoals, one assuming that `xs` is the empty list, the other assuming that `xs` is of the form `a # as` for some `a` and `as`. The two cases are separated by **next**, the assumptions are introduced by **assume** and the two necessary variables `a` and `as` are introduced by **fix**. For each case, a sequence of **have**/**then have** commands and a final **then show** explains the proof step by step. In this example, the proof method that is introduced by **by** is always `metis`, which is one of the most basic proof methods available. The name of the method `metis` is followed by the names of the lemmas that are necessary to complete the current proof step or alternatively a literal property enclosed by ``, e.g., `` `xs = []` ``. If a proof step is preceded by **then**, the previously proved property is also included in the proof search.

The text editor jEdit that is normally used for Isabelle development constantly runs the Isabelle process such that a proof method is immediately highlighted if it fails (Figure 3.1).

3.8.5. Theorem Modifiers

Theorem Modifiers such as `OF`, `of`, and `unfolded` alter or combine already proved lemmas in various ways. This simplifies the proof search and can make methods

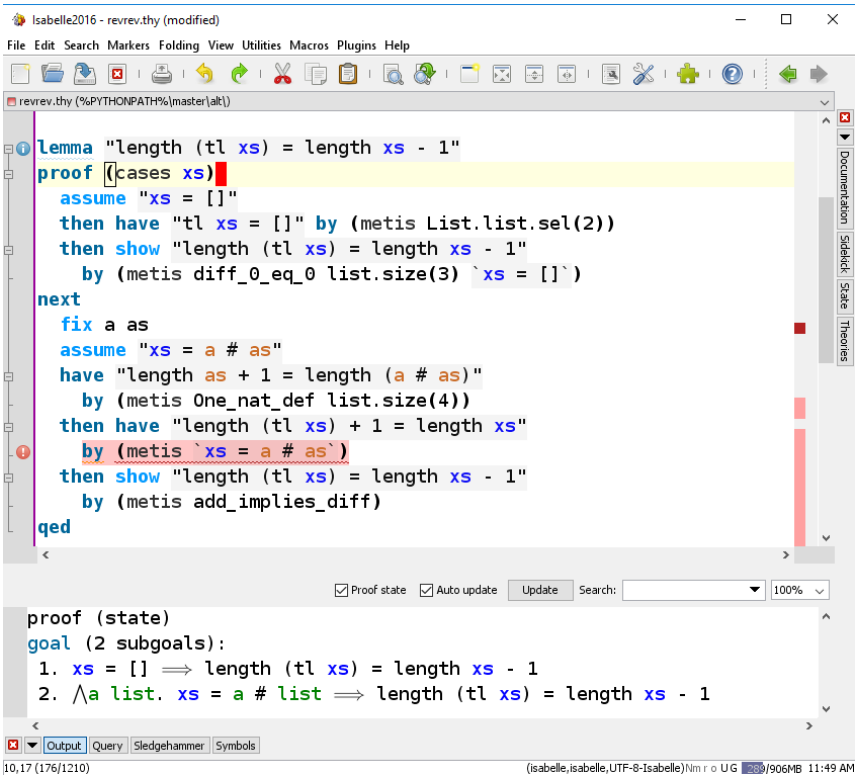


Figure 3.1.: The jEdit text editor is used for Isabelle development. Since a necessary lemma is missing in one of the `metis` calls, that line is highlighted in red.

succeed that normally would not.

The Lemma `add_implies_diff` in line 15 of the example above states `?c + ?b = ?a ⟹ ?c = ?a - ?b`. The question marks in front of the variables indicate that these variables are variables of the external lemma, and not variables of our proof, i.e., they can still be instantiated with any term. Instead of leaving that work to `metis` we can instantiate them using the modifier `of` as follows:

```
add_implies_diff[of "length (tl xs)" "1" "length xs"]
```

The terms instantiate the variables in order of appearance in the lemma. This modified lemma states

```
length (tl xs) + 1 = length xs ⟹ length (tl xs) = length xs - 1
```

and `metis` does not need to search for the right instantiations. Although in this example `metis` is able to find the proof without explicit values, in more complicated contexts it can drastically speed up `metis` or, in some higher-order cases, make the difference between unprovable and provable.

The modifier `OF` works similarly, but on the level of properties, not terms. For example, we can use `OF` to eliminate the premise `?c + ?b = ?a` from the

Lemma `add_implies_diff` as follows, given that `‘length (tl xs) + 1 = length xs‘` is already proved in line 12:

```
add_implies_diff[OF ‘length (tl xs) + 1 = length xs‘]
```

This modified lemma has no more premises and the variables are instantiated according to the property that was given to be unified with the premise. It states the following:

```
length (tl xs) = length xs - 1
```

Doing this would take all the work from `metis`, since the modified lemma is already identical with the proof goal.

The modifier `unfolded` modifies lemmas by expanding definitions or other equations, and can be used similarly to `of` and `OF`. Analogously, `unfolding` expands definitions, but in the proof goal. All of these keywords and some others are helpful when inspecting the details of a proof step and to manually assist the automated proof methods.

3.8.6. Sledgehammer

Sledgehammer is a subsystem of Isabelle that can find proofs automatically. It can be invoked on a proof (sub)goal to find relevant lemmas automatically and passes on this information to external automatic provers such as the superposition provers E, SPASS, Vampire and the SMT solvers CVC4, veriT, and Z3 [24, 7, 5]. If one of the external tools finds a proof, this proof is translated back into an expression that can be used to prove the goal inside Isabelle, i.e., without relying on the correctness of the external provers.

3.8.7. SMT Proofs

Satisfiability modulo theory (SMT) solvers are available in Isabelle through the `smt` proof method. It invokes the SMT solver Z3 and replays its proof in Isabelle, exploiting Z3's reasoning about equality, quantifiers, and linear arithmetic.

There are many different proof methods in addition to `metis`, such as `auto`, `simp`, `blast`, `force` and many more. While all these methods have there place and it is impossible to say that one is invariably superior to another, I want to highlight the method `smt` here, because I found it particularly useful in my formalization, especially when reasoning about sums or products of real numbers and existential quantification. At first I tried to avoid `smt`, because `smt` depends on external tools such as Z3, which are not under the control of the Isabelle developers such that future compatibility cannot be guaranteed. For this reason `smt` is not accepted in the *Archive of Formal Proofs* [20]. Nevertheless, it often showed up as the only one-line result of Sledgehammer.

Simple examples for the strengths of `smt` are rare, since they appear most frequently in complex contexts. But the following equation is an example for the

strengths of `smt` (for better readability displayed in standard mathematical typesetting):

$$\sum_{i\in I}\sum_{j\in J} a\cdot b\cdot f(i)\cdot g(j) = \left(\sum_{i\in I} a\cdot f(i)\right)\cdot\left(\sum_{j\in J} b\cdot g(j)\right)$$

where a, b are real numbers, I, J are arbitrary sets and f, g are real valued functions. The only one-line proof that Sledgehammer can find here is using `smt`. Slightly simplified, this one-line proof is

```
by (smt mult.assoc mult.commute setsum.cong setsum_product)
```

The lemmas `mult.assoc` and `mult.commute` are used to rearrange the factors on the left side. By lemma `setsum.cong`, the reordering is allowed inside of a sum, too. The lemma `setsum_product` transforms the two nested sums over a product into a product of sums.

It does not seem to be possible to solve it with other proof methods without performing several steps, which may easily take 10 minutes of manual work or more. Other proof methods fail, because they are not able to find the λ-expressions that are necessary to instantiate the lemmas.

3.9. Interactive Proof Development Workflow

The user's task in Isabelle/HOL is to specify sufficiently detailed proof steps for the automatic proof methods to fill in the missing steps. In my experience, one can estimate how large these steps must be. The more difficult task is to find the right lemmas and their names in the library that are needed for the proof. The most useful feature for this task is Sledgehammer.

Therefore, my workflow usually proceeds as follows: First, I decide what a reasonable next proof step might be. Isabelle includes counterexample generators, which can warn if the formula is provably false [4]. If the formula is plausible, I invoke Sledgehammer on it, supplying names of lemmas as hints that I believe might be useful. If I believe to know all lemmas necessary, I might try a concrete proof method or the command **try0**, which tries the most frequently used proof methods at once, but in most cases the proof steps are too large to know all necessary lemmas directly.

If Sledgehammer fails, I usually try to provide a slightly smaller proof step or an additional lemma that I think is useful, and give it to Sledgehammer again. Another method to find lemmas is **find_theorems** which can be used to search for lemmas and theorems by name, contained functions or by a pattern, for example `"_ < _ ⟹ _ < _"` where `_` is a wildcard.

If I know concrete instantiations of the involved lemmas or how the lemmas should be applied, I often use `OF`, `of`, `unfolded` or `unfolding`. This helps Sledgehammer avoid search paths that are not useful. Especially if variables must be instantiated with complex lambda-expressions, Sledgehammer usually fails to find

a proof without me providing these expressions using `of` or `OF`. On the other hand `OF` etc. are also useful to discover problems, such as noticing typos in the proof goal.

4. Formalization of Deep Learning in Isabelle/HOL

My formalization [1] provides a formal proof of Theorem 2.4.1 for the case of non-shared weights as presented in Chapter 2. The formalization does not rely on any axioms beyond those that are built into Isabelle/HOL. It has an approximate total size of 6500 lines, including

- a new tensor library (about 1800 lines)
- an extension for submatrices and ranks of Thiemann and Yamada's matrix library [29] (about 900 lines),
- an extension of Lochbihler and Haftmann's polynomial library [16] (about 900 lines)

To give an impression of my formalization, I compare an excerpt of Cohen et al.'s original work to the corresponding formal proof in Section 4.1.

My formalization is based on existing libraries for matrices, the Lebesgue measure and multivariate polynomials. Since my project requires an approach that can handle matrices of different dimensions easily, I decided to use Thiemann and Yamada's library [29] (Section 4.2). I developed a general-purpose library for tensors specifically for this project, because no other suitable tensor library exists (Section 4.3). The Lebesgue measure from Isabelle's probability library had to be adapted to the needs of my formalization (Section 4.4). The most suitable polynomial library is an unpublished project by Lochbihler and Haftmann [16], although many alternative libraries exist (Section 4.5).

A major challenge of the formalization of Theorem 2.4.1 is formalizing of the networks involved, and establishing their connection to tensor theory. Although tensors are used in the proof, the final formulation of the theorem is independent of tensor theory and therefore remains unaffected by any mistakes the tensor library may contain (Section 4.6).

4.1. A Comparison of Informal and Formal Proofs

To illustrate the process of formal proof development, I compare an excerpt from the original work by Cohen et al. [11, p. 21] to the corresponding lemma in my formalization.

The excerpt in question proves that rank$[\mathcal{A}] \leq$ CP-rank$(\mathcal{A})$ for a tensor $\mathcal{A}$ of even order (Lemma 2.3.2). Let $Z :=$ CP-rank $\mathcal{A}$. By definition of the CP-rank, there are vectors $\mathbf{a}_{z,1}, \ldots, \mathbf{a}_{z,N}$ and real numbers λ_z such that

$$\mathcal{A} = \sum_{z=1}^{Z} \lambda_z \cdot \mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N} \tag{4.1}$$

First, Cohen et al. show that

$$\text{rank}\,[\mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N}] = 1$$

I omit their proof for this equation here. With the linearity of the matricization and the sub-additivity of the matrix rank, it follows that:

$$\begin{aligned}
\text{rank}\,[\mathcal{A}] &= \text{rank}\left[\sum_{z=1}^{Z} \lambda_z \cdot \mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N}\right] \\
&= \text{rank} \sum_{z=1}^{Z} \lambda_z \cdot [\mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N}] \\
&\leq \sum_{z=1}^{Z} \text{rank}\,\lambda_z \cdot [\mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N}] = Z = \text{CP-rank}(\mathcal{A})
\end{aligned}$$

Before we can analyze the formalization of this proof, we must understand how the CP-rank is defined in my formalization. In this thesis, I define the CP-rank as the smallest Z that fulfills equation 4.1 (Definition 2.3.1). This is a possible formal definition as well but this approach makes the proofs of many properties cumbersome.

Instead, I divide the definition into three steps: I define the tensors with a CP-rank of at most 1, then the tensors with a CP-rank of at most $j \in \mathbb{N}$, and based on that the CP-rank itself.

The predicate `cprank_max1` is true for all tensors with a rank of at most 1:

```
inductive cprank_max1::"α::ring_1 tensor ⇒ bool" where
 order1: "order A ≤ 1 ⟹ cprank_max1 A" |
 higher_order: "order A = 1 ⟹ cprank_max1 B ⟹ cprank_max1 (A ⊗ B)"
```

The rule `order1` states that all tensors of order 0 and 1, i.e., scalars and vectors, have a CP-rank of at most 1. The rule `higher_order` states that a tensor of CP-rank at most 1 multiplied by a vector still has a CP-rank of at most 1. Inductively, this defines all tensors of the form $\lambda_z \cdot \mathbf{a}_{z,1} \otimes \cdots \otimes \mathbf{a}_{z,N}$.

The predicate `cprank_max j` is true for all tensors with a rank of at most `j`:

```
inductive cprank_max :: "nat ⇒ α::ring_1 tensor ⇒ bool" where
 cprank_max0: "cprank_max 0 (tensor0 ds)" |
 cprank_max_Suc: "dims A = dims B ⟹ cprank_max1 A ⟹ cprank_max j B
  ⟹ cprank_max (Suc j) (A+B)"
```

The rule `cprank_max0` declares all zero tensors to have CP-rank at most `0`. The expression `tensor0 ds` denotes the zero tensor with dimensions `ds`. The rule `cprank_max_Suc` states that adding a tensor of CP-rank at most `1` and a tensor of CP-rank at most `j` yields a tensor of CP-rank at most `Suc j = j + 1`. The dimensions in each mode of the tensors must be equal (i.e., `dims A = dims B`), because the addition of tensors of different dimensions is unspecified.

Finally, we define the CP-rank of a tensor `A` as the least `j`, such that `cprank_max j A`:

```
definition cprank :: "α::ring_1 tensor ⇒ nat" where
 "cprank A = (LEAST j.  cprank_max j A)"
```

Equipped with these definitions, we can analyze the formal version of the informal proof above:

```
 1 lemma matrix_rank_le_cprank_max:
 2  fixes A :: "(α::field) tensor"
 3  assumes "cprank_max r A"
 4  shows "mrank (matricize I A) ≤ r"
 5 using assms
 6 proof (induction rule:cprank_max.induct)
 7  fix ds :: "nat list"
 8  have "matricize I (tensor0 ds)
 9    = 0_m (dim_r (matricize I (tensor0 ds)))
10          (dim_c (matricize I (tensor0 ds)))"
11   using matricize_0 by auto
12  then show "mrank (matricize I (tensor0 ds)) ≤ 0"
13   using eq_imp_le vec_space.rank_0I by metis
14 next
15  fix A B::"α tensor" and j::nat
16  assume "dims A = dims B"
17  assume "cprank_max1 A"
18  assume "cprank_max j B"
19  assume "mrank (matricize I B) ≤ j"
20  have "mrank (matricize I A) ≤ 1"
21   using ‘cprank_max1 A‘ matricize_cprank_max1 by auto
22  have "mrank (matricize I (A + B))
23    ≤ mrank (matricize I A) + mrank (matricize I B)"
24   using matricize_add vec_space.rank_subadditive dims_matricize
25    mat_carrierI mat_index_add(2) ‘dims A = dims B‘ by metis
26  then show "mrank (matricize I (A + B)) ≤ Suc j"
27   using ‘mrank (matricize I A) ≤ 1‘ ‘mrank (matricize I B) ≤ j‘
28   by linarith
29 qed
```

The premise of this lemma is `cprank_max r A`, which subsumes the case that `r` is equal to the CP-rank of `A`. In this way, the lemma is more general than the informal Lemma 2.3.2.

Moreover, the function `matricize` is a generalization of the matricization described in Definition 2.3.2. It takes a set of natural numbers as first parameter `I` that determines which tensor modes are mapped to matrix rows, whereas all other modes are mapped to the matrix columns. If `I` is the set of even numbers, the definitions coincide. A more general statement often simplifies a formal proof.

The formal proof applies an induction over the CP-rank in line 6, whereas the informal proof leaves the induction implicit using the ... notation. The induction rule `cprank_max.induct` generated by the definition of `cprank_max` creates a subgoal for the base case of the zero tensor and a subgoal for the induction step:

```
1. ⋀ds. mrank (matricize I (tensor0 ds)) ≤ 0
2. ⋀A B j. dims A = dims B ⟹ cprank_max1 A ⟹ cprank_max j B ⟹
   mrank (matricize I B) ≤ j ⟹ mrank (matricize I (A + B)) ≤ Suc j
```

Both subgoals are proved separately.

For the base case, we show in line 8 that the matricization of a zero tensor is a zero matrix ($\mathbf{0}_m$) and deduce in line 12 that the matrix rank (`mrank`) must be `0`.

The induction step assumes two tensors `A` and `B` of equal dimensions, where `A` has a CP-rank of at most `1`, and `B` has a CP-rank of at most `j` for some natural number `j` (lines 16 to 18). The induction hypothesis is that the matricization of `B` has a rank of at most `j` (line 19).

We show that the rank of the matricization of `A` is at most `1` (line 20). Moreover, we show that the rank of the matricization of `A + B` is smaller than or equal to the sum of ranks of separate matricizations of `A` and `B` (line 22). By the induction hypothesis, it follows in line 26 that the rank of the matricization of `A + B` is at most `j + 1`, which concludes the proof.

4.2. Available Matrix Libraries

The proof of Theorem 2.4.1 relies crucially on matrix theory, for which different Isabelle libraries exist. My formalization requires in particular a formalization that can reason about matrices of different dimensions, in particular cope with submatrices, and implements the matrix determinant as well as the matrix rank.

4.2.1. Isabelle's Multivariate Analysis Library

Isabelle includes a large multivariate analysis (MVA) library of frequently used definitions and lemmas, including some properties of vectors and matrices in a collection of theories. The type that represents vectors is `('a,'b) vec`, which depends on two other types α and β. The type variable α stands for the type of the entries. The type variable β determines the dimension of the vector by the size of its universe. The type definition of `('a,'b) vec` requires β to be of class *finite*, which means that the universe of β is finite. The size of this finite universe is

the dimension of the vectors and the values of type β are used as indices for the vectors. For convenience, the type `(α,β) vec` is also abbreviated as `α^β`.

This approach seems to be a bit unnatural, it would seem to be more appropriate to define the vector dimension using a natural number instead of a type. Unfortunately this is not possible, as type definitions may only depend on other types, but not on values of variables. In higher-order logic (unlike in some stronger logics, such as dependent type theory), types cannot depend on terms.

This vector definition allows to define matrices as vectors of vectors, i.e., `α^β^γ`, which abbreviates `((α,β) vec,γ) vec`.

This definition of matrices cannot be used when matrices of different dimensions are involved, or if the dimension of a matrix depends on a variable in the proof (called *term variables*).

4.2.2. Sternagel and Thiemann's Matrix Library

Another approach to matrices can be found in the matrix library by Sternagel and Thiemann [28]: The authors define matrices as lists of lists, i.e., `α list list`. This definition has two other problems, though. Firstly, the definition cannot control whether all columns have the same length, which should be the case for all matrices. So this property must be added as a premise when the type is used. Secondly, the matrix dimensions cannot be determined in some edge cases. Normally the number of rows can be determined by the length of the first column. However, if the number of columns is zero, the (theoretical) number of rows cannot be determined. Considering $n \times 0$ matrices may seem to be nonsensical at first, but it can be useful for inductions over the matrix dimension.

Moreover, Sternagel and Thiemann's library does not contain some definitions and lemmas that are required for my formalization. Therefore I attempted to transfer lemmas from the Isabelle's MVA library to Sternagel and Thiemann's library. For example, the following property of the determinant and the rank does not exist in Sternagel and Thiemann's library, but can be easily derived in Isabelle's MVA library:

```
lemma det_rank:
fixes A::"real^(β::finite)^(β::finite)"
shows "det A ≠ 0 ⟷ rank A = nrows A"
```

The lemma states that for a real square matrix, a non-zero determinant is equivalent to full rank in terms of the MVA matrix definition. The function `nrows` returns the number of rows, which is equal to the size of the universe of β.

To transfer this lemma to Sternagel and Thiemann's definition, we first need analogous definitions of the determinant, the rank and the number-of-rows function, which we call `det'`, `rank'` and `nrows'`. We attempt to use the above lemma to prove the following:

```
lemma det'_rank':
```

```
fixes A'::"real list list"
assumes "square_mat A'"
shows "det' A' ≠ 0 ⟷ rank' A' = nrows' A'"
```

We have to a add the premise `square_mat A'`, meaning that `A'` is a square matrix because in Sternagel and Thiemann's definition this premise is not ensured by the type. We denote matrices of type `real^β^β` (MVA definition) as `A` and matrices of type `real list list` (Sternagel and Thiemann's definition) as `A'`.

To connect the two definitions, we need a function `T` that maps a matrix `A'` of type `real list list` to its corresponding matrix `A` of type `real^β^β`. Although we can name all of these functions the same using polymorphism, `T` represents one function for every instantiation of β.

For example, if we instantiate β with a type `three` that has three values, we get a specific instance of `T` of type

```
real list list ⇒ real^three^three
```

This specific instance of `T` can obviously map only those arguments to a reasonable function value that represent a 3×3 matrix. More generally, if `CARD` β denotes the size of β's universe, the function `T::(real list list ⇒ real^β^β)` can only be reasonably specified for arguments that represent `CARD` β $\times$ `CARD` β matrices. For this reason, properties such as

```
det (T A') = det' A'
rank (T A') = rank' A'
nrows (T A') = nrows' A'
```

with `T` of type `real list list ⇒ real^β^β` can only be proved if `A'` represents a square matrix and has the same size as the universe of β. This results in the following lemma:

```
lemma
 fixes T::"real list list ⇒ real^β^β"
  and A'::"real list list"
 assumes "CARD(β) = nrows' A'"
  and "square_mat A'"
 shows "det (T A') = det' A'"
  and "rank (T A') = rank' A'"
  and "nrows (T A') = nrows' A'"
```

Combined with the lemma `det_rank` from above with `A = T A'`, this lemma yields:

```
lemma det'_rank'_provisory:
 fixes A'::"real list list"
 assumes "CARD(β) = nrows' A'"
  and "square_mat A'"
 shows "det' A' ≠ 0 ⟷ rank' A' = nrows' A'"
```

This lemma is almost our desired lemma `det'_rank'`, the only difference being the additional premise `CARD(β) = nrows' A'`. Note that the type β only appears in this premise, not elsewhere in the lemma, and not implicitly as the type of a variable. Therefore, this premise amounts to stating that there is a type β whose universe has size `nrows' A'`.

From an outside perspective, it is obvious that there is such a type β except for the case `nrows' A' = 0`, because we can introduce a type from any non-empty set using the **typedef** command (Section 3.6.1). However, Isabelle does not allow to introduce types inside of a proof.

An extension of the axiom system called *local typedef* that would allow this kind of type introduction is currently being developed [21]. In particular, it enables us to remove the undesired premise from the lemma `det'_rank'_provisory` and to obtain the lemma `det'_rank'`.

The main challenge of this approach is to proof that the definitions of `det`, `rank` etc. for both matrix types are essentially the same in the sense of `det (T A') = det' A'`. In particular for as complicated definitions as the definition of `det`, this proof can be cumbersome. Moreover I had difficulties to understand how to employ the local typedef extension.

These difficulties led me to discard the idea of using local typedefs. Nevertheless, using local typedefs for this purpose is possible, as demonstrated in a formalization of the Perron-Frobenius theorem [14].

4.2.3. Thiemann and Yamada's Matrix Library

Since it was hidden in an AFP entry about the Jordan normal form, I only later discovered the matrix library by Thiemann and Yamada [29]. It completely subsumes the results of the matrix library by Sternagel and Thiemann and of Isabelle's MVA library, but it is based on a new definition of matrices:

```
typedef α mat =
 "{(nr, nc, mk_mat nr nc f)
   | (nr :: nat) (nc :: nat) (f :: nat × nat ⇒ α).  True}"
```

Unlike Sternagel and Thiemann's definition, this definition explicitly contains the matrix dimensions `nr` (number of rows) and `nc` (number of columns). This solves the problem of determining the number of rows in $n \times 0$ matrices. Moreover, instead of lists a function `f` is used to store the entries of the matrix. This function `f` maps a pair of natural numbers (the indices) to the according entry. The function `mk_mat` (whose definition I omit here) overwrites the entries of `f` that are outside of the bounds of `nr` times `nc` with a default value. If the definition allowed arbitrary values outside of the bounds, we would end up with multiple values that represent the same matrix.

One advantage of using a function instead of a list to store the values of the matrices is the similarity to the matrix definition from Isabelle's MVA library. It

allowed Thiemann and Yamada to copy the existing proofs from there and adapt them to their definition. The local typedef extension did not exist at the time, which is why they were not able to transfer the lemmas directly, but a direct transfer is also possible [14].

There are two properties of matrices that are not contained in Thiemann and Yamada's library but necessary for my formalization: the matrix rank and submatrices.

I defined a function `submatrix` with three arguments: the original matrix of type α `mat`, a set of row indices of type `nat set`, and a set of column indices of type `nat set`. It returns the submatrix obtained by deleting all rows and columns whose indices are not contained in the given sets.

While the rank can be defined in various other ways, the two defining properties that are most relevant for my formalization are:

- The rank of a matrix is the dimension of the space spanned by the columns of the matrix.
- The rank of a matrix is the maximum amount of independent columns of the matrix.

I used the first property as the definition and proved the second property as a lemma. Moreover, I formalized the fact that the matrix rank is larger than any submatrix with determinant $\neq 0$.

4.3. Design of My Tensor Library

A formalization of tensors in Isabelle has not been published before, although there is an AFP entry called 'Tensor' by Prathamesh [26]. Despite the name, this library discusses the Kronecker product, which is the equivalent of the tensor product on matrices. More precisely, the Kronecker product of two matrices A and B can be described as the mapping

$$(A, B) \mapsto [A \otimes B]$$

i.e., the Kronecker product is the matricization of the tensor product, treating the matrices as order 2 tensors.

In principle, this approach of representing tensors by their matricization is sufficient, since the entries of the matricization are the entries of the tensor. But it is only sufficient if the tensor dimensions are always known, since they cannot be determined from the matrix. I found it much easier to work with a proper definition of tensors.

One could found the definition of the `tensor` type on their matricization, but it turns out to be easier to use their vectorization instead. Vectorization is similar to matricization, but the entries of the tensor are put into a vector instead of a matrix. More precisely, given an order N tensor of dimension M_i in mode i, the

value of at position $d_1, \ldots, d_N$ is written into the vector at position $d_1 + M_1 \cdot (d_2 + M_2 \cdot (\cdots + M_{N-1} \cdot d_N))$.

As in Thiemann and Yamada's matrix definition, the tensor values must contain their order and dimensions. These can be specified in form of a list `ds`, whose length is the order and whose entries are the dimensions in the respective mode:

```
typedef α tensor =
 "{(ds,vs) | (ds::nat list) (vs::α list). length vs = listprod ds}"
```

Unlike in Thiemann and Yamada's matrix definition, I used a list rather than a function to store the tensor values. Therefore I do not need to specify default values for the out of bounds entries, but I must specify the length of the list instead, which is `listprod ds`, i.e., the product of all mode dimensions.

A function representing the tensor values was extremely valuable in the proofs, though, which is why I defined such a function on top of this type definition. It would have been equally possible to define the tensor type based on a function and define the list representation on top, e.g.,

```
typedef α::zero tensor =
 "{(ds,f) | (ds::nat list) (f::nat ⇒ α). ∀i ≥ listprod ds. f i = 0}"
```

Both type definitions are essentially equivalent and neither has considerable advantages. The above definition requires α to be of class `zero` such that `0` can be used as a default value, but other default values such as the underspecified value `undefined` would be possible without requiring a type class.

When introducing a type, we often need to ask ourselves which type classes they could belong to. Type classes can be helpful, because they include lemmas and definitions, which can save a lot of work. For this tensor definition, we should especially consider type classes representing algebraic structures, such as rings, groups, and monoids. There are two basic approaches to implement these algebraic structures:

- In the approach without carrier sets the algebraic structure has to apply to the entire universe of the type (as implemented in Isabelle's group theory library).
- The second approach works with so-called carrier sets, which are a subset of the type universe that the algebraic structure applies to (as implemented in Isabelle's algebra library). This second approach is the more general one, but a little harder to use.

Both approaches define a hierarchy of algebraic structures with increasingly restrictive axioms:

- semigroup:
  associativity $(a + b) + c = a + (b + c)$

- cancellative semigroup:
  semigroup + cancellation properties $a + c = b + c \Longrightarrow a = b$ and $c + a = c + b \Longrightarrow a = b$
- monoid:
  semigroup + neutral element e, i.e., $e + a = a$ and $a + e = a$
- group:
  monoid + inverse elements $-a$, i.e., $(-a) + a = e$ and $a + (-a) = e$
- commutative versions of each of the above, i.e., $a + b = b + a$

The approach without carrier sets is satisfactory for tensor multiplication, because tensor multiplication is defined for any two tensors. Tensor multiplication forms a monoid, which means it is associative and has a neutral element. I instantiated the according type class `monoid_mult` in my formalization.

Unfortunately, addition can be reasonably defined only for tensors of the same order and dimensions. Restricted to these, the addition is a commutative group, but this cannot be captured by the approach without carrier sets.

It is desirable to find a definition of tensor addition that generalizes to tensors of different orders or dimensions and preserves the group axioms. But whatever rule one chooses, the tensor addition is at most a commutative monoid (using some arbitrary zero tensor as neutral element).

Tensor addition cannot be generalized without violating the cancellation property $a + c = b + c \Rightarrow a = b$: Let e be the neutral element. Let e' be a zero tensor of a different order. Then we have $e' + e' = e' = e + e'$, but $e' \neq e$, which contradicts the cancellation property. Therefore the set of all tensors can never be instantiated as a group.

The approach from Isabelle's algebra library on the other hand allows us to define the tensors of identical order and dimensions as carrier sets and to prove that each of those sets forms a commutative group. Since this was irrelevant for my formalization, I did not implement this, but it would be helpful to complete my tensor library.

4.4. Adapting the Formalization of the Lebesgue Measure

The Lebesgue measure as defined in Isabelle's probability library raised similar difficulties as the matrix definition of Isabelle's MVA library. The Lebesgue measure is defined using the type α`::euclidean_space measure`, meaning that the sets being measured must be sets of a type from the type class `euclidian_space`. This type class fixes a finite basis such that it fixes especially the dimensionality of the space being measured. As for the matrices, the dimensionality is therefore parametrized by types. But the dimension of the space in my formalization depends

on the number of weights in the network, i.e., on a term variable. As discussed in Section 4.2.2, it is therefore impossible to use the according lemmas directly. The obvious alternative is to redefine the Lebesgue measure and to transfer lemmas either by finding analogous proofs or by transferring the results using local typedefs.

Fortunately, in this case there is no need to redefine everything. The measure `lborel` for the type `real` (i.e., for the one-dimensional case) can be reused to define a more flexible version. To this end I employed the definition of product measures from Isabelle's probability library. Since the multidimensional Lebesgue measure is the product measure of the one-dimensional Lebesgue measure, these two can be combined to redefine the multidimensional Lebesgue measure depending on a term variable which specifies the dimensionality:

```
definition lborel_f :: "nat ⇒ (nat ⇒ real) measure" where
  "lborel_f n = (Π_M b∈{..<n}. lborel)"
```

This version of the Lebesgue measure has one argument `n` of type `nat` which specifies the dimensionality of the space being measured. The measure itself is of type `(nat ⇒ real) measure`, i.e., it measures sets of functions mapping from `nat` to `real`, hence the name `lborel_f`, where `f` stands for "functional". These functions represent vectors of $\mathbb{R}^n$, the entries of the vector being the first `n` values of the function. All function values after that must be the default value `undefined`. Any set containing a function that violates this rule is not measurable. The syntax $\Pi_{M\_}\in\_.\_$ denotes the product measure over a finite number of measures, in this case the product of `n` times the `lborel` measure. From type inference it is clear that `lborel` denotes the one-dimensional Lebesgue measure.

The measure `lborel` for higher dimension is defined using the product measure as well, just in a slightly different way. Therefore, any proof for `lborel` can quite easily be transferred to `lborel_f`. Lemmas connecting different dimensionalities of `lborel_f` can mostly be easily obtained from lemmas about the product measure.

The notion "almost everywhere" from measure theory, meaning that a property holds for all $x \in \mathbb{R}$ except for a null set, is formalized as well in form of the `AE` quantifier. The syntax is `AE x in M. P x`, meaning that `P x` holds almost everywhere with respect to the measure `M`. In my formalization, this measure will be `lborel_f n` with `n` being the dimension of the space.

4.5. Formalization of Multivariate Polynomials

Isabelle/HOL does not yet include a standard library of multivariate polynomials. Multivariate polynomial libraries have been developed to support other formalization projects, but they are designed for the purposes of the specific project. I will present some libraries that are most relevant for my formalization, one of which I finally decided to extend and use.

4.5.1. Nested Univariate Polynomials

In the standard Isabelle libraries, there is no formalization of multivariate polynomials, but the formalization for univariate polynomials can be used for multivariate ones as well. These univariate polynomials are defined as follows:

```
typedef α poly = "f :: (nat ⇒ α::zero).  ∀∞n.  f n = 0"
```

The polynomials are identified with functions from `nat` to a type of class `zero`. The function values represent the coefficients of the polynomial at a given exponent. The type class `zero` simply claims that there must be an element called `0`, allowing us to use `0` in the definition. The syntax `∀∞n.` stands for "for all but finitely many n ...". So, the condition for a valid function is that it can be non-zero at only finitely many places, which forces the polynomials to have only finitely many terms, i.e., power series are not allowed.

This definition can be used for multivariate polynomials by nesting it. The type `poly` itself can be instantiated having the type class `zero`, using the zero polynomial as `0`. Then it is possible to use the type `(real poly) poly`, `((real poly) poly) poly`, etc, which means that the coefficients of the polynomials are polynomials as well. For the type `(real poly) poly` an example polynomial, expressed in traditional notation, follows:

$$(1 + 2y + 3y^2) + (2 - 4y)x + (3 + y^2)x^2$$

The names x and y are arbitrary. Note that any multivariate polynomial with two variables can be written this way. For the type `((real poly) poly) poly` the inner polynomials have polynomial coefficients as well, which results in multivariate polynomials with three variables and so on.

Unfortunately, this approach does not work for my formalization, since the number of variables depends on a term variable. It is impossible to reason about this nesting of types, such as "I need a type that is nesting `poly n` times."

4.5.2. Sternagel and Thiemann's Polynomial Library

Sternagel and Thiemann [27] formalized multivariate polynomials in a project particularly designed for execution, i.e., generating source code in a functional programming language from the Isabelle specification. For this reason, there are multiple values that in the standard view of multivariate polynomials would be the same, e.g., $x \cdot y + z$, $y \cdot x + z$ and $z + x \cdot y$ are considered different. To determine whether two values represent the same polynomial, they define a binary predicate, which determines whether two values represent the same polynomial. This is an obstacle for proof automation, which is designed to work with the default equality predicate `=`.

4.5.3. Lochbihler and Haftmann's Polynomial Library

My formalization builds on an unpublished multivariate polynomial library by Lochbihler and Haftmann [16]. It introduces a type constructor `poly_mapping`, which is similar to the function type constructor but allows only functions that are non-zero at only finitely many points:

```
typedef (α, β) poly_mapping =
 "{f :: (α ⇒ β::zero). finite {x. f x ≠ 0}}"
```

This type constructor is abbreviated using the infix $\Rightarrow_0$. In the definition of the type for multivariate polynomials, it is used twice: once for ensuring that each term contains only finitely many variables (no infinite products), and once for ensuring that a polynomial contains only finitely many terms (no infinite sums). The definition looks as follows:

```
typedef α mpoly = "UNIV :: ((nat ⇒₀ nat) ⇒₀ α::zero) set"
```

The keyword `UNIV` stands for the universe (i.e., the set of all values of that type) of the type `(nat ⇒₀ nat) ⇒₀ α::zero`. In this approach, variables are represented as natural numbers, which can be pictured as using x_0, x_1, x_2, etc. The functions of type `nat ⇒₀ nat` represent power products such as $x_0^3 x_1^1 x_2^5$, mapping each variable index to its exponent. The `poly_mapping` type ensures that all but finitely many exponents are `0`. The type α is the type of the coefficients, which is `real` in my formalization. So, the polynomials are represented by mappings from power products to their coefficients, while the `poly_mapping` type ensures that all but finitely many power products have the coefficient `0`.

The main theorem about polynomials that is needed for my formalization is Lemma 2.3.1, stating that the zero set of a multivariate polynomial $\neq 0$ is a Lebesgue null set. The proof (from [9]) uses an induction that is based on the fact that multivariate polynomials can be nested as described above (Section 4.5.1). I extended Lochbihler and Haftmann's library with lemmas discussing the type α `mpoly mpoly` and how it is equivalent to α `mpoly`. Towards this end, I added more basic lemmas, also introducing the function `vars` which determines the set of used variables in a polynomial. Moreover, I developed a powerful induction principle which states that instead of proving a property for polynomials directly, it suffices to show a property for one single variable, multiplication of two monomials, and addition (some additional constraints are possible).

Furthermore, the proof of Lemma 2.3.1 uses the fact that a univariate real polynomial $\neq 0$ has only finitely many zeros. This fact is formalized for the type `poly` from Isabelle's library, so it needed to be transferred to the type `mpoly` (adding the premise that the polynomial may have only one variable). This connection between the type `mpoly` and `poly` is part of my formalization.

4.5.4. Immler and Maletzky's Polynomial Library

As part of a formalization of Groebner bases [18], one more definition of multivariate polynomials was published too late to be used in my formalization. I mention it because it is supposed to become the standard formalization of multivariate polynomials.

However, the maximal number of variables to be used in the polynomial is determined by a type. Since the number of variables depends on a term variable in my formalization, it is impossible to use Immler and Maletzky's defintion.

4.6. Formalization of the Fundamental Theorem

The goal of this section is eventually to present the fundamental theorem of network capacity in its formalized version, but first I will discuss the involved definitions.

4.6.1. A Type for Convolutional Arithmetic Circuits

At the heart of this theorem are the convolutional arithmetic circuits, appearing in the shape of the deep and the shallow network model. I created a type `convnet` that can represent these kind of networks, and theoretically also the truncated network model. The type is based on the Figures 2.2a and 2.2b, but only formalizes the SPN, i.e., the representational layer is not included specifically.

Since the networks are trees, the formalization resembles the example of binary trees in Section 3.6.2. This tree structure is made out of three building blocks:

- Input nodes, which represent the leaf nodes of the tree. They are parametrized by the length of the input vector (corresponding to M in Figures 2.2a and 2.2b).
- Convolutional nodes, which appear inside the tree structure, but do not branch. They are parametrized either by a matrix that contains the weights, or by a pair of natural numbers that describe only the matrix size without storing concrete weights. Having these two options allows us using the same type for both networks with concrete weights and network templates without concrete weights.
- Pooling nodes, which allow branching. I decided to allow only binary branching, as higher-order branching is equivalent to stacking multiple binary pooling nodes on top of each other.

In Isabelle syntax the type is defined as follows:

```
datatype α convnet =
 Input nat |
 Conv α "α convnet" |
 Pool "α convnet" "α convnet"
```

The type parameter α can be used to reason about networks with concrete weights (using `real mat` for α) or network templates without concrete weights (using `nat × nat` for α).

This type definition allows us to describe the structure of the networks. It does not give any meaning to the parts on the networks. The function `evaluate_net` specifies how calculations in the networks are performed. The function `evaluate_net` takes two arguments: The first one is the network to be evaluated of type `real mat convnet`. The second argument are the input values of the network as a list of vectors (`vec list`). The output of the function is the output vector of the network. The definition looks as follows:

```
fun evaluate_net :: "real mat convnet ⇒ real vec list ⇒ real vec"
 where
 "evaluate_net (Input M) inputs = hd inputs" |
 "evaluate_net (Conv A m) inputs = A ⊗mv evaluate_net m inputs" |
 "evaluate_net (Pool m1 m2) inputs = component_mult
  (evaluate_net m1 (take (length (input_sizes m1)) inputs))
  (evaluate_net m2 (drop (length (input_sizes m1)) inputs))"
```

The operator $\otimes_{mv}$ multiplies a matrix with a vector. The function `component_mult` multiplies two vectors componentwise.

This function definition specifies the network calculations in a recursive case distinction. The base case is `Input`, which simply outputs the first given input vector. Normally the `inputs` list of the `Input` node should have length one, but if it does not, the remaining list entries are ignored.

The `Conv` node makes a recursive call and multiplies its response to the contained matrix. The `Pool` node makes one recursive calls to each branch. For this purpose the `inputs` list must be split into two. The expression `length (input_sizes m1)` calculates the correct amount of input vectors for the model `m1`. The functions `take` and `drop` split the `inputs` list in two halfs, the first half having that calculated length. Finally the `Pool` node calculates the componentwise product of the two incoming vectors.

Another important function operating on the `convnet` type is `insert_weights`. It connects the network templates without weights to the networks with weights. The first argument is the network to be filled with weights, i.e., of type `(nat × nat) convnet`. The second argument contains the weights in form of a `nat ⇒ real` function. Only the first few function values are used (i.e., as many as there are weights in the network), and the rest is ignored. I decided to use a `nat ⇒ real` function here instead of a list, since the Lebesgue measure `lborel_f` (Section 4.4) is also based on `nat ⇒ real` functions. The output of `insert_weights` is a network with the same structure storing the specified weights, i.e., a network of type `real mat convnet`. The function is defined as follows:

```
fun insert_weights
 :: "(nat × nat) convnet ⇒ (nat ⇒ real) ⇒ real mat convnet"
```

```
where
"insert_weights (Input M) w = Input M" |
"insert_weights (Conv (r0,r1) m) w = Conv
  (extract_matrix w r0 r1)
  (insert_weights m (λi. w (i+r0*r1)))" |
"insert_weights (Pool m1 m2) w = Pool
  (insert_weights m1 w)
  (insert_weights m2 (λi. w (i+(count_weights m1))))"
```

This function definition also makes a case distinction on the three network building blocks. In the base case `Input`, nothing has to be changed. However note that the argument is a network of type `(nat × nat) convnet`, whereas the output is of type `real mat convnet`, although they are syntactically identical. The `Conv` case uses the function `extract_matrix`, which produces a matrix containing the first `r0*r1` function values of `w`. Then it recursively calls the `insert_weights` function, but instead of using `w` itself, it shifts all values of `w` using the expression `λi. w (i+r0*r1)` such that the first `r0*r1` values cannot be reused. A similar shifting is done in the second recursive call of the `Prod` case. Here, I use a function `count_weights`, which calculates how many weights are contained in the left branch, i.e., how far the function values must be shifted such that no weights are used multiple times.

4.6.2. The Shallow and Deep Network Models

The type `convnet` could be used to describe all kinds of convolutional arithmetic circuits. In particular, we need a way to describe the deep and the shallow network model. To this end, I decided to use functions that generate the network structures depending on a set of parameters.

The shallow network

Using the `convnet` type, the shallow network looks as illustrated in Figure 4.1. The pooling layer with multiple branching must be formalized by multiple binary `Pool` nodes.

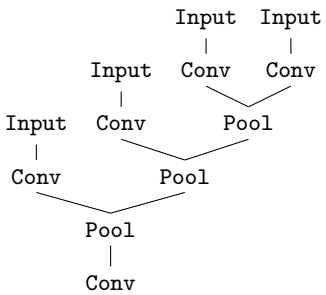


Figure 4.1.: Structure of the shallow model in the formalization

The definition of the generating function for the shallow network is divided into two parts. First, the auxiliary function `shallow_model'` produces the shallow model without the final `Conv` node:

```
fun shallow_model' where
"shallow_model' Z M 0 = Conv (Z,M) (Input M)" |
"shallow_model' Z M (Suc N)
 = Pool (shallow_model' Z M 0) (shallow_model' Z M N)"
```

The definition of `shallow_model'` takes the parameters `Z` (size of the output vectors of the first convolutional layer), `M` (size of the input vectors), and `N` (number of inputs) from Section 2.2 as arguments. More precisely the third parameter is equal to $N - 1$ for technical reasons.

The definition is recurses over the third argument, i.e., the number of inputs. The base case is `0`, which corresponds to $N = 1$ input node. The recursive case assumes that the third argument is `Suc N`, i.e., the successor of some number `N`. There are two recursive calls: one with third argument `0`, which creates the short left branch, and one with third argument `N`, which creates the longer right branch.

Finally the shallow model needs the final `Conv` node, which is done in the definition of `shallow_model`:

```
definition shallow_model where
"shallow_model Y Z M N = Conv (Y,Z) (shallow_model' Z M N)"
```

This definition has one additional parameter, `Y`, which corresponds to the length Y of the output vector as described in Section 2.2.

The deep network

The deep network consists of alternating convolutional and pooling layers. As for the shallow model, it makes sense for the recursive definition to employ an auxiliary function `deep_model'` that produces the deep network model without the last convolutional layer. But here, the two definitions of `deep_model'` and `deep_model` call each other recursively:

```
fun deep_model and deep_model' where
"deep_model' Y [] = Input Y" |
"deep_model' Y (r # rs)
 = Pool (deep_model Y r rs) (deep_model Y r rs)" |
"deep_model Y r rs = Conv (Y,r) (deep_model' r rs)"
```

The function `deep_model'` takes two arguments: the length of the output vector of the last layer, and the lengths of the output vectors of the other layers as a list bottom up. The function `deep_model` takes three arguments: the length of the output vector of the last layer, the length of the output vector of the before last layer, and the lengths of the output vectors of the other layers as a list bottom up.

To simplify the definition, the last and next-to-last layers are passed as separate arguments. When invoking the function, it makes more sense to combine all output vector lengths in one argument. Therefore I created the following abbreviations:

```
abbreviation "deep_model'_l rs == deep_model' (rs!0) (tl rs)"
abbreviation "deep_model_l rs  == deep_model (rs!0) (rs!1) (tl (tl rs))"
```

The function `deep_model_l` takes only one list as argument, which contains all output vector lengths bottom up. This corresponds to the values $Y, r_{L-1}, \ldots, r_0$, M from Section 2.2.

4.6.3. The Fundamental Theorem

The fundamental theorem is stated inside of a locale `deep_model_correct_params`, which puts up some requirements to the deep model structure:

```
locale deep_model_correct_params =
fixes rs::"nat list"
assumes deep:"length rs ≥ 3"
and no_zeros:"⋀r. r∈set rs ⟹ 0 < r"
```

This locale fixes the parameter `rs` for the deep model and requires `rs` to have at least three elements (corresponding to Y, r_0 and M) and to contain no zeros. With less than three elements, the corresponding deep model is shallower that the shallow model, and the theorem is no longer true. If one of the elements of `rs` is zero, the entire network can only produce the zero vector as output, therefore the theorem would be no longer true, too. In the original paper these two requirements are implicit in the definition of the deep model.

Based on the fixed `rs`, the locale defines the values `r` (corresponding to r), `N_half` (corresponding to $N/2$) and `weight_space_dim` (which defines the number of weights in the deep network, i.e., the number of dimensions of the weight space):

```
definition "r = min (last rs) (last (butlast rs))"
definition "N_half = 2^(length rs - 3)"
definition "weight_space_dim = count_weights(deep_model_l rs)"
```

I decided to use `N_half` instead of `N`, because it is much easier to reason about multiplication (e.g., `2*N_half`) than about division (e.g., `N div 2`). If I used `N`, I would need to thread the assumption that `N` is an even number through the entire formalization to use `N div 2` properly.

Equipped with these basic definitions, we are able to analyze the fundamental theorem in its formalized version:

```
theorem fundamental_theorem_network_capacity_v2:
 "AE weights_deep in lborel_f weight_space_dim.
   ¬(∃weights_shallow Z. Z < r ^ N_half ∧
(∀inputs. map dim_v inputs = input_sizes (deep_model_l rs) ⟶
```

```
evaluate_net (insert_weights (deep_model_l rs) weights_deep) inputs
= evaluate_net (insert_weights
 (shallow_model (rs ! 0) Z (last rs) (2*N_half-1))
weights_shallow) inputs))"
```

When comparing this with the formulation of Theorem 2.4.1, note that the formalized version does not define the set S and states that it is a null set. Instead, it states that the property to be in S does not hold almost everywhere.

The line `AE weights_deep in lborel_f weight_space_dim.` introduces an almost-everywhere quantification where `weights_deep` is the quantified variable. What follows is the negated property of being inside S, i.e., what follows the $\neg$ is the property to be in S as described in Theorem 2.4.1. There S is described as the set of weight configurations that represent a deep SPN function which can also be expressed by the shallow SPN model with $Z < r^{N/2}$.

The deep SPN function that is represented by our fixed weight configuration `weights_deep` is calculated using

```
evaluate_net (insert_weights (deep_model_l rs) weights_deep) inputs
```

for some valid list of input vectors `inputs`. The shallow SPN function to be compared to this depends on the parameters of the shallow network model Y, Z, M, N and on the weights of the shallow model. The SPN functions can only be the same if the input and output vector lengths are the same and if the number of inputs are the same, therefore we can use `rs ! 0` as the first, `last rs` as third `2*N_half-1` as forth argument of `shallow_model`.

The values for the second argument `Z` and the weights `weights_shallow` can be chosen arbitrarily, so we use an existential quantifier $\exists$ to introduce them. The only condition on `Z` is that `Z < r ^ N_half`.

Now, S is described as the set where the SPNs are equal, i.e., where they produce the same vectors for all valid inputs. Note that we could encounter invalid inputs, e.g., a list of the wrong length or vectors of a wrong size. The expression `map dim`$_v$ `inputs = input_sizes (deep_model_l rs)` excludes such invalid inputs.

It is not easy to see that the formalized theorem is equivalent to Theorem 2.4.1. Only a detailed study of the involved definitions can convince someone to accept that the formalization proves what it is supposed to prove. A formalization can never replace a paper proof, because a formal proof is difficult to read, and the human insight is hidden under the technical details.

4.7. Related Work

To the best of my knowledge this formalization is the first formal proof about deep learning. However, other machine learning algorithms including hidden Markov models [22], expectation maximization and support vector machines [3] have been formalized.

Concerning the theories that support my formalization, there are similar theories in the equally popular proof assistant Coq: The matrix theory of the Ssreflect project [15] is comprehensive, and it faces less challenges concerning the type system because Coq allows dependent types, i.e., types that depend on term variables. However, using dependent types for matrices turns out to be unwieldy in practice [13], which suggests that this is a general problem independent of the tool. The Coq tensor formalization by Boender [6] restricts itself to the Kronecker product on matrices, i.e., there is no type for tensors as provided in my formalization. Multivariate polynomials have been formalized to show the transcendence of e and π [2]. The Lebesgue measure only exists in a formalization for of the Lebesgue integral as part of a formalization of Markov's inequality [19].

5. Conclusion

This formalization is a case study of formalizing current research in the field of machine learning. It shows that the functionality and libraries of state of the art proof assistants such as Isabelle/HOL are up to the task. Admittedly, even the formalization of relatively short proofs such as the one presented here is labor-intensive. On the other hand, the process can not only lead to a computer verification of the result, but can also reveal new ideas and results, as the generalization obtained here.

Although convolutional arithmetic circuits are depth efficient, i.e., the functions generated by shallow networks are null set in the space of the functions generated by deeper networks, this generalization indicates that this null set might be extremely large when considering implementations that can naturally only use a finite subset of the real numbers. Concerning the theory of deep learning, this generalization raises the question as to whether the generally accepted notion of complete depth efficiency is the best way to determine the strength of a deep learning variant.

Concerning the development of proof assistants and formalizations, my work shows that we need ideas on how to solve the dependent type problem. In several areas (matrices, Lebesgue measure, polynomials) I ran into this issue, which was sometimes easier, sometimes harder to avoid. For Isabelle/HOL, a possible solution called "local typedefs" has recently been published [21]. But even in Coq, where dependent types are built into the formalism, they may "impose too many constraints on programming" [13] in practice.

Bibliography

[1] Alexander Bentkamp. Expressiveness of deep learning. *Archive of Formal Proofs*, November 2016. http://isa-afp.org/entries/Deep_Learning.shtml, Formal proof development.

[2] Sophie Bernard, Yves Bertot, Laurence Rideau, and Pierre-Yves Strub. Formal proofs of transcendence for e and pi as an application of multivariate and symmetric polynomials. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 76–87. ACM, 2016.

[3] Sooraj Bhat. *Syntactic foundations for machine learning*. PhD thesis, Georgia Institute of Technology, 2013.

[4] Jasmin Christian Blanchette. Counterexamples for Isabelle: Ground and beyond. *Symbolic Methods in Testing*, page 7, 2013.

[5] Jasmin Christian Blanchette, Sascha Böhme, Mathias Fleury, Steffen Juilf Smolka, and Albert Steckermeier. Semi-intelligible Isar proofs from machine-generated proofs. *Journal of Automated Reasoning*, 56(2):155–200, 2016.

[6] Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using Coq. *arXiv preprint arXiv:1511.01568*, 2015.

[7] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *International Conference on Interactive Theorem Proving*, pages 179–194. Springer, 2010.

[8] Peter Bürgisser, Felipe Cucker, and Martin Lotz. The probability that a slightly perturbed numerical analysis problem is difficult. *Mathematics of Computation*, 77(263):1559–1583, 2008.

[9] Richard Caron and Tim Traynor. The zero set of a polynomial. *WSMR Report*, pages 05–02, 2005.

[10] Nadav Cohen, Or Sharir, and Amnon Shashua. Deep simnets. *arXiv preprint arXiv:1506.03059*, 2015.

[11] Nadav Cohen, Or Sharir, and Amnon Shashua. On the expressive power of deep learning: A tensor analysis. *CoRR*, abs/1509.05009, 2015.

[12] Nadav Cohen and Amnon Shashua. Convolutional rectifier networks as generalized tensor decompositions. *arXiv preprint arXiv:1603.00162*, 2016.

[13] Maxime Dénes and Yves Bertot. Experiments with computable matrices in the Coq system. 2011.

[14] Jose Divasón, Ondřej Kunčar, René Thiemann, and Akihisa Yamada. Perron-Frobenius theorem for spectral radius analysis. *Archive of Formal Proofs*, May 2016. http://isa-afp.org/entries/Perron_Frobenius.shtml, Formal proof development.

[15] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A small scale reflection extension for the Coq system.* PhD thesis, Inria Saclay Ile de France, 2015.

[16] Florian Haftmann, Andreas Lochbihler, and Wolfgang Schreiner. Towards abstract and executable multivariate polynomials in Isabelle. In *Isabelle Workshop*, volume 201, 2014.

[17] Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In *Interactive Theorem Proving*, pages 135–151. Springer, 2011.

[18] Fabian Immler and Alexander Maletzky. Gröbner bases theory. *Archive of Formal Proofs*, May 2016. http://isa-afp.org/entries/Groebner_Bases.shtml, Formal proof development.

[19] Robert Kam. Case studies in proof checking. Master's thesis, San Jose State University, 2007.

[20] Gerwin Klein, Tobias Nipkow, and Lawrence Paulson. The Archive of Formal Proofs, 2010. http://isa-afp.org.

[21] Ondrej Kuncar and Andrei Popescu. From types to sets by local type definitions in higher-order logic. *Proc. ITP*, 2016.

[22] Liya Liu, Vincent Aravantinos, Osman Hasan, and Sofiene Tahar. On the formal analysis of HMM using theorem proving. In *International Conference on Formal Engineering Methods*, pages 316–331. Springer, 2014.

[23] Martin Lotz. On the volume of tubular neighborhoods of real algebraic varieties. *Proceedings of the American Mathematical Society*, 143(5):1875–1889, 2015.

[24] Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 1, 2010.

[25] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 689–690. IEEE, 2011.

[26] T.V.H. Prathamesh. Tensor product of matrices. *Archive of Formal Proofs*, January 2016. http://isa-afp.org/entries/Matrix_Tensor.shtml, Formal proof development.

[27] Christian Sternagel and René Thiemann. Executable multivariate polynomials. *Archive of Formal Proofs*, August 2010. http://isa-afp.org/entries/Polynomials.shtml, Formal proof development.

[28] Christian Sternagel and René Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. http://isa-afp.org/entries/Matrix.shtml, Formal proof development.

[29] René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. http://isa-afp.org/entries/Jordan_Normal_Form.shtml, Formal proof development.

A. List of Isabelle/HOL Symbols

Symbol	Description	Section
`.`	part of the quantifier syntax	3.4.1
`^`	exponentiation operator	
`::`	type or type class constraint	3.3.2, 3.3.3
`!`	nth element lookup for lists	3.4.4
`|`	case separator for `datatype` and `fun`	3.6.2, 3.6.4
$\wedge$	logical "and"	3.4.1
$\vee$	logical "or"	3.4.1
$\longrightarrow$	implication arrow	3.4.1
$\longleftrightarrow$	equivalence arrow	3.4.1
$\Rightarrow$	function type	3.3.1
$\Longrightarrow$	implication arrow (metalogic)	3.4.1
$\times$	pair type	3.4.3
$\otimes_{mv}$	matrix-vector-multiplication operator	4.6.1
`*`	multiplication operator	
`#`	list constructor (`Cons`)	3.4.4
`( , )`	pair syntax	3.4.4
`[]`	empty list (`Nil`)	3.4.4
`{}`	set syntax	3.4.5
$\forall$.	universal quantifier	3.4.1
$\forall_\infty$.	"for all but finitely many" quantifier	4.5.1
abbreviation	keyword for simple abbreviations	3.6.3
`AE` .	almost-everywhere quantifier	4.4
and	connector for multiple facts or symbols	3.8.1
`bool`	boolean type	3.4.1
`butlast`	returns a list without its last element	3.4.4
`component_mult`	componentwise multiplication of vectors	4.6.1
`convnet`	type for convolutional arithmetic circuits	4.6.1
$\exists$.	existential quantifier	3.4.1
datatype	keyword for inductive type definitions	3.6.2
defintion	keyword for simple definitions	3.6.3
`drop`	list function dropping the first n elements	3.4.4
`dvd`	divisibility predicate	3.6.3
`finite`	type class for types with a finite universe / predicate for finite sets	3.3.2, 4.5.3

`fst`	retrieves the first element of a pair	3.4.3
fun	keyword to define recursive functions	3.6.4
`hd`	returns the first element of a list	3.4.4
inductive	keyword to define inductive predicates	3.6.5
λ .	lambda expression	3.3.3
`last`	returns the last element of a list	3.4.4
`mat`	type for matrices	4.2.3
`nat`	type for natural numbers	3.4.2
$\Pi_{M\_\in\_.\_}$	product measure	4.4
`real`	type for real numbers	3.4.2
`snd`	retrieves the second element of a pair	3.4.3
`take`	list function taking the first n elements	3.4.4
`tl`	returns a list without its first element	3.4.4
typedef	keyword for defining types from sets	3.6.1
`vec`	type for vectors	