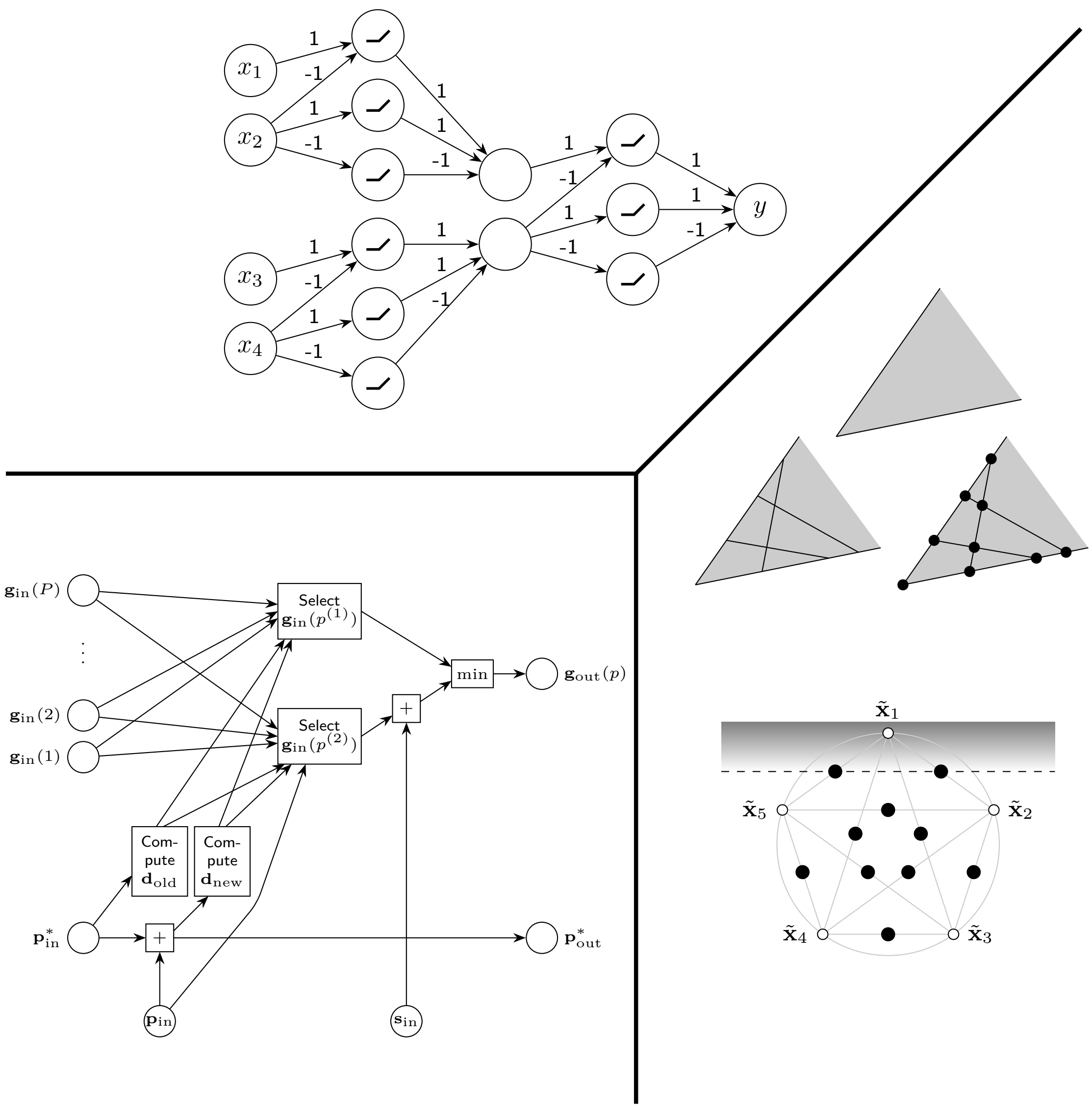
Christoph Hertrich

Facets of Neural Network Complexity



x_1
x_2
x_3
x_4
y
1
-1
$\mathbf{g}_{\text{in}}(P)$
$\mathbf{g}_{\text{in}}(2)$
$\mathbf{g}_{\text{in}}(1)$
Select $\mathbf{g}_{\text{in}}(p^{(1)})$
Select $\mathbf{g}_{\text{in}}(p^{(2)})$
min
$\mathbf{g}_{\text{out}}(p)$
Compute $\mathbf{d}_{\text{old}}$
Compute $\mathbf{d}_{\text{new}}$
$\mathbf{p}^*_{\text{in}}$
$\mathbf{p}^*_{\text{out}}$
$\mathbf{p}_{\text{in}}$
$\mathbf{s}_{\text{in}}$
$\tilde{\mathbf{x}}_1$
$\tilde{\mathbf{x}}_2$
$\tilde{\mathbf{x}}_3$
$\tilde{\mathbf{x}}_4$
$\tilde{\mathbf{x}}_5$

Facets of Neural Network Complexity

vorgelegt von
M. Sc.
Christoph Hertrich
ORCID: 0000-0001-5646-8567

an der Fakultät II – Mathematik und Naturwissenschaften
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. John Sullivan (TU Berlin)
Gutachter:	Prof. Dr. Martin Skutella (TU Berlin)
Gutachter:	Prof. Dr. Sebastian Pokutta (TU Berlin & Zuse-Institut Berlin)
Gutachter:	Prof. Dr. Amitabh Basu (Johns Hopkins University, Baltimore, USA)

Tag der wissenschaftlichen Aussprache: 2. März 2022

Berlin 2022

Abstract

Artificial **neural networks** are at the heart of some of the greatest advances in modern technology. They enable huge breakthroughs in applications ranging from computer vision via machine translation to speech recognition as well as autonomous driving and many more. However, we are still far away from a more rigorous theoretical explanation of these overwhelming success stories. Consequently, the development of a better mathematical understanding of neural networks is currently one of the hottest research topics in computer science. In this thesis we provide several contributions in that direction for the simple, but practically powerful and widely used model of feedforward neural networks with **rectified linear unit** (ReLU) activations.

Our focus is on various notions of what we call the **complexity** of such neural networks: how much computing resources (time, hardware, network size, etc.) are required to achieve a certain goal? Of course, such questions can be asked in various contexts. We identify and study the following three **facets** of complexity for neural networks with ReLU activations.

The first facet is neural networks' **expressivity**: What functions can be represented by certain neural network architectures? Even though this is such a fundamental question, very little is known so far. We make progress concerning the question whether the class of exactly representable functions strictly increases by adding more layers (with no restrictions on size). We also provide upper bounds on the number of neurons required to represent arbitrary piecewise linear functions with small-depth ReLU neural networks.

The second facet is neural networks' **computational power**. Here, we view neural networks as a model of computation, just like Boolean, or even closer, arithmetic circuits. We then investigate which network (or circuit) size is required to solve various problems, with a focus on combinatorial optimization problems. Even though this model is quite restrictive compared to other models of computation, we are able to show that comparably small neural networks can provably solve problems like the efficiently solvable Maximum Flow Problem or the NP-hard Knapsack Problem.

The third facet is neural networks' **training complexity**: How difficult is it to fit the weights of a neural network to training data? It is widely known that optimal solutions to the training problem are hard to obtain, which is why local optimization techniques like stochastic gradient descent are used in practice. We focus on the question whether the situation improves for fixed input dimension, leading to the paradigm of parameterized complexity analysis. We provide running time lower bounds in terms of W[1]-hardness results, proving that known brute-force strategies are essentially optimal. On the positive side, we extend a known polynomial-time algorithm for constant dimension and convex loss functions to a more general class of loss functions.

The **mathematical methods** used in this thesis include polyhedral theory, discrete and tropical geometry, mixed-integer and combinatorial optimization, as well as tools from complexity theory.

Zusammenfassung

Künstliche **neuronale Netze** sind der Schlüssel zu einigen der größten modernen Technologiefortschritten. Sie ermöglichen Durchbrüche in Anwendungen wie Computer Vision, maschinellem Übersetzen, Spracherkennung, bis hin zu autonomem Fahren und vielem mehr. Allerdings sind wir nach wie vor weit entfernt von rigorosen theoretischen Erklärungen dieser überwältigenden Erfolge. Daher ist die Entwicklung eines besseren mathematischen Verständnisses neuronaler Netze aktuell eines der wichtigsten Forschungsthemen der Informatik. In dieser Arbeit präsentieren wir einige Fortschritte in diese Richtung für das einfache, aber aus praktischer Sicht mächtige und viel verwendete Modell der Feedforward-Netze mit **ReLU**-Aktivierungsfunktionen.

Unser Fokus liegt auf verschiedenen Arten der **Komplexität** solcher neuronalen Netze: Wie viele Ressourcen (Zeit, Hardware, Netzwerkgröße etc.) werden benötigt, um ein bestimmtes Ziel zu erreichen? Selbstverständlich können solche Fragen in verschiedenen Kontexten gestellt werden. Wir identifizieren und untersuchen die folgenden drei **Facetten** der Komplexität für neuronale Netze mit ReLU-Aktivierungsfunktionen.

Die erste Facette ist **Expressivität**: Welche Funktionen können von bestimmten Netzwerkarchitekturen dargestellt werden? Obwohl es sich dabei um eine so fundamentale Frage handelt, ist bisher wenig bekannt. Wir präsentieren Fortschritte bezüglich der Frage, ob die Menge der exakt darstellbaren Funktionen echt größer wird, wenn wir mehr Schichten hinzufügen (ohne dabei die Größe zu beschränken). Wir beweisen außerdem obere Schranken an die Anzahl benötigter Neuronen, um beliebige stückweise lineare Funktionen mit neuronalen Netzen mit geringer Tiefe zu berechnen.

Die zweite Facette ist Berechnungskomplexität (**Computational Power**). Dafür betrachten wir neuronale Netze als ein Berechenbarkeitsmodell, ähnlich wie etwa Boolesche oder arithmetische Schaltkreise. Wir untersuchen, welche Netzwerkgröße nötig ist, um verschiedene Probleme, insbesondere aus der kombinatorischen Optimierung, zu lösen. Trotz der starken Eingeschränktheit dieses Modells zeigen wir, dass vergleichsweise kleine Netze beweisbar Lösungen für Probleme wie das Maximalflussproblem oder das NP-schwere Rucksackproblem finden können.

Die dritte Facette ist **Trainingskomplexität**: Wie schwer ist es, die Gewichte eines neuronalen Netzes an Trainingsdaten anzupassen? Bekanntermaßen ist das Finden exakter Lösungen ein schweres Problem, weshalb in der Praxis lokale Optimierungsverfahren wie etwa stochastischer Gradientenabstieg verwendet werden. Wir beschäftigen uns damit, ob sich die Lage für fixe Dimension verbessert, was uns zum Paradigma der parametrisierten Komplexität führt. Wir beweisen untere Laufzeitschranken in Form von W[1]-Härteresultaten, woraus folgt, dass bekannte Brute-Force-Strategien im Wesentlichen optimal sind. Auf der positiven Seite erweitern wir einen bekannten Polynomialzeitalgorithmus für konstante Dimension und konvexe Lossfunktionen auf eine allgemeinere Klasse von Lossfunktionen.

Die in dieser Arbeit verwendeten **mathematischen Methoden** umfassen Polyedertheorie, diskrete und tropische Geometrie, gemischt-ganzzahlige und kombinatorische Optimierung sowie Tools aus der Komplexitätstheorie.

Acknowledgments

During the past three years I have been a member of a wonderful environment consisting of the three groups *Combinatorial Optimization and Graph Algorithms*, *Discrete Optimization*, and *Discrete Mathematics* at TU Berlin. I particularly appreciated the relaxed and, at the same time, productive atmosphere including lots of social activities and I am grateful to everyone who contributed to this experience.

In addition, I have been part of the DFG Research Training Group GRK 2434 *Facets of Complexity*. I gratefully acknowledge the funding I received by this program. Furthermore, I really enjoyed the high level of exchange the program creates between different groups of FU, HU, and TU Berlin during the annual workshops and weekly Monday lectures.

A variety of people contributed to the success of this thesis. Particularly, I would like to thank ...

... my supervisor Martin Skutella for including me in his research group, for motivating me to dive into and stay with the topic of neural networks, for many inspiring research sessions and ideas, and for precious advice on various aspects of the academic world.

... Amitabh Basu for sharpening my understanding of ReLU neural networks, for planning to host me for a three-months research stay (which unfortunately never happened due to the pandemic), for always providing very supportive and motivating feedback, and for refereeing this thesis.

... Sebastian Pokutta for making it possible for me to attend the *Discrete Optimization and Machine Learning* conference in summer 2019 in Tokyo, for some helpful advice concerning publishing in the machine learning community, and for refereeing this thesis.

... Rolf Niedermeier for being an excellent BMS mentor, for making it possible to do my *Facets of Complexity* internship with his group, and for always spreading a cheerful mood.

... all my coauthors of papers included in this thesis. These are Amitabh Basu, Marco Di Summa, Vincent Froese, Rolf Niedermeier, Leon Sering, and Martin Skutella.

... Peter Bürgisser, Christian Haase, Michael Joswig, and Günter Rote for inspiring discussions, partially initiated at workshops by *Facets of Complexity*.

... Julia Amann, Tim Bergner, Niclas Böhmer, Sven Jäger, and Leonie Weißweiler for carefully proofreading (parts of) this thesis.

... Tim Bergner for an outstanding friendship on a personal and academic level, which started during our studies in Kaiserslautern and stayed active despite the distance.

… my flatmate Michael Rothgang for everything; let it be a sympathetic ear to all my problems, spontaneous nightly discussions, or managing daily life together.

… Frieder Smolny for emotional support in difficult times and a huge amount of valuable time spent together within and outside university.

… my brother Johannes Hertrich for bringing some sense of home to Berlin, sharing thoughts about neural networks and the academic world, and uncountably many silly moments.

… my parents Erika and Armin Hertrich for their unconditional support in all aspects of life.

Contents

1 Introduction

"Any sufficiently advanced technology is indistinguishable from magic."

Arthur C. Clarke's Third Law [Cla73]

1.1. Background

The idea that computers could behave intelligently, just like human beings, is almost as old as the idea of a computer itself. A famous early attempt to define how one could measure intelligence is Alan Turing's famous Turing test [Tur50], which he proposed in 1950. However, only in the last one or two decades the research field of *artificial intelligence* (AI) started to have a massive impact on our society. Today we can actually have conversations with our smartphones, computers can create "paintings" (see [GEB15]), and self-driving cars become reality.

While these innovations are undoubtedly based on many different factors, there is one particular subfield of AI that is responsible for a vast majority of modern breakthroughs: *deep learning*, that is, machine learning using *deep neural networks* [LBH15].

Also neural networks are not an invention of the 21st century, but are actually much older. The foundations were already laid in the middle of the 20th century by McCulloch and Pitts [MP43] and Rosenblatt [Ros58]. Around 1990, the famous LeNet [LeC+89] for recognizing handwritten digits already contained two key ingredients that are at the heart of many modern neural network architectures: a convolutional structure and training by the backpropagation algorithm. Although the basic principles have been present since then, it was not before the 2010s that neural networks became *the* major technology in many machine learning application domains. For example, the natural language processing community has completely been revolutionized in the past few years by neural networks [OMK20]. In the computer vision community, the work by Krizhevsky, Sutskever, and Hinton [KSH12] was the major game changer that established neural networks as the dominating technology. Interestingly, their network architecture is conceptually very similar to what has been used more than 20 years before by LeCun et al. [LeC+89]. However, a variety of nuances seemed to make the difference. First, increasing hardware power including massive parallelization on GPUs played a role. Second, clever tricks like data augmentation as well as a new regularization technique called drop-out have been important ingredients. Third, and most relevant to our work, the use of a different activation function seemed to play a crucial role in enhancing the performance: while previously the sigmoid function was used in most neural network models, rectified linear units (ReLUs) from now on became the standard choice; see also [GBB11].

It is worth noting that the vast majority of research about neural networks is of an empirical nature. Even though the basic building blocks of neural networks are very simple functions, it is hard to grasp what happens when these functions are combined into complicated network architectures. All major advances of the past few years have been driven by empirical evidence on practical tasks. As a result, theoretical explanations for the tremendous success of neural networks lag far behind the practical observations. Of course, there are attempts to close this gap in the literature. For example, there are approaches to explain concrete decisions of neural networks [Lap+19]. Also, from a more theoretical perspective, Vidal et al. [Vid+17] review mathematical justifications for beneficial properties of neural networks. Interesting recent developments are results around the so-called *neural tangent kernel* [JGH18], which could yield an explanation why highly overparameterized neural networks achieve good generalization results. However, the amount of open theoretical questions in the context of neural networks remains endless, which can be seen as the starting point for this thesis.

Mathematically, the change of paradigm concerning the activation functions mentioned above has interesting implications. The formerly used sigmoid function is perfectly smooth and, thus, the same holds for a whole neural network composed of sigmoid neurons or any part of it. In contrast, ReLUs are very simple continuous and piecewise linear (CPWL) functions. Consequently, ReLU neural networks naturally divide the input space into finitely many (polyhedral) pieces on which the prediction is linear, while being nondifferentiable at the breakpoints between those pieces. This enables us to analyze ReLU neural networks using discrete methods like polyhedral theory, combinatorics, or even tropical geometry.

1.2. Results and Overview of this Thesis

The goal of this thesis is to shed light on different aspects concerning the *complexity* of ReLU neural networks (NNs), using methods from discrete mathematics. In computer science, the term complexity usually refers to the amount of resources (time, memory, amount of data, etc.) required to achieve a certain goal (computing a function, solving a problem, learning a distribution, etc.). Naturally, many concrete questions can be asked in this broad context. In this thesis, we focus on three specific *facets* of complexity for ReLU NNs: expressivity, computational power, and training complexity. While our focus is on basic theoretical research about the model of ReLU NNs, a better understanding of this model will definitely have impact on practical aspects. For example, it might help to find appropriate network architectures for certain tasks, to explain decisions of NNs, or to improve existing training algorithms. In addition, we believe that our results are also interesting from a purely mathematical point of view. They yield new perspectives on the structure of polyhedral objects and piecewise linear functions, provide new insights about characteristics of combinatorial optimization problems, and build bridges between machine learning, discrete geometry, and (parameterized) complexity analysis.

After this introductory chapter, we provide fundamental definitions and results that are relevant for more than one of our facets in Chapter 2. The Chapters 3 to 5 contain an in-depth treatment of the three facets. Each of these chapters comes with its own introduction, literature review, and open problem section specific to it. It is possible to read the three chapters corresponding to the three facets independently from each other. We conclude with some final remarks in Chapter 6. In the following, we provide an overview of our results for each of our three facets.

First Facet: Expressivity

Which functions can be represented by certain architectures?

An important first step towards a better theoretical understanding of the practical success of NNs is to study their *expressivity.* This means identifying the class of functions that can be represented by NNs with a predefined size or architecture. While so-called *universal approximation theorems* [Hor91; Cyb89] show that one hidden layer is already sufficient to approximate any continuous function, surprisingly fundamental questions remain open in the context of exact representation.

It is known that a function is representable by a ReLU NN (of any size) if and only if it is continuous and piecewise linear (CPWL) [Aro+18]. However, in contrast to the case of approximations, the minimum number of hidden layers needed to represent a general CPWL function is unknown. The construction by Arora et al. [Aro+18] shows that logarithmically many layers (in the input dimension) are sufficient, but there is no function known that needs provably more than two hidden layers.

In Chapter 3, we conjecture that the logarithmic upper bound is indeed tight, that is, for every input dimension n there exists a function that indeed needs $\lceil \log_2(n+1) \rceil$ many hidden layers to be represented. We significantly simplify this conjecture by showing that it is equivalent to asking how many layers one needs to represent the function computing the maximum of n numbers. As a first step towards proving the conjecture, we use methods from polyhedral combinatorics and mixed-integer programming to provide a computational proof that the maximum of five numbers cannot be computed with two hidden layers. Unfortunately, our proof relies on an intuitive, but yet unproven assumption.

Even though our conjecture boils down to the required depth for the maximum function, as a second result, we show that NNs of a certain depth can compute strictly more functions than only those representable as linear combinations of certain maximum functions. The main tool for arriving at this result is the theory of polyhedral complexes.

As a third result, we provide upper bounds on the width required to represent a CPWL function with a given number of linear pieces by an NN with logarithmically many layers. We achieve these bounds by using a recently established correspondence between NNs and tropical geometry [ZNL18] that allows us to translate our question into discrete geometry and tackle it with classical methods from this field.

Finally, we give an outlook on how these relations to tropical geometry might also be beneficial for proving the conjecture itself via polytope theory.

All in all, we see the results of this chapter as an important first step towards shedding light on the expressivity of ReLU NNs by means of discrete and polyhedral geometry.

Second Facet: Computational Power

Which network size is sufficient to solve (combinatorial optimization) problems?

As the second facet we study the ability of NNs to solve computational problems, particularly from combinatorial optimization. To do so, we view NNs as a model of computation, similar to Boolean, or even closer, arithmetic circuits, operating on real numbers. We then study the complexity of various problems in this model. In this way, we obtain interesting insights regarding the interplay of NNs and classical algorithms and contribute towards explaining recent empirical success stories of applying NNs to combinatorial optimization problems in the literature.

NNs are a special type of arithmetic circuits that are only allowed to perform a very limited set of operations, namely a fixed number of affine transformations and maximum computations. In particular, due to the continuous nature of these operations, it is not possible to realize conditional branchings based on the comparison of real numbers. For this reason, it is a priori unclear whether polynomial-size NNs exist for a given problem, even if it admits strongly polynomial-time algorithms. Despite these difficulties, we constructively show that a variety of algorithmic questions can be precisely answered with NNs of bounded size.

For some problems, like computing the length of a shortest path in a network from individual arc lengths, this can be achieved by directly translating standard algorithms, for example, special dynamic programs, to NNs. For other problems, like the Minimum Spanning Tree Problem or the Maximum Flow Problem, all classical algorithms seem to make use of conditional branchings, making it impossible to implement them on NNs. New algorithmic ideas are required to obtain polynomial-size NNs for these problems. Using a result from arithmetic circuit complexity, we prove that the value of a minimum spanning tree in a graph with n nodes can be computed by an NN of size $\mathcal{O}(n^3)$. Moreover, we develop a completely new maximum flow algorithm without conditional branchings to show that, given a directed graph with n nodes and m arcs, there exists an NN with size $\mathcal{O}(m^2n^2)$ computing a maximum flow from any possible real-valued arc capacities as input.

Finally, we treat the NP-hard Knapsack Problem. We use a nontrivial modification of a well-known dynamic programming scheme to provide explicit pseudo-polynomial upper bounds on the size of NNs to compute exact solutions to the Knapsack Problem. Additionally, by incorporating a sophisticated rounding procedure directly in our NNs, we show that approximate solutions of provable quality can be achieved with significantly smaller NNs. In other words, a so-called *fully polynoimal time approximation scheme* (FPTAS) can be realized on NNs. In this way, we establish a rigorous trade-off between the size of NNs and their worst-case solution quality.

Summarizing, with this facet we demonstrate that, despite severe limitations with regard to the set of possible operations, ReLU NNs appear to be a surprisingly powerful model of real-valued computation.

Third Facet: Training Complexity

Does a small dimensionality help when fitting neural network weights to data?

A traditional use-case of NNs is *supervised learning.* This means learning to map input data to labels on the basis of the *training data*, that is, a set of given, labeled examples. In this context, the set of functions representable by a fixed NN architecture specifies the so-called *hypothesis class*, that is, the set of functions from which the prediction map should be chosen. For foundations of learning theory, we refer to the book by Shalev-Shwartz and Ben-David [SSBD14].

Once having specified a hypothesis class by fixing an NN architecture, the central question is the following: how can one select the "best" function in the hypothesis class, that is, the "best" NN weights, to predict the labels of unknown inputs?

One of the most popular learning paradigms, both from a theoretical and from a practical perspective, is *empirical risk minimization*. This means choosing a function from the hypothesis class that minimizes the error with respect to a so-called *loss function* on the training data. Applied to a fixed NN architecture, this paradigm results in the highly complex non-linear optimization problem of finding NN weights that minimize the training loss. In our third facet of NN complexity, we study the computational complexity of this training problem.

It is well-known that training an NN to minimize the empirical risk is NP-hard. Therefore, in practice, local search algorithms without optimality guarantees, like stochastic gradient descent, are usually applied. What we strive for in Chapter 5 is a more sophisticated understanding of the training complexity of NNs that goes beyond NP-hardness. More specifically, we investigate the question of whether the situation improves in case of a small input dimension. To this end, we employ methods from parameterized complexity.

In parameterized complexity, one usually distinguishes two types of algorithms to solve a parameterized problem. An algorithm is said to be *slice-wise polynomial* (XP) if its running time can be bounded by a polynomial for each fixed value of the parameter. However, since the degree of this polynomial might quickly increase with the parameter, one usually strives for the stronger notion of *fixed-parameter tractable* algorithms, where the degree of the polynomial must not depend on the parameter, but the running time itself can still exponentially increase with the parameter.

Our two main results of Chapter 5 can be summarized as follows. Even for the simplest possible network architecture with only one hidden neuron, fixed-parameter tractable algorithms with respect to parameter input dimension are impossible, unless widely believed complexity-theoretic conjectures fail. On the other hand, XP-algorithms exist for training arbitrary 2-layer NNs with a large variety of loss functions. Hence, we may conclude that a small dimension might indeed be helpful to reduce the training complexity, but only to a very limited extent.

2 Preliminaries

The goal of this chapter is to introduce the main object of study in this thesis: ReLU neural networks. We also review some knowledge from literature relevant to our work.

There are two different common ways to define (ReLU) neural networks, which are more or less equivalent: either as weighted directed acyclic graphs or as alternating composition of affine transformations and activation functions. We will introduce both ways in this chapter.

Since different chapters of this thesis build upon different existing literature that sometimes differ in notation, we have decided to slightly change our notation between chapters, in particular between Chapter 4 and Chapter 5. Whenever we deviate from previously introduced notation, there will always be a subsection explaining the chapter-specific notation in the introductory section of the respective chapter.

We write $[n] \coloneqq \{1, 2, \ldots, n\}$ for the set of natural numbers up to n (without zero) and $[n]_0 \coloneqq [n] \cup \{0\}$ for the same set including zero. For any $n \in \mathbb{N}$, let $\sigma\colon \mathbb{R}^n \to \mathbb{R}^n$ be the (component-wise) *rectifier* function

$$\sigma(x) = (\max\{0, x_1\}, \max\{0, x_2\}, \ldots, \max\{0, x_n\}).$$

2.1. Neural Networks as Directed Acyclic Graphs

The definitions and notations in this section are similar to Shalev-Shwartz and Ben-David [SSBD14, Chapter 20]. A *feedforward neural network with rectified linear units*, abbreviated by ReLU NN, or simply NN, is a finite, directed, acyclic, and weighted graph (V, E) with the following properties. The nodes V, which are also called *neurons*, are grouped into *layers* $V = V_0 \mathbin{\dot\cup} V_1 \mathbin{\dot\cup} \cdots \mathbin{\dot\cup} V_{k+1}$, $k \geq 0$, such that the layer index strictly increases along each arc. Sometimes, depending on the context, we additionally assume that the graph is *properly layered*, that is, arcs only occur between successive layers. The sets V_0 and V_{k+1}, whose elements are called *input neurons* and *output neurons*, are precisely the sets of neurons with in-degree and out-degree zero, respectively. Neurons in $V \setminus (V_0 \cup V_{k+1})$ are called *hidden neurons.* The corresponding node sets V_ℓ are called *input layer* ($\ell = 0$), *hidden layers* ($1 \leq \ell \leq k$), and *output layer* ($\ell = k+1$), respectively. We say the NN has $k + 1$ layers (usually not counting the input layer), or *depth* $k + 1$, that is, precisely the number of hidden layers plus one. Let $n_\ell = |V_\ell|$ be the number of neurons in the ℓ-th layer, called the *width* of that layer. The *width* and *size* of the NN are defined to be $\max\{n_1, \ldots, n_k\}$ and $\sum_{\ell=1}^{k} n_\ell$, respectively.

The graph is equipped with arc *weights* $w_{uv} \in \mathbb{R}$ for each $(u, v) \in E$ and node *biases* $b_v \in \mathbb{R}$ for each node $v \in V \setminus V_0$. Often, it is assumed that the biases at the output layer are zero. Every NN *computes* (we also say *represents*) a function $\mathbb{R}^{n_0} \to \mathbb{R}^{n_{k+1}}$ as follows. Given an input vector $x \in \mathbb{R}^{n_0}$, we associate an *activation* $a(v) \in \mathbb{R}$ with every neuron $v \in V \setminus V_0$ and an *output* $o(v) \in \mathbb{R}$ with every neuron $v \in V \setminus V_{k+1}$. First, the output values $o(v)$, $v \in V_0$, of the n_0 input neurons equal the n_0 components of the input vector x. Second, the activation of a neuron $v \in V \setminus V_0$ is the weighted sum of

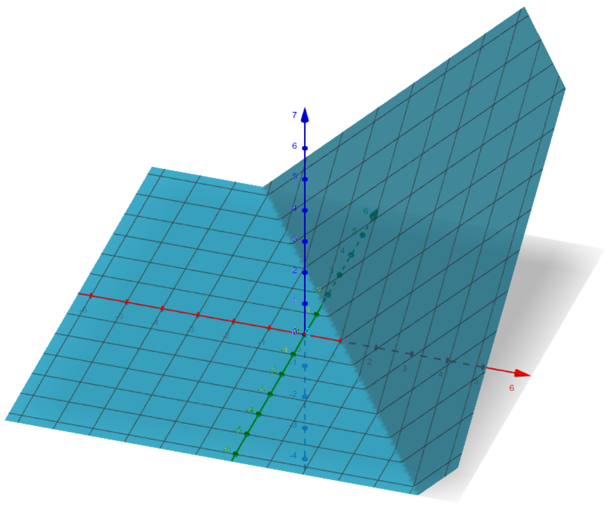


Figure 2.1.: Function computed by a ReLU neuron with two-dimensional input. The neuron first computes an affine transformation, having a hyperplane as graph. Afterwards, the rectifier function is applied, which sets all negative values to zero.

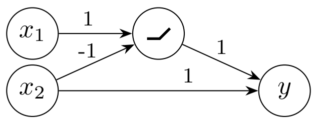


Figure 2.2.: An NN with two input neurons, labeled x_1 and x_2, one hidden neuron, labeled with the shape of the rectifier function, and one output neuron, labeled y. The arcs are labeled with their weights and all biases are zero. The network has depth 2, width 1, and size 1. It computes the function $x \mapsto y = x_2 + \max\{0, x_1 - x_2\} = \max\{x_1, x_2\}$.

outputs of all predecessors plus its bias, that is, $a(v) := b_v + \sum_{u\colon (u,v)\in E} w_{uv} o(u)$. Third, for each hidden neuron $v \in V \setminus (V_0 \cup V_{k+1})$, the output is determined by $o(v) := \sigma(a(v))$, where σ is the so-called *activation function*. In our case, σ is always the *rectifier function* $\sigma(z) = \max\{0, z\}$, as defined above. Consequently, neurons of these NNs are called *rectified linear units* (ReLUs). Finally, the output vector $y \in \mathbb{R}^{n_{k+1}}$ consists of the n_{k+1} activation values $a(v)$ of the n_{k+1} output neurons $v \in V_{k+1}$. Figure 2.1 illustrates the function computed by a ReLU neuron with two-dimensional input. Figure 2.2 gives an example showing a simple neural network computing the maximum of two numbers.

2.2. Neural Networks as Alternating Function Compositions

In this section, we consider properly layered NNs. Before we do so, let us explain why we can assume this without loss of generality for many purposes. As an example, consider the network depicted in Figure 2.2. The arc on the bottom points directly from an input node (in V_0) to an output node (in V_2), so it does not fulfill the required property.

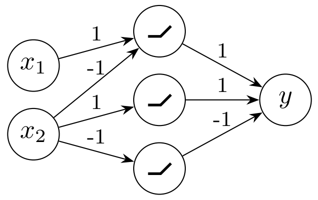


Figure 2.3.: Modified NN to compute the maximum of two numbers. Note that the two hidden neurons on the bottom and the arcs incident with them replace the direct connection from x_2 to y in Figure 2.2: If x_2 is positive, its value will be propagated via the hidden neuron in the middle; if it is negative, its value will be propagated via the hidden neuron at the bottom.

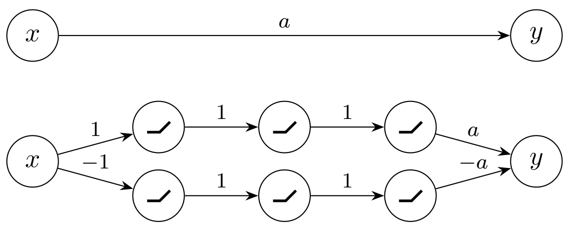


Figure 2.4.: Replacing an arc (with weight a) skipping three hidden layers (top) with a layered subnetwork (bottom). The value of x is split into positive and negative part and combined later on, weighted with weight a.

However, what we can do instead is to add two new neurons realizing an identity function in the hidden layer. This results in the NN depicted in Figure 2.3.

In general, any arc skipping z many layers can be replaced with two paths of length $z+1$ and $2z$ intermediate vertices, where one of the paths propagates the positive part and the other one the negative part of the input, see Figure 2.4. Doing this for every arc that skips at least one layer increases the total network size only by polynomial factors[1] and does not increase the depth of the NN. Hence, when interested in depth (precisely) or in size (up to polynomial factors), we may assume without loss of generality that the NN is properly layered.

For properly layered NNs, it makes sense to represent the weights and biases as affine transformations between layers as follows.

For a given $x \in \mathbb{R}^{n_0}$, let $x^{(\ell)} \in \mathbb{R}^{n_\ell}$ be the vector of outputs $o(v)$, $v \in V_\ell$, of the ℓ-th layer and let $y^{(\ell)} \in \mathbb{R}^{n_\ell}$ be the vector of activations $a(v)$, $v \in V_\ell$, of the ℓ-th layer. In particular, $x^{(0)}$ is the input to the NN and $y^{(k+1)}$ is the overall output of the NN.

With this notation, the weights and biases of all layers $\ell \in [k+1]$ define affine transformations $T^{(\ell)} \colon \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$, $x \mapsto A^{(\ell)}x + b^{(\ell)}$, such that $y^{(\ell)} = T^{(\ell)}(x^{(\ell-1)}) \in \mathbb{R}^{n_\ell}$. In particular, the entries of $A^{(\ell)}$ are the weights w_{uv} for arcs (u,v) between layers $V_{\ell-1}$ and V_ℓ (being zero if the arc is non-existent), and the entries of $b^{(\ell)}$ are the biases b_v for all $v \in V_\ell$. Then, the function *computed* (or *represented*) by the NN is the function $f \colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_{k+1}}$ given by $f = T^{(k+1)} \circ \sigma \circ T^{(k)} \circ \sigma \circ \cdots \circ T^{(2)} \circ \sigma \circ T^{(1)}$.

[1]We may assume without loss of generality that our NN does not have parallel arcs, that is, more than one arc with the same start and end vertex. Therefore, the total number of arcs is polynomially bounded in the number of vertices.

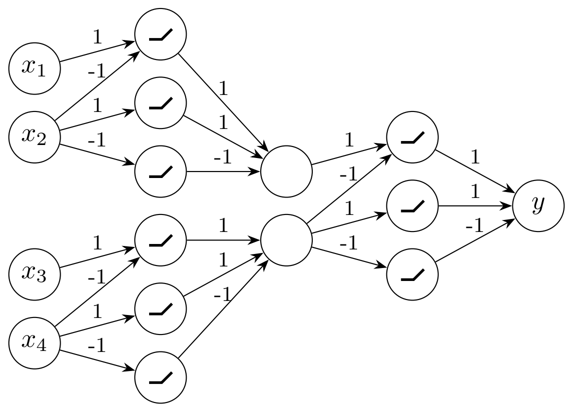


Figure 2.5.: An NN to compute the maximum of four numbers that consists of three copies of the NN in Figure 2.3. Note that no activiation function is applied at the two unlabeled middle vertices (representing $\max\{x_1, x_2\}$ and $\max\{x_3, x_4\}$). Therefore, the linear transformations directly before and after these vertices can be combined into a single one. Thus, the network has total depth three (two hidden layers).

2.3. ReLU Neural Networks and Piecewise Linear Functions

We state some results from the literature that will be useful throughout this thesis. The basic building block for many of our investigations is the following theorem by Arora, Basu, Mianjy, and Mukherjee [Aro+18]. By definition, a *continuous* function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ is *piecewise linear* (CPWL) in case there is a finite set of polyhedra whose union is $\mathbb{R}^n$, and f is affine linear over each such polyhedron; see, e.g., Schrijver [Sch86] for an introduction to the theory of polyhedra.

Theorem 2.1 (Arora et al. [Aro+18])**.** *A function $f\colon \mathbb{R}^n \to \mathbb{R}$ can be represented by a ReLU NN if and only if it is CPWL. In this case, depth $\lceil \log_2(n+1) \rceil + 1$ is sufficient.*

One direction of this statement is obvious from the definition of NNs: Since every operation involved in the computation of an NN is CPWL, the same must be true for the function computed by the full NN. The nontrivial direction is the converse: why can we represent any CPWL function defined on $\mathbb{R}^n$ with an NN of depth $\lceil \log_2(n+1) \rceil + 1$?

In the following, we would like to give an intuition for why this result holds true. For this purpose, we start with a simple special case of a CPWL function: the maximum of n numbers. Recall that one hidden layer suffices to compute the maximum of two numbers, see Figures 2.2 and 2.3. Now one can easily stack this operation: in order to compute the maximum of four numbers, we divide them into two pairs with two numbers each, compute the maximum of each pair and then the maximum of the two results. This idea results in the (properly layered) NN depicted in Figure 2.5, which has two hidden layers.

Repeating this procedure, one can compute the maximum of eight numbers with three hidden layers, and, in general, the maximum of 2^k numbers with k hidden layers. Additionally, by multiplying the weights in the first and last layer with -1, we can alternatively compute the minimum. Keeping track of size and width, we obtain the following proposition.

Proposition 2.2 (compare Lemma D.3 by Arora et al. [Aro+18]). *The maximum or minimum of n numbers can be computed with a ReLU NN of depth $\lceil \log_2(n) \rceil + 1$, width $\lceil 3n/2 \rceil$, and size $3(n-1)$.*

It is an open question whether the depth of this construction is best possible or whether the maximum of n numbers can be computed with less than $\lceil \log_2(n) \rceil$ hidden layers. We will study this question in detail in Chapter 3.

It is worth to note that the depth in Proposition 2.2 is almost equal to the general depth bound in Theorem 2.1, except for the additional plus one within the logarithm in the general bound. This is not a coincidence because the proof of Theorem 2.1 is based on Proposition 2.2, as well as the following (nontrivial) result about CPWL functions.

Theorem 2.3 (Wang and Sun [WS05]). *Every CPWL function $f\colon \mathbb{R}^n \to \mathbb{R}$ can be written in the form*

$$f(x) = \sum_{i=1}^{p} \lambda_i \max\{\ell_{i1}(x), \ldots, \ell_{i(n+1)}(x)\} \tag{2.1}$$

where $p \in \mathbb{N}$, $\lambda_1, \ldots, \lambda_p \in \mathbb{R}$, and $\ell_{ij}\colon \mathbb{R}^n \to \mathbb{R}$ is an affine linear function for every $i \in [p]$ and $j \in [n+1]$.

Note that maxima with less than $n+1$ terms are also allowed, as some functions ℓ_{ij} may coincide. In other words, Theorem 2.3 states that every CPWL function defined on $\mathbb{R}^n$ can be written as a linear combination of maxima of at most $n+1$ affine terms. The proof given by Wang and Sun [WS05] is technically involved and we will not go into details here. However, in Chapter 3 we will provide an alternative proof yielding a slightly stronger result. This will also be useful to bound the width of NNs representing arbitrary CPWL functions.

In fact, having Proposition 2.2 and Theorem 2.3 is already enough to prove Theorem 2.1: Having a CPWL function in the form of (2.1), one can obtain an NN representing it as follows. All the maxima of $n+1$ affine terms can be computed in parallel with $\lceil \log_2(n+1) \rceil$ many hidden layers. Apart from that, only affine transformations need to be realized, which does not increase the number of hidden layers. Therefore, Theorem 2.1 follows.

2.4. Recurrent Neural Networks

Since feedforward NNs have a fixed input size, a common way of handling sequential inputs of arbitrary length is to use *recurrent neural networks* (RNNs). This type of NNs has become very popular, e.g., for tasks in language or speech processing. Essentially, an RNN is a feedforward NN that is used repeatedly for every piece of the input sequence and maintains a hidden state by passing (part of) its output in each step as an additional input to the next step. More precisely, let $n, m, q \in \mathbb{N}$ be the dimensions of the input, output, and hidden state vector, respectively. Suppose we are given a sequence of $p \in \mathbb{N}$ input vectors $x_i \in \mathbb{R}^n$, $i \in [p]$, and a generically chosen (sometimes application-specific)

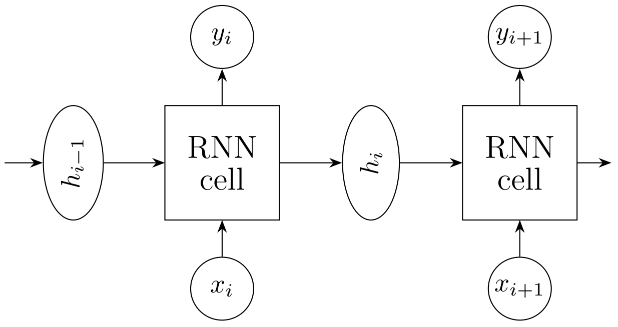


Figure 2.6.: Basic structure of an (unfolded) RNN.

initial hidden state vector $h_0 \in \mathbb{R}^q$. In the i-th step, the input of the RNN consists of the i-th input vector x_i, as well as the previous hidden state vector $h_{i-1} \in \mathbb{R}^q$. In the same manner as a feedforward NN described above, it then computes the i-th output vector $y_i \in \mathbb{R}^m$, as well as the new hidden state vector $h_i \in \mathbb{R}^q$. The crucial point is that the NN (called *RNN cell*) applied in each of these iterations is the same, with the same weights and biases (parameter sharing). That way, the p input vectors x_i can be mapped to p output vectors y_i for arbitrary values of p using a fixed-size NN. The basic structure of an RNN is illustrated in Figure 2.6. Sometimes it holds that $y_i = h_i$, that is, the i-th output is actually equal to the i-th hidden state.

3 Expressivity

Which functions can be represented by certain architectures?

The results in this chapter are based on a paper with Amitabh Basu, Marco Di Summa, and Martin Skutella [Her+21] that appeared in the proceedings of the NeurIPS 2021 conference.

3.1. Introduction

A core problem in machine learning and statistics is the estimation of an unknown data distribution with access to independent and identically distributed samples from the distribution. It is well-known that there is a tension between how much prior information one has about the data distribution and how many samples one needs to solve the problem with high confidence (or equivalently, how much variance one has in one's estimate). This is referred to as the *bias-variance* trade-off or the *bias-complexity* trade-off. Neural networks provide a way to turn this bias-complexity knob in a controlled manner. This is done by modifying the *architecture* of a neural network class of functions, in particular their *size* in terms of *depth* and *width.* As one increases these parameters, the class of functions becomes more expressive. In terms of the bias-variance trade-off, the "bias" decreases as the class of functions becomes more expressive, but the "variance" or "complexity" increases.

So-called *universal approximation theorems* [Cyb89; Hor91; AB99] show that even with a single hidden layer, that is, when the depth of the architecture is the smallest possible value, one can essentially reduce the "bias" as much as one desires, by increasing the width. Nevertheless, it can be advantageous both theoretically and empirically to increase the depth because a substantial reduction in the size can be achieved by this [Aro+18; ES16; LS17; SS17; Tel15; Tel16; Yar17]. To get a better quantitative handle on these trade-offs, it is important to understand what classes of functions are exactly representable by neural networks with a certain architecture. The precise mathematical statements of universal approximation theorems show that single layer networks can arbitrarily well *approximate* any continuous function (under some additional mild hypotheses). While this suggests that single layer networks are good enough from a learning perspective, from a mathematical perspective, one can ask the question if the class of functions represented by a single layer is a *strict* subset of the class of functions represented by two or more hidden layers. On the question of size, one can ask for precise bounds on the width of the network of a given depth to represent a certain class of functions. We believe that a better understanding of the function classes exactly represented by different architectures will have implications not just for mathematical foundations, but also algorithmic and statistical learning aspects of neural networks. The task of searching for the "best" function in that class can only benefit from a better understanding of the nature of functions in that class. A motivating question behind the results in this chapter is to understand the hierarchy of function classes exactly represented by neural networks of increasing depth.

3.1.1. Classes of CPWL Functions

For the investigations in this chapter, we introduce the following classes of CPWL functions for each $k \in \mathbb{N}$:

$$\begin{aligned} \mathrm{ReLU}_n(k) &\coloneqq \{f\colon \mathbb{R}^n \to \mathbb{R} \mid f \text{ can be represented by a } (k+1)\text{-layer NN}\}, \\ \mathrm{CPWL}_n &\coloneqq \{f\colon \mathbb{R}^n \to \mathbb{R} \mid f \text{ is continuous and piecewise linear}\}. \end{aligned}$$

In order to analyze $\mathrm{ReLU}_n(k)$, we use another function class defined as follows. For $p \in \mathbb{N}$, we call a function g a *p-term max* function if it can be expressed as maximum of p affine terms, that is, $g(x) = \max\{\ell_1(x), \dots, \ell_p(x)\}$ where $\ell_i\colon \mathbb{R}^n \to \mathbb{R}$ is affinely linear for $i \in [p]$. Based on that, we define

$$\mathrm{MAX}_n(p) \coloneqq \{f\colon \mathbb{R}^n \to \mathbb{R} \mid f \text{ is a linear combination of } p\text{-term max functions}\}.$$

If the input dimension n is not important for the context, we might drop the index and use $\mathrm{ReLU}(k) \coloneqq \bigcup_{n\in\mathbb{N}} \mathrm{ReLU}_n(k)$ and $\mathrm{MAX}(p) \coloneqq \bigcup_{n\in\mathbb{N}} \mathrm{MAX}_n(p)$ instead.

In this chapter, we deal with polyhedra a lot. We use the standard notations $\operatorname{conv} A$ and $\operatorname{cone} A$ for the convex and conic hulls of a set $A \subseteq \mathbb{R}^n$, respectively. We also need the notion of the *Minkowski sum* of two polyhedra P and Q: it is given as the set $P + Q = \{p + q \mid p \in P, q \in Q\}$. For an in-depth treatment of polyhedra and (mixed-integer) optimization, we refer to Schrijver [Sch86].

3.1.2. Our Main Conjecture

Recall that, by Theorem 2.1, every CPWL function defined on $\mathbb{R}^n$ can be represented by a ReLU neural network with $\lceil \log_2(n+1) \rceil$ hidden layers. We wish to understand whether one can do better. We believe it is not possible to do better and we pose the following conjecture to better understand the importance of depth in neural networks.

Conjecture 3.1. *For any $n \in \mathbb{N}$, let $k^* \coloneqq \lceil \log_2(n+1) \rceil$. Then it holds that*

$$\mathrm{ReLU}_n(0) \subsetneq \mathrm{ReLU}_n(1) \subsetneq \cdots \subsetneq \mathrm{ReLU}_n(k^*-1) \subsetneq \mathrm{ReLU}_n(k^*) = \mathrm{CPWL}_n\,. \tag{3.1}$$

Conjecture 3.1 claims that any additional layer up to k^* hidden layers strictly increases the set of representable functions. This would imply that the construction of Theorem 2.1 is actually depth-minimal.

Observe that in order to prove Conjecture 3.1, it is sufficient to find a single function $f \in \mathrm{CPWL}_n \setminus \mathrm{ReLU}_n(k^*-1)$ with $n = 2^{k^*-1}$ for all $k^* \in \mathbb{N}$. This also implies that all remaining strict inclusions $\mathrm{ReLU}_n(i-1) \subsetneq \mathrm{ReLU}_n(i)$ for $i < k^*$ are valid since $\mathrm{ReLU}_n(i-1) = \mathrm{ReLU}_n(i)$ directly implies that $\mathrm{ReLU}_n(i-1) = \mathrm{ReLU}_n(i')$ for all $i' \geq i-1$.

In fact, there is a canonical candidate for such a function, allowing us to reformulate the conjecture as follows.

Conjecture 3.2. *For any $k \in \mathbb{N}$, $n = 2^k$, the function $f_n(x) = \max\{0, x_1, \dots, x_n\}$ cannot be represented with k hidden layers, that is, $f_n \notin \mathrm{ReLU}_n(k)$.*

Proposition 3.3. *Conjecture 3.1 and Conjecture 3.2 are equivalent.*

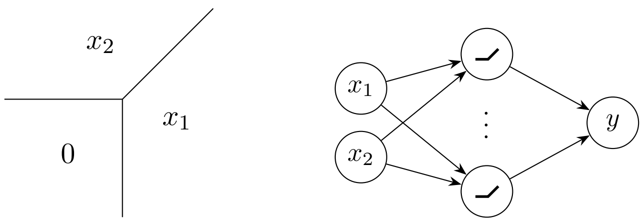


Figure 3.1.: Set of breakpoints of the function $\max\{0, x_1, x_2\}$ (left). This function cannot be computed by a 2-layer NN (right), since the set of breakpoints of any function computed by such an NN is a union of lines.

Proof. We argued above that Conjecture 3.2 implies Conjecture 3.1. For the other direction, we prove the contraposition, that is, assuming that Conjecture 3.2 is violated, we show that Conjecture 3.1 is violated, as well. To this end, suppose there is a $k \in \mathbb{N}$, $n = 2^k$, such that f_n is representable with k hidden layers. We argue that, under this hypothesis, any $(n+1)$-term max function can be represented with k hidden layers. To see this, observe that

$$\max\{\ell_1(x), \ldots, \ell_{n+1}(x)\} = \max\{0, \ell_1(x) - \ell_{n+1}(x), \ldots, \ell_n(x) - \ell_{n+1}(x)\} + \ell_{n+1}(x).$$

Modifying the first-layer weights of the NN computing f_n such that input x_i is replaced by the affine expression $\ell_i(x) - \ell_{n+1}(x)$, one obtains a k-hidden-layer NN computing the function $\max\{0, \ell_1(x) - \ell_{n+1}(x), \ldots, \ell_n(x) - \ell_{n+1}(x)\}$. Moreover, since affine functions, in particular also $\ell_{n+1}(x)$, can easily be represented by k-hidden-layer NNs, we obtain that any $(n+1)$-term maximum is in $\mathrm{ReLU}_n(k)$. Using Theorem 2.3, it follows that $\mathrm{ReLU}_n(k) = \mathrm{CPWL}_n$. In particular, since $k^* := \lceil \log_2(n+1) \rceil = k+1$, we obtain that Conjecture 3.1 must be violated as well. □

It is known that Conjecture 3.2 holds for $k = 1$ [MB17], that is, the CPWL function $\max\{0, x_1, x_2\}$ cannot be computed by a 2-layer NN. The reason for this is that the set of breakpoints of a CPWL function computed by a 2-layer NN is always a union of lines, while the set of breakpoints of $\max\{0, x_1, x_2\}$ is a union of three half-lines; compare Figure 3.1 and the detailed proof by Mukherjee and Basu [MB17].

However, the conjecture remains open for all $k \geq 2$.

3.1.3. Overview and Main Results

In this chapter, we present the following results as partial progress towards resolving our conjecture.

In Section 3.2, we resolve Conjecture 3.2 for $k = 2$, under a natural assumption on the breakpoints of the function represented by any intermediate neuron. We achieve this result by leveraging techniques from mixed-integer programming to analyze the set of functions computable by certain NNs.

By Proposition 2.2 it follows that $\mathrm{MAX}(2^k) \subseteq \mathrm{ReLU}(k)$ for all $k \in \mathbb{N}$, that is, any 2^k-term max function (and linear combinations thereof) can be expressed with k hidden layers. One might ask whether the converse is true as well, that is, whether the classes

$\mathrm{MAX}(2^k)$ and $\mathrm{ReLU}(k)$ are actually equal. This would not only provide a neat characterization of $\mathrm{ReLU}(k)$, but also prove Conjecture 3.2 without any additional assumption since one can show that $\max\{0, x_1, \ldots, x_{2^k}\}$ is not contained in $\mathrm{MAX}(2^k)$.

In fact, this is true for $k = 1$, that is, a function is computable with one hidden layer if and only if it is a linear combination of 2-term max functions. However, in Section 3.3, we show that for $k \geq 2$, the class $\mathrm{ReLU}(k)$ is a strict superset of $\mathrm{MAX}(2^k)$. To achieve this result, the key technical ingredient is the theory of polyhedral complexes associated with CPWL functions. This way, we provide important insights concerning the richness of the class $\mathrm{ReLU}(k)$.

So far, we have focused on understanding the smallest depth needed to express CPWL functions using neural networks with ReLU activations. In Section 3.4, we complement these results by upper bounds on the sizes of the networks needed for expressing arbitrary CPWL functions. In particular, Theorem 3.30 shows that any CPWL function with p affine pieces on $\mathbb{R}^n$ can be expressed by a network with depth at most $\mathcal{O}(\log n)$ and width at most $p^{\mathcal{O}(n^2)}$. We arrive at this result by introducing a novel application of recently established interactions between neural networks and tropical geometry.

Finally, in Section 3.5, we provide an outlook how these interactions between tropical geometry and NNs could possibly also be useful to provide a full, unconditional proof of Conjecture 3.1 by means of polytope theory. In Section 3.6, we point out further open research questions.

3.1.4. Related Work

Depth versus size. Soon after the original universal approximation theorems [Cyb89; Hor91], concrete bounds were obtained on the number of neurons needed in the hidden layer to achieve a certain level of accuracy. The literature on this is vast and we refer to a small representative sample here [Bar93; Bar94; Pin99; Mha96; MM95]. More recently, work has focused on how deeper networks can have exponentially or super exponentially smaller size compared to shallower networks [Tel16; ES16; Aro+18; Var+21]. See also Gribonval et al. [Gri+21] for another perspective on the relationship between expressivity and architecture, and the references therein. We reiterate that the list of references above is far from complete.

Mixed-integer optimization and machine learning. Over the past decade, a growing body of work has emerged that explores the interplay between mixed-integer optimization and machine learning. On the one hand, researchers have attempted to improve mixed-integer optimization algorithms by exploiting novel techniques from machine learning [BLZ18; Gas+19; HDE14; Kha+16; Kha+17b; KLP17; LZ17; ALW17]; see also Bengio, Lodi, and Prouvost [BLP21] for a recent survey. On the flip side, mixed-integer optimization techniques have been used to analyze function classes represented by neural networks [STR18; And+20; FJ17; SR20; SKR20]. In Section 3.2 below, we show another new use of mixed-integer optimization tools for understanding function classes represented by neural networks.

Design of training algorithms. We believe that a better understanding of the function classes represented exactly by a neural architecture also has benefits in terms of understanding the complexity of the training problem. For instance, in a paper by Arora et al. [Aro+18], an understanding of single layer ReLU networks enables the design of a globally optimal algorithm for solving the empirical risk minimization (ERM) problem that runs in polynomial time in the number of data points in fixed dimension. There are many works going in a similar direction [Goe+17; GKM18; GK19; DWX20; BDL20; Goe+21]; compare also Chapter 5 of this thesis.

Neural Networks and Tropical Geometry. A recent stream of research involves the interplay between neural networks and tropical geometry. The piecewise linear functions computed by neural networks can be seen as (tropical quotients of) tropical polynomials. Linear regions of these functions correspond to vertices of so-called *Newton polytopes* associated with these tropical polynomials. Applications of this correspondence include bounding the number of linear regions of a neural network [ZNL18; CM18; MRZ21] and understanding decision boundaries [Alf+20]. In Section 3.4 we present a novel application of tropical concepts to understand neural networks. We refer to Maragos, Charisopoulos, and Theodosis [MCT21] for a recent survey of connections between machine learning and tropical geometry, as well as to the textbooks by Maclagan and Sturmfels [MS15] and Joswig [Jos21] for in-depth introductions to tropical geometry and tropical combinatorics.

3.2. Conditional Lower Bounds on Depth via Mixed-Integer Programming

In this section, we provide a computer-aided proof that, under a natural, yet unproven assumption, the function $f(x) := \max\{0, x_1, x_2, x_3, x_4\}$ cannot be represented by a 3-layer NN. It is worth to note that, to the best of our knowledge, no CPWL function is known for which the non-existence of a 3-layer NN can be proven without additional assumption. For easier notation, we write $x_0 := 0$.

We first prove that we may restrict ourselves to NNs without biases. This holds true independent of our assumption, which we introduce afterwards.

Definition 3.4. A function $g\colon \mathbb{R}^n \to \mathbb{R}^m$ is called *positively homogeneous* if it satisfies $g(\lambda x) = \lambda g(x)$ for all $\lambda \geq 0$.

Definition 3.5. For an NN given by affine transformations $T^{(\ell)}(x) = A^{(\ell)}x + b^{(\ell)}$, we define the corresponding *homogenized NN* to be the NN given by $\tilde{T}^{(\ell)}(x) = A^{(\ell)}x$ with all biases set to zero.

Proposition 3.6. *If an NN computes a positively homogeneous function, then the corresponding homogenized NN computes the same function.*

Proof. Let $g\colon \mathbb{R}^{n_0} \to \mathbb{R}^{n_{k+1}}$ be the function computed by the original NN and $\tilde{g}$ the one computed by the homogenized NN. Further, for any $0 \leq \ell \leq k$, let

$$g^{(\ell)} = T^{(\ell+1)} \circ \sigma \circ T^{(\ell)} \circ \cdots \circ T^{(2)} \circ \sigma \circ T^{(1)}$$

be the function computed by the sub-NN consisting of the first $(\ell+1)$-layers and let $\tilde{g}^{(\ell)}$ be the function computed by the corresponding homogenized sub-NN. We first show by induction on ℓ that the norm of $\|g^{(\ell)}(x) - \tilde{g}^{(\ell)}(x)\|$ is bounded by a global constant that only depends on the parameters of the NN but not on x.

For $\ell = 0$, we obviously have $\|g^{(0)}(x) - \tilde{g}^{(0)}(x)\| = \|b^{(1)}\| =: C_0$, settling the induction base. For the induction step, let $\ell \geq 1$ and assume that $\|g^{(\ell-1)}(x) - \tilde{g}^{(\ell-1)}(x)\| \leq C_{\ell-1}$, where $C_{\ell-1}$ only depends on the parameters of the NN. Since component-wise ReLU activation has Lipschitz constant 1, this implies $\|(\sigma \circ g^{(\ell-1)})(x) - (\sigma \circ \tilde{g}^{(\ell-1)})(x)\| \leq C_{\ell-1}$. Using any matrix norm that is compatible with the Euclidean vector norm, we obtain:

$$\begin{aligned} \|g^{(\ell)}(x) - \tilde{g}^{(\ell)}(x)\| &= \|b^{(\ell+1)} + A^{(\ell+1)}((\sigma \circ g^{(\ell-1)})(x) - (\sigma \circ \tilde{g}^{(\ell-1)})(x))\| \\ &\leq \|b^{(\ell+1)}\| + \|A^{(\ell+1)}\| \cdot C_{\ell-1} \ =: \ C_\ell \end{aligned}$$

Since the right-hand side only depends on NN parameters, the induction is completed.

Finally, we show that $g = \tilde{g}$. For the sake of contradiction, suppose that there is an $x \in \mathbb{R}^{n_0}$ with $\|g(x) - \tilde{g}(x)\| = \delta > 0$. Let $x' := \frac{C_k+1}{\delta} x$; then, by positive homogeneity, it follows that $\|g(x') - \tilde{g}(x')\| = C_k + 1 > C_k$, contradicting the property shown above. Thus, we have $g = \tilde{g}$. □

Since $f = \max\{0, x_1, x_2, x_3, x_4\}$ is positively homogeneous, Proposition 3.6 implies that, if there is a 3-layer NN computing f, then there also is one that has no biases. Therefore, in the remainder of this section, we only consider NNs without biases and assume implicitly that all considered CPWL functions are positively homogeneous. In particular, any piece of such a CPWL function is linear and not only affine linear.

Observe that, for the function f, the only points of non-differentiability (a.k.a. *breakpoints*) are at places where at least two of the five numbers $x_0 = 0$, x_1, x_2, x_3, and x_4 are equal. Hence, if some neuron of an NN computing f introduces breakpoints at other places, these breakpoints must be canceled out by other neurons. Therefore, it is a natural assumption that such breakpoints are not introduced at all in the first place.

To make this assumption formal, let $H_{ij} = \{x \in \mathbb{R}^4 \mid x_i = x_j\}$, for $0 \leq i < j \leq 4$, be ten hyperplanes in $\mathbb{R}^4$ and $H = \bigcup_{0 \leq i < j \leq 4} H_{ij}$ be the corresponding hyperplane arrangement. The *regions* or *cells* of H are defined to be the closures of the connected components of $\mathbb{R}^4 \setminus H$. It is easy to see that these regions are in one-to-one correspondence to the $5! = 120$ possible orderings of the five numbers $x_0 = 0$, x_1, x_2, x_3, and x_4. More precisely, for a permutation π of the five indices $[4]_0 = \{0, 1, 2, 3, 4\}$, the corresponding region is the polyhedron

$$C_\pi \ := \ \{x \in \mathbb{R}^4 \mid x_{\pi(0)} \leq x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)} \leq x_{\pi(4)}\}.$$

We say that a (positively homogeneous) CPWL function g is *H-conforming*, if it is linear within any of these regions of H, that is, if it only has breakpoints where the relative ordering of the five values $x_0 = 0$, x_1, x_2, x_3, x_4 changes; see Figure 3.2 for an illustration of the (simpler) two-dimensional case. Moreover, an NN is said to be H-conforming if the output of each neuron contained in the NN is H-conforming. Equivalently, this is the case if and only if all intermediate functions $\sigma \circ T^{(\ell)} \circ \sigma \circ T^{(\ell-1)} \circ \cdots \circ \sigma \circ T^{(1)}$, $\ell \in [k]$, are H-conforming. Now our assumption can be formally phrased as follows.

Assumption 3.7. *If there exists a 3-layer NN computing $f(x) = \max\{0, x_1, x_2, x_3, x_4\}$, then there also exists one that is H-conforming.*

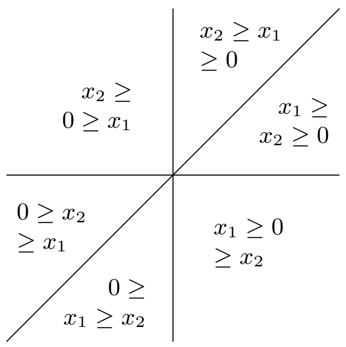


Figure 3.2.: A function is H-conforming if the set of breakpoints is a subset of the hyperplane arrangement H. The arrangement H consists of all hyperplanes where two of the coordinates (possibly including $x_0 = 0$) are equal. Here, H is illustrated for the (simpler) two-dimensional case, where it consists of three hyperplanes that divide the space into six cells.

We use mixed-integer programming to prove the following theorem.

Theorem 3.8. *Under Assumption 3.7, there does not exist a 3-layer NN that computes the function* $f(x) = \max\{0, x_1, x_2, x_3, x_4\}$.

The remainder of this section is devoted to proving this theorem. The rough outline of the proof is as follows. We first study some geometric properties of the hyperplane arrangement H. This will show that each of the 120 cells of H is a simplicial polyhedral cone spanned by 4 extreme rays. In total, there are 30 such rays (because rays are used multiple times to span different cones). This implies that each H-conforming function is uniquely determined by its values on the 30 rays and, therefore, the set of H-conforming functions of type $\mathbb{R}^4 \to \mathbb{R}$ is a 30-dimensional vector space. We then use linear algebra to show that the space of functions generated by H-conforming two-layer NNs is a 14-dimensional subspace. Moreover, with two hidden layers, at least 29 of the 30 dimensions can be generated and f is not contained in this 29-dimensional subspace. So the remaining question is whether the 14 dimensions producible with the first hidden layer can be combined in such a way that after applying a ReLU activation in the second hidden layer, we do not end up within the 29-dimensional subspace. We model this question as a mixed-integer program (MIP). Solving the MIP yields that we always end up within the 29-dimensional subspace, implying that f cannot be represented by a 3-layer NN. This provides a computational proof of Theorem 3.8.

Let us start with investigating the structure of the hyperplane arrangement H. For readers familiar with the interplay between hyperplane arrangements and polytopes, it is worth noting that H is dual to a combinatorial equivalent of the 4-dimensional permutahedron. Hence, what we are studying in the following are some combinatorial properties of the permutahedron.

Recall that the regions of H are given by the 120 polyhedra

$$C_\pi := \{x \in \mathbb{R}^4 \mid x_{\pi(0)} \leq x_{\pi(1)} \leq x_{\pi(2)} \leq x_{\pi(3)} \leq x_{\pi(4)}\}$$

for each permutation π of $[4]_0$. With this representation, one can see that C_π is a pointed polyhedral cone (with the origin as its only vertex) spanned by the four half-lines (a.k.a. *rays*)

$$\begin{aligned}
R_{\{\pi(0)\}} &:= \{x \in \mathbb{R}^4 \mid x_{\pi(0)} \le x_{\pi(1)} = x_{\pi(2)} = x_{\pi(3)} = x_{\pi(4)}\},\\
R_{\{\pi(0),\pi(1)\}} &:= \{x \in \mathbb{R}^4 \mid x_{\pi(0)} = x_{\pi(1)} \le x_{\pi(2)} = x_{\pi(3)} = x_{\pi(4)}\},\\
R_{\{\pi(0),\pi(1),\pi(2)\}} &:= \{x \in \mathbb{R}^4 \mid x_{\pi(0)} = x_{\pi(1)} = x_{\pi(2)} \le x_{\pi(3)} = x_{\pi(4)}\},\\
R_{\{\pi(0),\pi(1),\pi(2),\pi(3)\}} &:= \{x \in \mathbb{R}^4 \mid x_{\pi(0)} = x_{\pi(1)} = x_{\pi(2)} = x_{\pi(3)} \le x_{\pi(4)}\}.
\end{aligned}$$

With that notation, we see that each of the 120 cells of H is a simplicial cone spanned by four out of the 30 rays R_S with $\emptyset \subsetneq S \subsetneq [4]_0$. For each such set S, denote its complement by $\bar{S} := [4]_0 \setminus S$. Let us use a generating vector $r_S \in \mathbb{R}^4$ for each of these rays such that $R_S = \operatorname{cone} r_S$ as follows: If $0 \in S$, then $r_S := \mathbb{1}_{\bar{S}} \in \mathbb{R}^4$, otherwise $r_S := -\mathbb{1}_S \in \mathbb{R}^4$, where for each $S \subseteq [4]$, the vector $\mathbb{1}_S \in \mathbb{R}^4$ contains entries 1 at precisely those index positions that are contained in S and entries 0 elsewhere. For example, $r_{\{0,2,3\}} = (1,0,0,1) \in \mathbb{R}^4$ and $r_{\{1,4\}} = (-1,0,0,-1) \in \mathbb{R}^4$. Then, the set R containing conic generators of all the 30 rays of H consists of the 30 vectors $R = (\{0,1\}^4 \cup \{0,-1\}^4) \setminus \{0\}^4$.

Let $\mathcal{S}^{30}$ be the space of all H-conforming CPWL functions of type $\mathbb{R}^4 \to \mathbb{R}$. We show that $\mathcal{S}^{30}$ is a 30-dimensional vector space.

Lemma 3.9. *The map $g \mapsto (g(r))_{r\in R}$ that evaluates a function $g \in \mathcal{S}^{30}$ at the 30 rays in R is an isomorphism between $\mathcal{S}^{30}$ and $\mathbb{R}^{30}$. In particular, $\mathcal{S}^{30}$ is a 30-dimensional vector space.*

Proof. First note that $\mathcal{S}^{30}$ is closed under addition and scalar multiplication. Therefore, it is a subspace of the vector space of continuous functions of type $\mathbb{R}^4 \to \mathbb{R}$, and thus, in particular, a vector space. We show that the map $g \mapsto (g(r))_{r\in R}$ is in fact a vector space isomorphism. The map is obviously linear, so we only need to show that it is a bijection. In order to do so, remember that $\mathbb{R}^4$ is the union of the $5! = 120$ simplicial cones C_π. In particular, given the function values on the extreme rays of these cones, there is a unique positively homogeneous, continuous continuation that is linear within each of the 120 cones. This implies that the considered map is a bijection between $\mathcal{S}^{30}$ and $\mathbb{R}^{30}$. □

The previous lemma also provides a canonical basis of the vector space $\mathcal{S}^{30}$: the one consisting of all CPWL functions attaining value 1 at one ray $r \in R$ and value 0 at all other rays. However, it turns out that for our purposes it is more convenient to work with a different basis. For this purpose, let $g_M(x) = \max_{i\in M} x_i$ for each $\{\emptyset, \{0\}\} \not\ni M \subseteq [4]_0$. These 30 functions contain, among other functions, the four (linear) coordinate projections $g_{\{i\}}(x) = x_i$, $i \in [4]$, and the function $f(x) = g_{[4]_0}(x) = \max\{0, x_1, x_2, x_3, x_4\}$.

Lemma 3.10. *The 30 functions $g_M(x) = \max_{i\in M} x_i$ with $\{\emptyset, \{0\}\} \not\ni M \subseteq [4]_0$ form a basis of $\mathcal{S}^{30}$.*

Proof. Evaluating the 30 functions g_M at all 30 rays $r \in R$ yields 30 vectors in $\mathbb{R}^{30}$. It can be easily verified (e.g., using a computer) that these vectors form a basis of $\mathbb{R}^{30}$. Thus, due to the isomorphism of Lemma 3.9, the functions g_M form a basis of $\mathcal{S}^{30}$. □

Next, we focus on particular subspaces of $\mathcal{S}^{30}$ generated by only some of the 30 functions g_M. We prove that they correspond to the spaces of functions computable by H-conforming 2- and 3-layer NNs, respectively.

To this end, let $\mathcal{B}^{14}$ be the set of the 14 basis functions g_M with $\{\emptyset, \{0\}\} \not\ni M \subseteq [4]_0$ and $|M| \leq 2$. Let $\mathcal{S}^{14}$ be the 14-dimensional subspace spanned by $\mathcal{B}^{14}$. Similarly, let $\mathcal{B}^{29}$ be the set of the 29 basis functions g_M with $\{\emptyset, \{0\}\} \not\ni M \subsetneq [4]_0$ (all but $[4]_0$). Let $\mathcal{S}^{29}$ be the 29-dimensional subspace spanned by $\mathcal{B}^{29}$.

Lemma 3.11. *The space $\mathcal{S}^{14}$ consists of all functions computable by H-conforming 2-layer NNs.*

Proof. Each function in $\mathcal{B}^{14}$ is a maximum of at most 2 numbers and can thus be represented by a 2-layer NN by Proposition 2.2. By putting the corresponding networks in parallel and adding appropriate weights to the connections to the output, also all linear combinations of these 14 functions, and thus, the full space $\mathcal{S}^{14}$, can be represented by a 2-layer NN.

Conversely, we show that any function representable by a 2-layer NN is indeed contained in $\mathcal{S}^{14}$. It suffices to show that the output of every neuron in the first (and only) hidden layer of an H-conforming ReLU NN is in $\mathcal{S}^{14}$ because the output of a 2-layer NN is a linear combination of such outputs. Let $a \in \mathbb{R}^4$ be the first-layer weights of such a neuron, computing the function $g_a(x) := \max\{a^T x, 0\}$, which has the hyperplane $\{x \in \mathbb{R}^4 \mid a^T x = 0\}$ as breakpoints (or is constantly zero). Since the NN must be H-conforming, this must be one of the ten hyperplanes $x_i = x_j$, $0 \leq i < j \leq 4$. Thus, $g_a(x) = \max\{\lambda(x_i - x_j), 0\}$ for some $\lambda \in \mathbb{R}$. If $\lambda \geq 0$, it follows that $g_a = \lambda g_{\{i,j\}} - \lambda g_{\{j\}} \in \mathcal{S}^{14}$, and if $\lambda \leq 0$, we obtain $g_a = -\lambda g_{\{i,j\}} + \lambda g_{\{i\}} \in \mathcal{S}^{14}$. This concludes the proof. $\square$

For 3-layer NNs, an analogous statement can be made. However, only one direction can be easily seen.

Lemma 3.12. *Any function in $\mathcal{S}^{29}$ can be represented by an H-conforming 3-layer NN.*

Proof. Each function in $\mathcal{B}^{29}$ is a maximum of at most 4 numbers and can thus be represented by a 3-layer NN by Proposition 2.2. As in the previous lemma, also linear combinations of those can be represented. $\square$

Our goal is to prove the converse as well: any H-conforming function represented by a 3-layer NN is in $\mathcal{S}^{29}$. Since $f(x) = \max\{0, x_1, x_2, x_3, x_4\}$ is the 30th basis function, which is linearly independent from $\mathcal{B}^{29}$ and thus not contained in $\mathcal{S}^{29}$, this implies Theorem 3.8. To achieve this goal, we first provide another characterization of $\mathcal{S}^{29}$, which can be seen as an orthogonal direction to $\mathcal{S}^{29}$ in $\mathcal{S}^{30}$. For a function $g \in \mathcal{S}^{30}$, let

$$\phi(g) := \sum_{\emptyset \subsetneq S \subsetneq [4]_0} (-1)^{|S|} g(r_S)$$

be a linear map from $\mathcal{S}^{30}$ to $\mathbb{R}$.

Lemma 3.13. *A function $g \in \mathcal{S}^{30}$ is contained in $\mathcal{S}^{29}$ if and only if $\phi(g) = 0$.*

Proof. Any $g \in \mathcal{S}^{30}$ can be represented as a unique linear combination of the 30 basis functions g_M and is contained in $\mathcal{S}^{29}$ if and only if the coefficient of $f = g_{[4]_0}$ is zero. One can easily check (with a computer) that ϕ maps all functions in $\mathcal{B}^{29}$ to 0, but not the 30th basis function f. Thus, g is contained in $\mathcal{S}^{29}$ if and only if it satisfies $\phi(g) = 0$. $\square$

In order to make use of our Assumption 3.7, we need the following insight about when the property of being H-conforming is preserved after applying a ReLU activation.

Lemma 3.14. *Let $g \in \mathcal{S}^{30}$. The function $h = \sigma \circ g$ is H-conforming (and thus in $\mathcal{S}^{30}$ as well) if and only if there is no pair of sets $\emptyset \subsetneq S \subsetneq S' \subsetneq [4]_0$ with $g(r_S)$ and $g(r_{S'})$ being nonzero and having different signs.*

Proof. The key observation to prove this lemma is the following: for two rays r_S and $r_{S'}$, there exists a cell C of the hyperplane arrangement H for which both r_S and $r_{S'}$ are extreme rays if and only if $S \subsetneq S'$ or $S' \subsetneq S$.

Hence, if there exists a pair of sets $\emptyset \subsetneq S \subsetneq S' \subsetneq [4]_0$ with $g(r_S)$ and $g(r_{S'})$ being nonzero and having different signs, then the function g restricted to C is a linear function with both strictly positive and strictly negative values. Therefore, after applying the ReLU activation, the resulting function h has breakpoints within C and is not H-conforming.

Conversely, if for each pair of sets $\emptyset \subsetneq S \subsetneq S' \subsetneq [4]_0$, both $g(r_S)$ and $g(r_{S'})$ are either nonpositive or nonnegative, then g restricted to any cell C of H is either nonpositive or nonnegative everywhere. In the first case, h restricted to that cell C is the zero function, while in the second case, h coincides with g in C. In both cases, h is linear within all cells and, thus, H-conforming. $\square$

Having collected all these lemmas, we are finally able to construct a MIP whose solution proves that any function computed by an H-conforming 3-layer NN is in $\mathcal{S}^{29}$. As in the proof of Lemma 3.11, it suffices to focus on the output of a single neuron in the second hidden layer. Let $h = \sigma \circ g$ be the output of such a neuron with g being its input. Observe that, by construction, g is a function computed by a 2-layer NN, and thus, by Lemma 3.11, a linear combination of the 14 functions in $\mathcal{B}^{14}$. The MIP contains three types of variables, which we denote in bold to distinguish them from constants:

- 14 continuous variables $\mathbf{a}_M \in [-1, 1]$, being the coefficients of the linear combination of the basis of $\mathcal{S}^{14}$ forming g, that is, $g = \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M$ (since multiplying g and h with a nonzero scalar does not alter the containment of h in $\mathcal{S}^{29}$, we may assume unit interval bounds),
- 30 binary variables $\mathbf{z}_S \in \{0, 1\}$ for $\emptyset \subsetneq S \subsetneq [4]_0$, determining whether the considered neuron is strictly active at ray r_S, that is, whether $g(r_S) > 0$,
- 30 continuous variables $\mathbf{y}_S \in \mathbb{R}$ for $\emptyset \subsetneq S \subsetneq [4]_0$, representing the output of the considered neuron at all rays, that is, $\mathbf{y}_S = h(r_S)$.

To ensure that these variables interact as expected, we need two types of constraints:

- For each of the 30 rays r_S, $\emptyset \subsetneq S \subsetneq [4]_0$, the following constraints ensure that $\mathbf{z}_S$ and output $\mathbf{y}_S$ are correctly calculated from the variables $\mathbf{a}_M$, that is, $\mathbf{z}_S = 1$

if and only if $g(r_S) = \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M(r_S)$ is positive, and $\mathbf{y}_S = \max\{0, g(r_S)\}$. Also compare the references given in Section 3.1.4 concerning MIP models for ReLU units. Note that the restriction of the coefficients $\mathbf{a}_M$ to $[-1, 1]$ ensures that the absolute value of $g(r_S)$ is always bounded by 14, allowing us to use 15 as a replacement for $+\infty$:

$$\begin{aligned} \mathbf{y}_S &\geq 0 \\ \mathbf{y}_S &\geq \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M(r_S) \\ \mathbf{y}_S &\leq 15\mathbf{z}_S \\ \mathbf{y}_S &\leq \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M(r_S) + 15(1 - \mathbf{z}_S) \end{aligned} \tag{3.2}$$

Observe that these constraints ensure that one of the following two cases occurs: If $\mathbf{z}_S = 0$, then the first and third line imply $\mathbf{y}_S = 0$ and the second line implies that the incoming activation is in fact nonpositive. The fourth line is always satisfied in that case. Otherwise, if $\mathbf{z}_S = 1$, then the second and fourth line imply that $\mathbf{y}_S$ equals the incoming activation, and, in combination with the first line, this has to be nonnegative. The third line is always satisfied in that case. Hence, the set of constraints (3.2) correctly models the ReLU activation function.

- For each of the 150 pairs of sets $\emptyset \subsetneq S \subsetneq S' \subsetneq [4]_0$, the following constraints ensure that the property in Lemma 3.14 is satisfied. More precisely, if one of the variables $\mathbf{z}_S$ or $\mathbf{z}_{S'}$ equals 1, then the ray of the other set has nonnegative activation, that is, $g(r_{S'}) \geq 0$ or $g(r_S) \geq 0$, respectively:

$$\begin{aligned} \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M(r_S) &\geq 15(\mathbf{z}_{S'} - 1) \\ \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M(r_{S'}) &\geq 15(\mathbf{z}_S - 1) \end{aligned} \tag{3.3}$$

  Observe that these constraints successfully prevent that the two rays r_S and $r_{S'}$ have nonzero activations with different signs. Conversely, if this is not the case, then we can always satisfy constraints (3.3) by setting only those variables $\mathbf{z}_S$ to value 1 where the activation of ray r_S is *strictly* positive. (Note that, if the incoming activation is precisely zero, constraints (3.2) make it possible to choose both values 0 or 1 for $\mathbf{z}_S$.) Hence, these constraints are in fact appropriate to model H-conformity.

In the light of Lemma 3.13, the objective function of our MIP is to maximize $\phi(h)$, that is, the expression

$$\sum_{\emptyset \subsetneq S \subsetneq [4]_0} (-1)^{|S|} \mathbf{y}_S.$$

The MIP has a total of 30 binary and 44 continuous variables, as well as 420 inequality constraints. The next proposition formalizes how this MIP can be used to check whether a 3-layer NN function can exist outside $\mathcal{S}^{29}$.

Proposition 3.15. *There exists an H-conforming 3-layer NN computing a function not contained in $\mathcal{S}^{29}$ if and only if the objective value of the MIP defined above is strictly positive.*

Proof. For the first direction, assume that such an NN exists. Since its final output is a linear combination of the outputs of the neurons in the second hidden layer, one of these neurons must compute a function $\tilde{h} = \sigma \circ \tilde{g} \notin \mathcal{S}^{29}$, with $\tilde{g}$ being the input to that neuron. By Lemma 3.13, it follows that $\phi(\tilde{h}) \neq 0$. Moreover, we can even assume without loss of generality that $\phi(\tilde{h}) > 0$, as we argue now. If this is not the case, multiply all first-layer weights of the NN by -1 to obtain a new NN computing function $\hat{h}$ instead of $\tilde{h}$. Observing that $r_S = -r_{[4]_0 \setminus S}$ for all $r_S \in R$, we obtain $\hat{h}(r_S) = \tilde{h}(-r_S) = \tilde{h}(r_{[4]_0 \setminus S})$ for all $r_S \in R$. Plugging this into the definition of ϕ and using that the cardinalities of S and $[4]_0 \setminus S$ have different parity, we further obtain $\phi(\hat{h}) = -\phi(\tilde{h})$. Therefore, we can assume that $\phi(\tilde{h})$ was already positive in the first place.

Using Lemma 3.11, $\tilde{g}$ can be represented as a linear combination $\tilde{g} = \sum_{g_M \in \mathcal{B}^{14}} \tilde{\mathbf{a}}_M g_M$ of the functions in $\mathcal{B}^{14}$. Let $\alpha \coloneqq \max_M |\tilde{\mathbf{a}}_M| > 0$. Let us define modified functions g and h from $\tilde{g}$ and $\tilde{h}$ as follows. Let $\mathbf{a}_M \coloneqq \tilde{\mathbf{a}}_M/\alpha \in [-1,1]$, $g \coloneqq \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M$, and $h \coloneqq \sigma \circ g$. Moreover, for all rays $r_S \in R$, let $\mathbf{y}_S \coloneqq h(r_S)$, as well as $\mathbf{z}_S \coloneqq 1$ if $\mathbf{y}_S > 0$, and $\mathbf{z}_S \coloneqq 0$ otherwise.

It is easy to verify that the variables $\mathbf{a}_M$, $\mathbf{y}_S$, and $\mathbf{z}_S$ defined that way satisfy (3.2). Moreover, since the NN is H-conforming, they also satisfy (3.3). Finally, they also yield a strictly positive objective function value since $\phi(h) = \phi(\tilde{h})/\alpha > 0$.

For the reverse direction, assume that there exists a MIP solution consisting of $\mathbf{a}_M$, $\mathbf{y}_S$, and $\mathbf{z}_S$, satisfying (3.2) and (3.3), and having a strictly positive objective function value. Define the functions $g \coloneqq \sum_{g_M \in \mathcal{B}^{14}} \mathbf{a}_M g_M$ and $h \coloneqq \sigma \circ g$. One concludes from (3.2) that $h(r_S) = \mathbf{y}_S$ for all rays $r_S \in R$. Lemma 3.11 implies that g can be represented by a 2-layer NN. Thus, h can be represented by a 3-layer NN. Moreover, constraints (3.3) guarantee that this NN is H-conforming. Finally, since the MIP solution has strictly positive objective function value, we obtain $\phi(h) > 0$, implying that $h \notin \mathcal{S}^{29}$. $\square$

In order to use the MIP as part of a mathematical proof, we employed a MIP solver that uses exact rational arithmetics without numerical errors, namely the solver by the Parma Polyhedral Library (PPL) [BHZ08]. We called the solver from a SageMath (Version 9.0) [Sag20] script on a machine with an Intel Core i7-8700 6-Core 64-bit CPU and 15.5 GB RAM, using the openSUSE Leap 15.2 Linux distribution. SageMath, which natively includes the PPL solver, is published under the GPLv3 license. After a total running time of almost 7 days (153 hours), we obtained optimal objective function value zero. This makes it possible to prove Theorem 3.8.

Proof of Theorem 3.8. Since the MIP has optimal objective function value zero, Proposition 3.15 implies that any function computed by an H-conforming 3-layer NN is contained in $\mathcal{S}^{29}$. In particular, under Assumption 3.7, it is not possible to compute the function $f(x) = \max\{0, x_1, x_2, x_3, x_4\}$ with a 3-layer NN. $\square$

We remark that state-of-the-art MIP solver Gurobi (version 9.1.1) [Gur21], which is commercial but offers free academic licenses, is able to solve the same MIP within less than a second, providing the same result. However, Gurobi does not employ exact arithmetics, making it impossible to exclude numerical errors and use it as a mathematical proof.

The SageMath code can be found on GitHub at

`https://github.com/ChristophHertrich/relu-mip-depth-bound`.

Additionally, the MIP can be found there as `.mps` file, a standard format to represent MIPs. This allows one to use any solver of choice to reproduce our result.

3.3. Going Beyond Linear Combinations of Max Functions

In this section we prove the following result, showing that NNs with k hidden layers can compute more functions than only linear combinations of 2^k-term max functions.

Theorem 3.16. *For any $k \geq 2$, the class* $\mathrm{ReLU}(k)$ *is a strict superset of* $\mathrm{MAX}(2^k)$.

To prove this theorem, we provide a specific function that is in $\mathrm{ReLU}(k) \setminus \mathrm{MAX}(2^k)$ for any number of hidden layers $k \geq 2$. The challenging part is to show that the function is in fact not contained in $\mathrm{MAX}(2^k)$.

Proposition 3.17. *For any $n \geq 3$, the function*

$$f\colon \mathbb{R}^n \to \mathbb{R}, \quad f(x) = \max\{0, x_1, x_2, \dots, x_{n-3}, \max\{x_{n-2}, x_{n-1}\} + \max\{0, x_n\}\} \tag{3.4}$$

cannot be written as a linear combination of n-term max functions.

The above proposition means that it is not possible to write $f(x)$ in the form

$$f(x) = \sum_{i=1}^{p} \lambda_i \max\{\ell_{i1}(x), \dots, \ell_{in}(x)\}$$

where $p \in \mathbb{N}$, $\lambda_1, \dots, \lambda_p \in \mathbb{R}$, and $\ell_{ij} : \mathbb{R}^n \to \mathbb{R}$ is an affine linear function for every $i \in [p]$ and $j \in [n]$. (Note that max functions with less than n terms are also allowed, as some functions ℓ_{ij} may coincide.)

Before we prove Proposition 3.17, we show that it implies Theorem 3.16.

Proof of Theorem 3.16. For $k \geq 2$, let $n := 2^k$. By Proposition 3.17, function f defined in (3.4) is not contained in $\mathrm{MAX}(2^k)$. It remains to show that it can be represented using a ReLU NN with k hidden layers. To see this, first observe that any of the $n/2 = 2^{k-1}$ terms $\max\{0, x_1\}$, $\max\{x_{2i}, x_{2i+1}\}$ for $i \in [n/2 - 2]$, and $\max\{x_{n-2}, x_{n-1}\} + \max\{0, x_n\}$ can be expressed by a one-hidden-layer NN since all these are (linear combinations of) 2-term max functions. Since f is the maximum of these 2^{k-1} terms, and since the maximum of 2^{k-1} numbers can be computed with $k-1$ hidden layers (Proposition 2.2), this implies that f is in $\mathrm{ReLU}(k)$. $\square$

In order to prove Proposition 3.17, we need the concept of polyhedral complexes. A *polyhedral complex* $\mathcal{P}$ is a finite set of polyhedra such that each face of a polyhedron in $\mathcal{P}$ is also in $\mathcal{P}$, and for two polyhedra $P, Q \in \mathcal{P}$, their intersection $P \cap Q$ is a common face of P and Q (possibly the empty face). Given a polyhedral complex $\mathcal{P}$ in $\mathbb{R}^n$ and an integer $m \in [n]$, we let $\mathcal{P}^m$ denote the collection of all m-dimensional polyhedra in $\mathcal{P}$.

For a convex CPWL function f, we define its *underlying polyhedral complex* as follows: it is the unique polyhedral complex covering $\mathbb{R}^n$ (i.e., each point in $\mathbb{R}^n$ belongs to some polyhedron in $\mathcal{P}$) whose n-dimensional polyhedra coincide with the domains of the (maximal) affine pieces of f. In particular, f is affinely linear within each $P \in \mathcal{P}$, but not within any strict superset of a polyhedron in $\mathcal{P}^n$.

Exploiting properties of polyhedral complexes associated with CPWL functions, we prove the following proposition below.

Proposition 3.18. *Let $f_0 \colon \mathbb{R}^n \to \mathbb{R}$ be a convex CPWL function and let $\mathcal{P}_0$ be the underlying polyhedral complex. If there exists a hyperplane $H \subseteq \mathbb{R}^n$ such that the set*

$$T \coloneqq \bigcup \left\{ F \in \mathcal{P}_0^{n-1} \;\middle|\; F \subseteq H \right\}$$

is nonempty and contains no line, then f_0 cannot be expressed as a linear combination of n-term maxima of affine linear functions.

Again, before we proceed to the proof of Proposition 3.18, we show that it implies Proposition 3.17.

Proof of Proposition 3.17. Observe that f (defined in (3.4)) has the alternate representation

$$f(x) = \max\{0,\, x_1,\, x_2,\, \ldots,\, x_{n-3},\, x_{n-2},\, x_{n-1},\, x_{n-2} + x_n,\, x_{n-1} + x_n\}$$

as a maximum of $n + 2$ terms. Let $\mathcal{P}$ be its underlying polyhedral complex. Let the hyperplane H be defined by $x_1 = 0$.

Observe that any facet in $\mathcal{P}^{n-1}$ is a polyhedron defined by two of the $n+2$ terms that are equal and at least as large as each of the remaining n terms. Hence, the only facet that could possibly be contained in H is

$$F \coloneqq \{x \in \mathbb{R}^n \mid x_1 = 0 \geq x_2,\, \ldots,\, x_{n-3},\, x_{n-2},\, x_{n-1},\, x_{n-2} + x_n,\, x_{n-1} + x_n\}.$$

Note that F is indeed an $(n-1)$-dimensional facet in $\mathcal{P}^{n-1}$, because, for example, the full neighborhood of $(0, -1, \ldots, -1) \in \mathbb{R}^n$ intersected with H is contained in F.

Finally, we need to show that F is pointed, that is, it contains no line. A well-known fact from polyhedral theory says if there is any line in F with direction $d \in \mathbb{R}^n \setminus \{0\}$, then d must satisfy the defining inequalities with equality. However, only the zero vector does this. Hence, F cannot contain a line.

Therefore, when applying Proposition 3.18 to f with underlying polyhedral complex $\mathcal{P}$ and hyperplane H, we have $T = F$, which is nonempty and contains no line. Hence, f cannot be written as linear combination of n-term maxima. □

The remainder of this section is devoted to proving Proposition 3.18. The proof presented here is based on private communication with Amitabh Basu and Marco Di Summa. In order to exploit properties of the underlying polyhedral complex of the considered CPWL functions, we will first introduce some terminology, notation, and results related to polyhedral complexes in $\mathbb{R}^n$ for any $n \geq 1$.

Definition 3.19. Given an abelian group $(G,+)$, we define $\mathcal{F}^n(G)$ as the family of all functions ϕ of type $\phi\colon \mathcal{P}^n \to G$, where $\mathcal{P}$ is a polyhedral complex that covers $\mathbb{R}^n$. We say that $\mathcal{P}$ is the *underlying* polyhedral complex, or the polyhedral complex *associated* with ϕ.

Just to give an intuition of the reason for this definition, let us mention that later we will choose $(G,+)$ to be the set of affine linear maps $\mathbb{R}^n \to \mathbb{R}$ with respect to the standard operation of sum of functions. Moreover, given a convex CPWL function $f\colon \mathbb{R}^n \to \mathbb{R}$ with underlying polyhedral complex $\mathcal{P}$, we will consider the following function $\phi \in \mathcal{F}^n(G)$: for every $P \in \mathcal{P}^n$, $\phi(P)$ will be the affine linear map that coincides with f over P. It can be helpful, though not necessary, to keep this in mind when reading the next definitions and observations.

It is useful to observe that the functions in $\mathcal{F}^n(G)$ can also be described in a different way. Before explaining this, we need to define an ordering between the two elements of each pair of opposite halfspaces. More precisely, let H be a hyperplane in $\mathbb{R}^n$ and let H', H'' be the two closed halfspaces delimited by H. We choose an arbitrary rule to say that H' "precedes" H'', which we write as $H' \prec H''$.[1] We can then extend this ordering rule to those pairs of n-dimensional polyhedra of a polyhedral complex in $\mathbb{R}^n$ that share a facet. Specifically, given a polyhedral complex $\mathcal{P}$ in $\mathbb{R}^n$, let $P', P'' \in \mathcal{P}^n$ be such that $F := P' \cap P'' \in \mathcal{P}^{n-1}$. Further, let H be the unique hyperplane containing F. We say that $P' \prec P''$ if the halfspace delimited by H and containing P' precedes the halfspace delimited by H and containing P''.

We can now explain the alternate description of the functions in $\mathcal{F}^n(G)$, which is based on the following notion.

Definition 3.20. Let $\phi \in \mathcal{F}^n(G)$, with associated polyhedral complex $\mathcal{P}$. The *facet-function* associated with ϕ is the function $\psi\colon \mathcal{P}^{n-1} \to G$ defined as follows: given $F \in \mathcal{P}^{n-1}$, let P', P'' be the two polyhedra in $\mathcal{P}^n$ such that $F = P' \cap P''$, where $P' \prec P''$; then we set $\psi(F) := \phi(P') - \phi(P'')$.

Although it will not be used, we observe that knowing ψ is sufficient to reconstruct ϕ up to an additive constant. This means that a function $\phi' \in \mathcal{F}^n(G)$ associated with the same polyhedral complex $\mathcal{P}$ has the same facet-function ψ if and only if there exists $g \in G$ such that $\phi(P) - \phi'(P) = g$ for every $P \in \mathcal{P}^n$. (However, it is not true that every function $\psi\colon \mathcal{P}^{n-1} \to G$ is the facet-function of some function in $\mathcal{F}^n(G)$.)

We now introduce a sum operation over $\mathcal{F}^n(G)$.

Definition 3.21. Given p functions $\phi_1, \dots, \phi_p \in \mathcal{F}^n(G)$, with associated polyhedral complexes $\mathcal{P}_1, \dots, \mathcal{P}_p$, the sum $\phi := \phi_1 + \cdots + \phi_p$ is the function in $\mathcal{F}^n(G)$ defined as follows:

- the polyhedral complex associated with ϕ is

$$\mathcal{P} := \{P_1 \cap \cdots \cap P_p \mid P_i \in \mathcal{P}_i \text{ for every } i\};$$

[1] In case one wants to see such a rule explicitly, this is a possible way: Fix an arbitrary $\bar{x} \in H$. We can say that $H' \prec H''$ if and only if $\bar{x} + e_i \in H'$, where e_i is the first vector in the standard basis of $\mathbb{R}^d$ that does not lie on H (i.e., $e_1, \dots, e_{i-1} \in H$ and $e_i \notin H$). Note that this definition does not depend on the choice of $\bar{x}$.

- given $P \in \mathcal{P}^n$, P can be uniquely obtained as $P_1 \cap \cdots \cap P_p$, where $P_i \in \mathcal{P}_i^n$ for every i; we then define
$$\phi(P) = \sum_{i=1}^{p} \phi_i(P_i).$$

The term "sum" is justified by the fact that when $\mathcal{P}_1 = \cdots = \mathcal{P}_p$ (and thus $\phi_1, \dots, \phi_p$ have the same domain) we obtain the standard notion of the sum of functions.

The next results shows how to compute the facet-function of a sum of functions in $\mathcal{F}^n(G)$.

Observation 3.22. *With the notation of Definition 3.21, let $\psi_1, \dots, \psi_p$ be the facet-functions associated with $\phi_1, \dots, \phi_p$, and let ψ be the facet-function associated with ϕ. Given $F \in \mathcal{P}^{n-1}$, let I be the set of indices $i \in \{1, \dots, p\}$ such that $\mathcal{P}_i^{n-1}$ contains a (unique) element F_i with $F \subseteq F_i$. Then*
$$\psi(F) = \sum_{i \in I} \psi_i(F_i). \tag{3.5}$$

Proof. Let P', P'' be the two polyhedra in $\mathcal{P}^n$ such that $F = P' \cap P''$, with $P' \prec P''$. We have $P' = P_1' \cap \cdots \cap P_p'$ and $P'' = P_1'' \cap \cdots \cap P_p''$ for a unique choice of $P_i', P_i'' \in \mathcal{P}_i^n$ for every i. Then
$$\psi(F) = \phi(P') - \phi(P'') = \sum_{i=1}^{p} (\phi_i(P_i') - \phi_i(P_i'')). \tag{3.6}$$
Now fix $i \in [p]$. Since $F \subseteq P_i' \cap P_i''$, $\dim(P_i' \cap P_i'') \geq n-1$. If $\dim(P_i' \cap P_i'') = n-1$, then $F_i := P_i' \cap P_i'' \in \mathcal{P}_i^{n-1}$ and $\phi_i(P_i') - \phi_i(P_i'') = \psi_i(F_i)$. Furthermore, $i \in I$ because $F \subseteq F_i$. If, on the contrary, $\dim(P_i' \cap P_i'') = n$, the fact that $\mathcal{P}_i$ is a polyhedral complex implies that $P_i' = P_i''$, and thus $\phi_i(P_i') - \phi_i(P_i'') = 0$. Moreover, in this case $i \notin I$: this is because $P' \cup P'' \subseteq P_i'$, which implies that the relative interior of F is contained in the relative interior of P_i'. With these observations, from (3.6) we obtain (3.5). $\square$

Definition 3.23. Fix $\phi \in \mathcal{F}^n(G)$, with associated polyhedral complex $\mathcal{P}$. Let H be a hyperplane in $\mathbb{R}^n$, and let H', H'' be the closed halfspaces delimited by H. Define the polyhedral complex
$$\widehat{\mathcal{P}} = \{P \cap H \mid P \in \mathcal{P}\} \cup \{P \cap H' \mid P \in \mathcal{P}\} \cup \{P \cap H'' \mid P \in \mathcal{P}\}.$$
The refinement of ϕ with respect to H is the function $\widehat{\phi} \in \mathcal{F}^n(G)$ with associated polyhedral complex $\widehat{\mathcal{P}}$ defined as follows: given $\widehat{P} \in \widehat{\mathcal{P}}^n$, $\widehat{\phi}(\widehat{P}) := \phi(P)$, where P is the unique polyhedron in $\mathcal{P}$ that contains $\widehat{P}$.

The next results shows how to compute the facet-function of a refinement.

Observation 3.24. *With the notation of Definition 3.23, let ψ be the facet-function associated with ϕ. Then, the facet-function $\widehat{\psi}$ associated with $\widehat{\phi}$ is given by*
$$\widehat{\psi}(\widehat{F}) = \begin{cases} \psi(F) & \text{if there exists a (unique) } F \in \mathcal{P}^{n-1} \text{ containing } \widehat{F} \\ 0 & \text{otherwise,} \end{cases}$$
for every $\widehat{F} \in \widehat{\mathcal{P}}^{n-1}$.

Proof. Let $\widehat{P}', \widehat{P}''$ be the polyhedra in $\widehat{\mathcal{P}}^n$ such that $\widehat{F} = \widehat{P}' \cap \widehat{P}''$, with $\widehat{P}' \prec \widehat{P}''$. Further, let P', P'' be the unique polyhedra in $\mathcal{P}^n$ that contain $\widehat{P}', \widehat{P}''$ (respectively); note that $P' \prec P''$.

If there exists $F \in \mathcal{P}^{n-1}$ containing $\widehat{F}$, then the fact that $\mathcal{P}$ is a polyhedral complex implies that $F = P' \cap P''$. Thus $\widehat{\psi}(\widehat{F}) = \widehat{\phi}(\widehat{P}') - \widehat{\phi}(\widehat{P}'') = \phi(P') - \phi(P'') = \psi(F)$.

Assume now that no element of $\mathcal{P}^{n-1}$ contains $\widehat{F}$. Then there exists $P \in \mathcal{P}^n$ such that $\widehat{F} = P \cap H$ and H intersects the interior of P. Then $\widehat{P}' = P \cap H'$ and $\widehat{P}'' = P \cap H''$ (or vice versa). It follows that $\widehat{\psi}(\widehat{F}) = \widehat{\phi}(\widehat{P}') - \widehat{\phi}(\widehat{P}'') = \phi(P) - \phi(P) = 0$. □

We now prove that the operations of sum and refinement commute: the refinement of a sum is the sum of the refinements.

Observation 3.25. *Let p functions $\phi_1, \ldots, \phi_p \in \mathcal{F}^n(G)$, with associated polyhedral complexes $\mathcal{P}_1, \ldots, \mathcal{P}_p$, be given. Define $\phi := \phi_1 + \cdots + \phi_p$. Let H be a hyperplane in $\mathbb{R}^n$, and let H', H'' be the closed halfspaces delimited by H. Then $\widehat{\phi} = \widehat{\phi}_1 + \cdots + \widehat{\phi}_p$.*

Proof. Define $\widetilde{\phi} := \widehat{\phi}_1 + \cdots + \widehat{\phi}_p$. It can be verified that $\widehat{\phi}$ and $\widetilde{\phi}$ are defined on the same poyhedral complex, which we denote by $\widehat{\mathcal{P}}$. We now fix $\widehat{P} \in \widehat{\mathcal{P}}^n$ and show that $\widehat{\phi}(\widehat{P}) = \widetilde{\phi}(\widehat{P})$.

Since $\widehat{P} \in \widehat{\mathcal{P}}^n$, we have $\widehat{P} = P_1 \cap \cdots \cap P_p \cap H'$, where $P_i \in \mathcal{P}_i^n$ for every i. (We ignore the case $\widehat{P} = P_1 \cap \cdots \cap P_p \cap H''$, which is identical.) Then

$$\widehat{\phi}(\widehat{P}) = \phi(P_1 \cap \cdots \cap P_p) = \sum_{i=1}^{p} \phi_i(P_i) = \sum_{i=1}^{p} \widehat{\phi}_i(P_i \cap H') = \widetilde{\phi}(P_1 \cap \cdots \cap P_p \cap H') = \widetilde{\phi}(P),$$

where the first and third equations follow from the definition of refinement, while the second and fourth equations follow from the definition of the sum. □

The *lineality space* of a (nonempty) polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is the null space of the constraint matrix A. In other words, it is the set of vectors $y \in \mathbb{R}^n$ such that for every $x \in P$ the whole line $\{x + \lambda y \mid \lambda \in \mathbb{R}\}$ is a subset of P. We say that the lineality space of P is *trivial*, if it contains only the zero vector, and *nontrivial* otherwise.

Since, given a polyhedral complex $\mathcal{P}$ that covers $\mathbb{R}^n$, all the nonempty polyhedra in $\mathcal{P}$ share the same lineality space L, we will call L the lineality space of $\mathcal{P}$.

Lemma 3.26. *Given an abelian group $(G, +)$, pick $\phi_1, \ldots, \phi_p \in \mathcal{F}^n(G)$, with associated polyhedral complexes $\mathcal{P}_1, \ldots, \mathcal{P}_p$. Assume that for every $i \in [p]$ the lineality space of $\mathcal{P}_i$ is nontrivial. Define $\phi := \phi_1 + \cdots + \phi_p$, $\mathcal{P}$ as the underlying polyhedral complex, and ψ as the facet-function of ϕ. Then for every hyperplane $H \subseteq \mathbb{R}^n$, the set*

$$S := \bigcup \left\{ F \in \mathcal{P}^{n-1} \mid F \subseteq H,\, \psi(F) \neq 0 \right\}$$

is either empty or contains a line.

Proof. The proof is by induction on n. For $n = 1$, the assumptions imply that all $\mathcal{P}_i$ are equal to $\mathcal{P}$, and each of these polyhedral complexes has $\mathbb{R}$ as its only nonempty face. Since $\mathcal{P}^{n-1}$ is empty, no hyperplane H such that $S \neq \emptyset$ can exist.

Now fix $n \geq 2$. Assume by contradiction that there exists a hyperplane H such that S is nonempty and contains no line. Let $\widehat{\phi}$ be the refinement of ϕ with respect to H, $\widehat{\mathcal{P}}$

be the underlying polyhedral complex, and $\widehat{\psi}$ be the associated facet-function. Further, we define $\mathcal{Q} := \{P \cap H \mid P \in \widehat{\mathcal{P}}\}$, which is a polyhedral complex that covers H. Note that if H is identified with $\mathbb{R}^{n-1}$ then we can think of $\mathcal{Q}$ as a polyhedral complex that covers $\mathbb{R}^{n-1}$, and the restriction of $\widehat{\psi}$ to $\mathcal{Q}^{n-1}$, which we denote by ϕ', can be seen as a function in $\mathcal{F}^{n-1}(G)$. We will prove that ϕ' does not satisfy the lemma, contradicting the inductive hypothesis.

Since $\phi = \phi_1 + \cdots + \phi_p$, by Observation 3.25 we have $\widehat{\phi} = \widehat{\phi}_1 + \cdots + \widehat{\phi}_p$. Note that for every $i \in [p]$ the hyperplane H is covered by the elements of $\widehat{\mathcal{P}}^{n-1}$. This implies that for every $\widehat{F} \in \widehat{\mathcal{P}}^{n-1}$ and $i \in [p]$ there exists $\widehat{F}_i \in \widehat{\mathcal{P}}_i^{n-1}$ such that $\widehat{F} \subseteq \widehat{F}_i$. Then, by Observation 3.22, $\widehat{\psi}(\widehat{F}) = \widehat{\psi}_1(\widehat{F}_1) + \cdots + \widehat{\psi}_p(\widehat{F}_p)$.

Now, additionally suppose that $\widehat{F}$ is contained in H, that is, $\widehat{F} \in \mathcal{Q}^{n-1}$. Let $i \in [p]$ be such that the lineality space of $\mathcal{P}_i$ is not parallel to H. Then no element of $\mathcal{P}_i^{n-1}$ contains $\widehat{F}_i$. By Observation 3.24, $\widehat{\psi}_i(\widehat{F}_i) = 0$. We then conclude that

$$\widehat{\psi}(\widehat{F}) = \sum_{i \in J} \widehat{\psi}_i(\widehat{F}_i) \quad \text{for every } \widehat{F} \in \mathcal{Q}^{n-1},$$

where J is the set of indices i such that the lineality space of $\mathcal{P}_i$ is parallel to H. This means that

$$\phi' = \sum_{i \in J} \phi_i',$$

where ϕ_i' is the restriction of $\widehat{\psi}_i$ to $\mathcal{Q}_i^{n-1}$, with $\mathcal{Q}_i := \{P \cap H \mid P \in \widehat{\mathcal{P}_i}\}$. Note that for every $i \in J$ the lineality space of $\mathcal{Q}_i$ is clearly nontrivial, as it coincides with the lineality space of $\mathcal{P}_i$.

Now pick any $\widehat{F} \in \mathcal{Q}^{n-1}$. Note that if there exists $F \in \mathcal{P}^{n-1}$ such that $\widehat{F} \subseteq F$, then $\widehat{F} = F$. It then follows from Observation 3.24 that

$$\bigcup \left\{ \widehat{F} \in \mathcal{Q}^{n-1} \;\middle|\; \widehat{\psi}(\widehat{F}) \neq 0 \right\} = S.$$

In other words,

$$\bigcup \left\{ F \in \mathcal{Q}^{n-1} \;\middle|\; \phi'(F) \neq 0 \right\} = S. \tag{3.7}$$

Since $S \neq H$ (as S contains no line), there exists a polyhedron $F \in \mathcal{Q}^{n-1}$ such that $F \subseteq S$ and F has a facet F_0 which does not belong to any other polyhedron in $\mathcal{Q}^{n-1}$ contained in S. Then the facet-function ψ' associated with ϕ' satisfies $\psi'(F_0) \neq 0$. Let H' be the $(n-2)$-dimensional affine space containing F_0. Then the set

$$S' := \bigcup \left\{ F \in \mathcal{Q}^{n-2} \;\middle|\; F \subseteq H',\, \psi'(F) \neq 0 \right\}$$

is nonempty, as $F_0 \subseteq S'$. Furthermore, we claim that S' contains no line. To see why this is true, take any $F \in \mathcal{Q}^{n-2}$ such that $F \subseteq H'$ and $\psi'(F) \neq 0$, and let F', F'' be the two polyhedra in $\mathcal{Q}^{n-1}$ having F as facet. Then $\phi'(F') \neq \phi'(F'')$, and thus at least one of these values (say $\phi'(F')$) is nonzero. Then, by (3.7), $F' \subseteq S$, and thus also $F \subseteq S$. This shows that $S' \subseteq S$ and therefore S' contains no line.

We have shown that ϕ' does not satisfy the lemma. This contradicts the inductive assumption that the lemma holds in dimension $n-1$. □

Finally, we can use this lemma to prove Proposition 3.18.

Proof of Proposition 3.18. Assume for the sake of a contradiction that

$$f_0(x) = \sum_{i=1}^{p} \lambda_i \max\{\ell_{i1}(x), \dots, \ell_{in}(x)\} \quad \text{for every } x \in \mathbb{R}^n,$$

where $p \in \mathbb{N}$, $\lambda_1, \dots, \lambda_p \in \mathbb{R}$ and $\ell_{ij} \colon \mathbb{R}^n \to \mathbb{R}$ is an affine linear function for every $i \in [p]$ and $j \in [n]$. Define $f_i(x) := \lambda_i \max\{\ell_{i1}(x), \dots, \ell_{in}(x)\}$ for every $i \in [p]$, which is a CPWL function.

Fix any $i \in [p]$ such that $\lambda_i \geq 0$. Then f_i is convex. Note that its epigraph

$$E_i := \{(x, z) \in \mathbb{R}^n \times \mathbb{R} \mid z \geq \ell_{ij}(x) \text{ for } j \in [n]\}$$

is a polyhedron in $\mathbb{R}^{n+1}$ defined by n inequalities, and thus has nontrivial lineality space. Furthermore, the line orthogonal to the x-space is not contained in E_i. Since the underlying polyhedral complex $\mathcal{P}_i$ of f_i consists of the orthogonal projections of the faces of E_i (excluding E_i itself) onto the x-space, this implies that $\mathcal{P}_i$ has also nontrivial lineality space. (More precisely, the lineality space of $\mathcal{P}_i$ is the projection of the lineality space of E_i.)

If $\lambda_i < 0$, then f_i is concave. By arguing as above on the convex function $-f_i$, one obtains that the underlying polyhedral complex $\mathcal{P}_i$ has again nontrivial lineality space. Thus this property holds for every $i \in [p]$.

The set of affine linear functions $\mathbb{R}^n \to \mathbb{R}$ forms an abelian group (with respect to the standard operation of sum of functions), which we denote by $(G, +)$. For every $i \in [p]_0$, let ϕ_i be the function in $\mathcal{F}^n(G)$ with underlying polyhedral complex $\mathcal{P}_i$ defined as follows: for every $P \in \mathcal{P}_i^n$, $\phi_i(P)$ is the affine linear function that coincides with f_i over P. Define $\phi := \phi_1 + \dots + \phi_p$ and let $\mathcal{P}$ be the underlying polyhedral complex.

Note that for every $P \in \mathcal{P}^n$, $\phi(P)$ is precisely the affine linear function that coincides with f_0 within P. However, $\mathcal{P}$ may not coincide with $\mathcal{P}_0$, as there might exist $P', P'' \in \mathcal{P}^d$ sharing a facet such that $\phi(P') = \phi(P'')$; when this happens, f_0 is affine linear over $P' \cup P''$ and therefore P' and P'' are merged together in $\mathcal{P}_0$. Nonetheless, $\mathcal{P}$ is a refinement of $\mathcal{P}_0$, i.e., for every $P \in \mathcal{P}_0^n$ there exist $P_1, \dots, P_k \in \mathcal{P}^n$ (for some $k \geq 1$) such that $P = P_1 \cup \dots \cup P_k$. Moreover, $\phi_0(P) = \phi(P_1) = \dots = \phi(P_k)$. Denoting by ψ the facet-function associated with ϕ, this implies for a facet $F \in \mathcal{P}^{n-1}$ that $\psi(F) = 0$ if and only if F is not subset of any facet $F' \in \mathcal{P}_0^{n-1}$.

Let H be a hyperplane as in the statement of the proposition. The above discussion shows that

$$T = \bigcup \left\{ F \in \mathcal{P}_0^{n-1} \;\middle|\; F \subseteq H \right\} = \bigcup \left\{ F \in \mathcal{P}^{n-1} \;\middle|\; F \subseteq H,\, \psi(F) \neq 0 \right\}.$$

Using $S := T$, we obtain a contradiction to Lemma 3.26. □

3.4. A Width Bound for NNs with Small Depth

While the proof of Theorem 2.1 by Arora et al. [Aro+18] shows that

$$\mathrm{CPWL}_n = \mathrm{ReLU}_n(\lceil \log_2(n+1) \rceil),$$

it does not provide any bound on the width of the NN required to represent any particular CPWL function. The purpose of this section is to prove that for fixed dimension n, the required width for exact, depth-minimal representation of a CPWL function can be polynomially bounded in the number p of affine pieces; specifically by $p^{\mathcal{O}(n^2)}$. This is closely related to works that bound the number of linear pieces of an NN as a function of the size [Mon+14; PMB14; Rag+17; MRZ21]. It can also be seen as a counterpart, in the context of exact representations, to quantitative universal approximation theorems that bound the number of neurons required to achieve a certain approximation guarantee; see, e.g., Barron [Bar93; Bar94], Pinkus [Pin99], Mhaskar [Mha96], and Mhaskar and Micchelli [MM95].

3.4.1. The Convex Case

We first derive our result for the case of convex CPWL functions and then use this to also prove the general nonconvex case. Our width bound is a consequence of the following theorem about convex CPWL functions, for which we are going to provide a geometric proof later.

Theorem 3.27. *Let $f(x) = \max\{a_i^T x + b_i \mid i \in [p]\}$ be a convex CPWL function defined on $\mathbb{R}^n$. Then f can be written as*

$$f(x) = \sum_{\substack{S\subseteq[p],\\ |S|\leq n+1}} c_S \max\{a_i^T x + b_i \mid i \in S\}$$

with coefficients $c_S \in \mathbb{Z}$, for $S \subseteq [p]$, $|S| \leq n + 1$.

For the convex case, this yields a stronger version of Theorem 2.3, stating that any (not necessarily convex) CPWL function can be written as a linear combination of $(n + 1)$-term maxima. Theorem 3.27 is stronger in the sense that it guarantees that all pieces of the $(n + 1)$-term maxima must be pieces of the original function, making it possible to bound the total number of these $(n + 1)$-term maxima and, therefore, the size of an NN representing f.

Theorem 3.28. *Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be a convex CPWL function with p affine pieces. Then f can be represented by a ReLU NN with depth $\lceil \log_2(n + 1)\rceil + 1$ and width $\mathcal{O}(p^{n+1})$.*

Proof. Since the number of possible subsets $S \subseteq [p]$ with $|S| \leq n+1$ is bounded by p^{n+1}, the theorem follows by Theorem 3.27 and the construction by Arora et al. [Aro+18] to show Theorem 2.1. □

Before we present the proof of Theorem 3.27, we show how we can generalize its consequences to the nonconvex case.

3.4.2. The General (Nonconvex) Case

It is a well-known fact that every CPWL function can be expressed as a difference of two convex CPWL functions, see, e.g., Wang [Wan04, Theorem 1]. This allows us to derive the general case from the convex case. What we need, however, is to bound the number of affine pieces of the two convex CPWL functions in terms of the number of pieces of the original function. Therefore, we consider a specific decomposition for which such bounds can easily be achieved.

Proposition 3.29. *Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be a CPWL function with p affine pieces. Then, one can write f as $f = g - h$ where both g and h are convex CPWL functions with at most p^{2n+1} pieces.*

Proof. Suppose the p affine pieces of f are given by $x \mapsto a_i^T x + b_i$, $i \in [p]$. Define the function $h(x) \coloneqq \sum_{1\leq i<j\leq p} \max\{a_i^T x + b_i, a_j^T x + b_j\}$ and let $g \coloneqq f + h$. Then, obviously, $f = g - h$. It remains to show that both g and h are convex CPWL functions with at most p^{2n+1} pieces.

The convexity of h is clear by definition. Consider the $\binom{p}{2} = \frac{p(p-1)}{2} < p^2$ hyperplanes given by $a_i^T x + b_i = a_j^T x + b_j$, $1 \leq i < j \leq p$. They divide $\mathbb{R}^n$ into at most $\binom{p^2}{n} + \binom{p^2}{n-1} + \cdots + \binom{p^2}{0} \leq p^{2n}$ regions (compare Edelsbrunner [Ede87, Theorem 1.3]) in each of which h is affine. In particular, h has at most $p^{2n} \leq p^{2n+1}$ pieces.

Next, we show that $g = f + h$ is convex. Intuitively, this holds because each possible breaking hyperplane of f is made convex by adding h. To make this formal, note that by the definition of convexity, it suffices to show that g is convex along each affine line. For this purpose, consider an arbitrary line $x(t) = ta + b$, $t \in \mathbb{R}$, given by $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Let $\tilde{f}(t) \coloneqq f(x(t))$, $\tilde{g}(t) \coloneqq g(x(t))$, and $\tilde{h}(t) \coloneqq h(x(t))$. We need to show that $\tilde{g}\colon \mathbb{R} \to \mathbb{R}$ is a convex function. Observe that $\tilde{f}$, $\tilde{g}$, and $\tilde{h}$ are clearly one-dimensional CPWL functions with the property $\tilde{g} = \tilde{f} + \tilde{h}$. Hence, it suffices to show that $\tilde{g}$ is locally convex around each of its breakpoints. Let $t \in \mathbb{R}$ be an arbitrary breakpoint of $\tilde{g}$. If $\tilde{f}$ is already locally convex around t, then the same holds for $\tilde{g}$ as well since $\tilde{h}$ inherits convexity from h. Now suppose that t is a nonconvex breakpoint of $\tilde{f}$. Then there exist two distinct pieces of f, indexed by $i, j \in [p]$ with $i \neq j$, such that $\tilde{f}(t') = \min\{a_i^T x(t') + b_i, a_j^T x(t') + b_j\}$ for all t' sufficiently close to t. By construction, $\tilde{h}(t')$ contains the summand $\max\{a_i^T x(t') + b_i, a_j^T x(t') + b_j\}$. Thus, adding this summand to $\tilde{f}$ linearizes the nonconvex breakpoint of $\tilde{f}$, while adding all the other summands preserves convexity. In total, $\tilde{g}$ is locally convex around t, which finishes the proof that g is a convex function.

Finally, observe that pieces of $g = f + h$ are always intersections of pieces of f and h, for which we have only $p \cdot p^{2n} = p^{2n+1}$ possibilities. □

Having this, we may conclude the following.

Theorem 3.30. *Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be a CPWL function with p affine pieces. Then f can be represented by a ReLU NN with depth $\lceil \log_2(n+1) \rceil + 1$ and width $\mathcal{O}(p^{2n^2+3n+1})$.*

Proof. Consider the decomposition $f = g - h$ from Proposition 3.29. Using Theorem 3.28, we obtain that both g and h can be represented with the required depth $\lceil \log_2(n+1) \rceil + 1$ and with width $\mathcal{O}((p^{2n+1})^{n+1}) = \mathcal{O}(p^{2n^2+3n+1})$. Thus, the same holds true for f. □

3.4.3. Extended Newton Polyhedra of Convex CPWL Functions

For our proof of Theorem 3.27, we use a correspondence of convex CPWL functions with certain polyhedra, which are known as (extended) Newton polyhedra in tropical geometry [MS15]. These relations between tropical geometry and neural networks have previously been applied to investigate expressivity of NNs; compare our references in Section 3.1.4.

In order to formalize this correspondence, let $\mathrm{CCPWL}_n \subseteq \mathrm{CPWL}_n$ be the set of convex CPWL functions of type $\mathbb{R}^n \to \mathbb{R}$. For $f(x) = \max\{a_i^T x + b_i \mid i \in [p]\}$ in CCPWL_n, we define its so-called *extended Newton polyhedron* to be

$$\mathcal{N}(f) \coloneqq \operatorname{conv}(\{(a_i, b_i) \in \mathbb{R}^n \times \mathbb{R} \mid i \in [p]\}) + \operatorname{cone}(\{-e_{n+1}\}) \subseteq \mathbb{R}^{n+1}.$$

We denote the set of all possible extended Newton polyhedra in $\mathbb{R}^{n+1}$ as Newt_n. That is, Newt_n is the set of (unbounded) polyhedra in $\mathbb{R}^{n+1}$ that emerge from a polytope by adding the negative of the $(n+1)$-st unit vector $-e_{n+1}$ as an extreme ray. Hence, a set $P \subseteq \mathbb{R}^{n+1}$ is an element of Newt_n if and only if P can be written as

$$P = \operatorname{conv}(\{(a_i, b_i) \in \mathbb{R}^n \times \mathbb{R} \mid i \in [p]\}) + \operatorname{cone}(\{-e_{n+1}\}).$$

Conversely, for a polyhedron $P \in \mathrm{Newt}_n$ of this form, let $\mathcal{F}(P) \in \mathrm{CCPWL}_n$ be the function defined by $\mathcal{F}(P)(x) = \max\{a_i^T x + b_i \mid i \in [p]\}$.

There is an intuitive way of thinking about the extended Newton polyhedron P of a convex CPWL function f: it consists of all hyperplane coefficients $(a, b) \in \mathbb{R}^n \times \mathbb{R}$ such that $a^T x + b \leq f(x)$ for all $x \in \mathbb{R}^n$. This also explains why we add the extreme ray $-e_{n+1}$: decreasing b obviously maintains the property of $a^T x + b$ being a lower bound on the function f.

In fact, there is a one-to-one correspondence between elements of CCPWL_n and Newt_n, which is nicely compatible with some (functional and polyhedral) operations. This correspondence has been studied before in tropical geometry [MS15; Jos21], convex geometry[2] [HUL93b], as well as neural network literature [ZNL18; CM18; Alf+20; MRZ21]. We summarize the key findings about this correspondence relevant to our work in the following proposition:

Proposition 3.31. *Let $n \in \mathbb{N}$ and $f_1, f_2 \in \mathrm{CCPWL}_n$. Then it holds that*

(i) *the functions $\mathcal{N}\colon \mathrm{CCPWL}_n \to \mathrm{Newt}_n$ and $\mathcal{F}\colon \mathrm{Newt}_n \to \mathrm{CCPWL}_n$ are well-defined, that is, their output is independent from the representation of the input by pieces or vertices, respectively,*

(ii) *$\mathcal{N}$ and $\mathcal{F}$ are bijections and inverse to each other,*

(iii) $\mathcal{N}(\max\{f_1, f_2\}) = \operatorname{conv}(\mathcal{N}(f_1), \mathcal{N}(f_2)) \coloneqq \operatorname{conv}(\mathcal{N}(f_1) \cup \mathcal{N}(f_2))$,

(iv) $\mathcal{N}(f_1 + f_2) = \mathcal{N}(f_1) + \mathcal{N}(f_2)$, *where the $+$ on the right-hand side is Minkowski addition.*

An algebraic way of phrasing this proposition is as follows: $\mathcal{N}$ and $\mathcal{F}$ are isomorphisms between the semirings $(\mathrm{CCPWL}_n, \max, +)$ and $(\mathrm{Newt}_n, \operatorname{conv}, +)$.

3.4.4. Proof of Theorem 3.27

The rough idea to prove Theorem 3.27 is as follows. Suppose we have a p-term max function f with $p \geq n+2$. By Proposition 3.31, f corresponds to a polyhedron $P \in \mathrm{Newt}_n$ with at least $n+2$ vertices. Applying a classical result from discrete geometry known

[2] $\mathcal{N}(f)$ is the negative of the epigraph of the convex conjugate of f.

as *Radon's theorem* allows us to carefully decompose P into a “signed”[3] Minkowski sum of polyhedra in Newt_n whose vertices are subsets of at most $p-1$ out of the p vertices of P. Translating this back into the world of CPWL functions by Proposition 3.31 yields that f can be written as linear combination of p'-term maxima with $p' < p$, where each of them involves a subset of the p affine terms of f. We can then obtain Theorem 3.27 by iterating until every occurring maximum expression involves at most $n+1$ terms.

We start with a proposition that will be useful for our proof of Theorem 3.27. Although its statement is well-known in the discrete geometry community, we include a proof for the sake of completeness. To show the proposition, we make use of Radon's theorem (compare Edelsbrunner [Ede87, Theorem 4.1]), stating that any set of at least $n+2$ points in $\mathbb{R}^n$ can be partitioned into two nonempty subsets such that their convex hulls intersect.

Proposition 3.32. *Given $p > n+1$ vectors $(a_i, b_i) \in \mathbb{R}^n \times \mathbb{R}$, $i \in [p]$, there exists a nonempty subset $U \subsetneq [p]$ featuring the following property: there is no $c \in \mathbb{R}^{n+1}$ with $c_{n+1} \geq 0$ and $\gamma \in \mathbb{R}$ such that*

$$\begin{aligned} c^T(a_i, b_i) &> \gamma \quad \text{for all } i \in U, \text{ and} \\ c^T(a_i, b_i) &\leq \gamma \quad \text{for all } i \in [p] \setminus U. \end{aligned} \tag{3.8}$$

Proof. Radon's theorem applied to the at least $n+2$ vectors a_i, $i \in [p]$, yields a nonempty subset $U \subsetneq [p]$ and coefficients $\lambda_i \in [0,1]$ with $\sum_{i \in U} \lambda_i = \sum_{i \in [p]\setminus U} \lambda_i = 1$ such that $\sum_{i \in U} \lambda_i a_i = \sum_{i \in [p]\setminus U} \lambda_i a_i$. Suppose that $\sum_{i \in U} \lambda_i b_i \leq \sum_{i \in [p]\setminus U} \lambda_i b_i$ without loss of generality (otherwise exchange the roles of U and $[p] \setminus U$).

For any c and γ that satisfy (3.8) and $c_{n+1} \geq 0$ it follows that

$$\gamma < c^T \sum_{i \in U} \lambda_i (a_i, b_i) \leq c^T \sum_{i \in [p]\setminus U} \lambda_i (a_i, b_i) \leq \gamma,$$

proving that no such c and γ can exist. □

The following proposition is a crucial step in order to show that any convex CPWL function with $p > n+1$ pieces can be expressed as an integer linear combination of convex CPWL functions with at most $p-1$ pieces.

Proposition 3.33. *Let $f(x) = \max\{a_i^T x + b_i \mid i \in [p]\}$ be a convex CPWL function defined on $\mathbb{R}^n$ with $p > n+1$. Then there exist a subset $U \subseteq [p]$ such that*

$$\sum_{\substack{W \subseteq U, \\ |W| \text{ even}}} \max\{a_i^T x + b_i \mid i \in [p] \setminus W\} = \sum_{\substack{W \subseteq U, \\ |W| \text{ odd}}} \max\{a_i^T x + b_i \mid i \in [p] \setminus W\} \tag{3.9}$$

Proof. Consider the $p > n+1$ vectors $(a_i, b_i) \in \mathbb{R}^{n+1}$, $i \in [p]$. Choose U according to Proposition 3.32. We show that this choice of U guarantees equation (3.9).

[3] Some polyhedra may occur with “negative” coefficents in that sum, meaning that they are actually added to P instead of the other polyhedra. The corresponding CPWL functions will then have negative coefficients in the linear combination representing f.

For $W \subseteq U$, let $f_W(x) = \max\{a_i^T x + b_i \mid i \in [p] \setminus W\}$ and consider its extended Newton polyhedron $P_W = \mathcal{N}(f_W) = \operatorname{conv}(\{(a_i, b_i) \mid i \in [p] \setminus W\}) + \operatorname{cone}(\{-e_{n+1}\})$. By Proposition 3.31, equation (3.9) is equivalent to

$$P_{\text{even}} \coloneqq \sum_{\substack{W \subseteq U, \\ |W| \text{ even}}} P_W = \sum_{\substack{W \subseteq U, \\ |W| \text{ odd}}} P_W \eqqcolon P_{\text{odd}},$$

where the sums are Minkowski sums.

We show this equation by showing that for all cost vectors $c \in \mathbb{R}^{n+1}$ it holds that

$$\max\{c^T x \mid x \in P_{\text{even}}\} = \max\{c^T x \mid x \in P_{\text{odd}}\}. \tag{3.10}$$

Let $c \in \mathbb{R}^{n+1}$ be an arbitrary cost vector. If $c_{n+1} < 0$, both sides of (3.10) are infinite. Hence, from now on, assume that $c_{n+1} \geq 0$. Then, both sides of (3.10) are finite since $-e_{n+1}$ is the only extreme ray of all involved polyhedra.

Due to our choice of U according to Proposition 3.32, there exists an index $u \in U$ such that

$$c^T(a_u, b_u) \leq \max_{i \in [p] \setminus U} c^T(a_i, b_i). \tag{3.11}$$

We define a bijection φ_u between the even and the odd subsets of U as follows:

$$\varphi_u(W) \coloneqq \begin{cases} W \cup \{u\}, & \text{if } u \notin W, \\ W \setminus \{u\}, & \text{if } u \in W. \end{cases}$$

That is, φ_u changes the parity of W by adding or removing u. Considering the corresponding polyhedra P_W and $P_{\varphi_u(W)}$, this means that φ_u adds or removes the extreme point (a_u, b_u) to or from P_W. Due to (3.11) this does not change the optimal value of maximizing in c-direction over the polyhedra, that is,

$$\max\{c^T x \mid x \in P_W\} = \max\{c^T x \mid x \in P_{\varphi_u(W)}\}.$$

Hence, we may conclude

$$\begin{aligned} \max\{c^T x \mid x \in P_{\text{even}}\} &= \sum_{\substack{W \subseteq U, \\ |W| \text{ even}}} \max\{c^T x \mid x \in P_W\} \\ &= \sum_{\substack{W \subseteq U, \\ |W| \text{ even}}} \max\{c^T x \mid x \in P_{\varphi_u(W)}\} \\ &= \sum_{\substack{W \subseteq U, \\ |W| \text{ odd}}} \max\{c^T x \mid x \in P_W\} \\ &= \max\{c^T x \mid x \in P_{\text{odd}}\}, \end{aligned}$$

which proves (3.10). Thus, the claim follows. □

With the help of this result, we can now prove Theorem 3.27.

Proof of Theorem 3.27. Let $f(x) = \max\{a_i^T x + b_i \mid i \in [p]\}$ be a convex CPWL function defined on $\mathbb{R}^n$. Having a closer look at the statement of Proposition 3.33, observe that only one term at the left-hand side of (3.9) contains all p affine combinations $a_i^T x + b_i$. Putting all other maximum terms on the other side, we may write f as an integer linear combination of maxima of at most $p - 1$ summands. Repeating this procedure until we have eliminated all maximum terms with more than $n + 1$ summands yields the desired representation. □

3.4.5. Potential Approaches to Show Lower Bounds on the Width

In light of the upper width bounds shown in this section, a natural question to ask is whether also meaningful lower bounds can be achieved. This would mean constructing a family of CPWL functions with p pieces defined on $\mathbb{R}^n$ (with different values of p and n), for which we can prove that a large width is required to represent these functions with NNs of depth $\lceil \log_2(n + 1)\rceil + 1$.

A trivial and not very satisfying answer follows, e.g., from Raghu et al. [Rag+17] or Serra, Tjandraatmadja, and Ramalingam [STR18]: for fixed input dimension n, they show that a function computed by an NN with k hidden layers and width w has at most $\mathcal{O}(w^{kn})$ pieces. For our setting, this means that an NN with logarithmic depth needs a width of at least $\mathcal{O}(p^{1/(n \log n)})$ to represent a function with p pieces. This is, of course, very far away from our upper bounds.

Similar upper bounds on the number of pieces have been proven by many other authors and are often used to show depth-width trade-offs [Mon+14; MRZ21; PMB14; Tel16; Aro+18]. However, there is a good reason why all these results only give rise to very trivial lower bounds for our setting: the focus is always on functions with considerably many pieces, which then, consequently, need many neurons to be represented (with small depth). However, since the lower bounds we strive for depend on the number of pieces, we would need to construct a family of functions with comparably few pieces that still need a lot of neurons to be represented. In general, it seems to be a tough task to argue why such functions should exist.

A different approach could leverage methods from complexity theory, in particular from circuit complexity. Neural networks are basically arithmetic circuits with very special operations allowed. In fact, they can be seen as a tropical variant of arithmetic circuits. Showing circuit lower bounds is a notoriously difficult task in complexity theory, but maybe some conditional result (based on common conjectures similar to P $\neq$ NP) could be established.

We think that the question whether our bounds are tight, or whether at least some non-trivial lower bounds on the width for NNs with logarithmic depth can be shown, is an exciting question for further research.

3.5. Understanding Expressivity via Newton Polytopes

In Section 3.2, we have presented a mixed-integer programming approach towards proving that deep NNs can strictly represent more functions than shallow ones (Conjecture 3.1). In this section, we present another potential approach that is based on Newton polytopes of convex CPWL functions. Using a homogenized version of Proposition 3.31, we provide an equivalent formulation of Conjecture 3.1 that is completely phrased in the language of discrete geometry.

Recall that, by Proposition 3.6, we may restrict ourselves to NNs without biases. In particular, all CPWL functions represented by such NNs, or parts of it, are positively homogeneous. For the associated extended Newton polyhedra (compare Proposition 3.31), this has the following consequence: all vertices $(a, b) \in \mathbb{R}^n \times \mathbb{R}$ lie in the hyperplane $b = 0$, that is, their $(n + 1)$-st coordinate is 0. Therefore, the extended Newton polyhedron of a positively homogeneous, convex CPWL function $f(x) = \max\{a_i^T x \mid i \in [p]\}$ is completely characterized by the so-called *Newton polytope*, that is, the bounded polytope $\operatorname{conv}(\{a_i \mid i \in [p]\})$.

To make this formal, let $\overline{\mathrm{CCPWL}}_n$ be the set of all positively homogeneous, convex CPWL functions of type $\mathbb{R}^n \to \mathbb{R}$ and let $\overline{\mathrm{Newt}}_n$ be the set of all bounded, convex polytopes in $\mathbb{R}^n$. Moreover, for $f(x) = \max\{a_i^T x \mid i \in [p]\}$ in $\overline{\mathrm{CCPWL}}_n$, let

$$\overline{\mathcal{N}}(f) := \operatorname{conv}(\{a_i \mid i \in [p]\}) \in \overline{\mathrm{Newt}}_n$$

be the associated Newton polytope of f and for $P = \operatorname{conv}(\{a_i \mid i \in [p]\}) \in \overline{\mathrm{Newt}}_n$ let

$$\overline{\mathcal{F}}(P)(x) = \max\{a_i^T x \mid i \in [p]\}$$

be the so-called associated *support function* [HUL93a] of P in $\overline{\mathrm{CCPWL}}_n$. With this notation, we obtain the following variant of Proposition 3.31.

Proposition 3.34. *Let $n \in \mathbb{N}$ and $f_1, f_2 \in \overline{\mathrm{CCPWL}}_n$. Then it holds that*

(i) the functions $\overline{\mathcal{N}} \colon \overline{\mathrm{CCPWL}}_n \to \overline{\mathrm{Newt}}_n$ and $\overline{\mathcal{F}} \colon \overline{\mathrm{Newt}}_n \to \overline{\mathrm{CCPWL}}_n$ are well-defined, that is, their output is independent from the representation of the input by pieces or vertices, respectively,

(ii) $\overline{\mathcal{N}}$ and $\overline{\mathcal{F}}$ are bijections and inverse to each other,

(iii) $\overline{\mathcal{N}}(\max\{f_1, f_2\}) = \operatorname{conv}(\overline{\mathcal{N}}(f_1), \overline{\mathcal{N}}(f_2)) := \operatorname{conv}(\overline{\mathcal{N}}(f_1) \cup \overline{\mathcal{N}}(f_2))$,

(iv) $\overline{\mathcal{N}}(f_1 + f_2) = \overline{\mathcal{N}}(f_1) + \overline{\mathcal{N}}(f_2)$, where the $+$ on the right-hand side is Minkowski addition.

In other words, $\overline{\mathcal{N}}$ and $\overline{\mathcal{F}}$ are isomorphisms between the semirings $(\overline{\mathrm{CCPWL}}_n, \max, +)$ and $(\overline{\mathrm{Newt}}_n, \operatorname{conv}, +)$.

Next, we study which polytopes can appear as Newton polytopes of convex CPWL functions computed by NNs with a certain depth; compare Zhang, Naitzat, and Lim [ZNL18], who were the first to rigorously establish this correspondence between NNs and tropical geometry.

Before we apply the first ReLU activation, any function computed by an NN is linear. Thus, the corresponding Newton polytope is a single point. Starting from that, let us investigate a neuron in the first hidden layer. Here, the ReLU activation function computes a maximum of a linear function and 0. Therefore, the Newton polytope of the resulting function is the convex hull of two points, that is, a line segment. After the first hidden layer, arbitrary many functions of this type can be added up. For the corresponding Newton polytopes, this means that we take the Minkowski sum of line segments, resulting in a so-called *zonotope*.

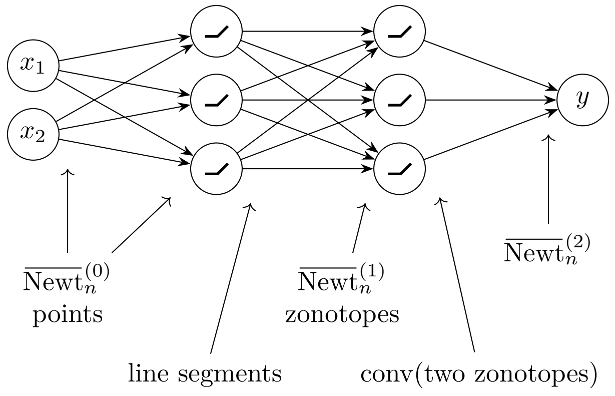

x_1
x_2
y
$\overline{\mathrm{Newt}}_n^{(0)}$
points
line segments
$\overline{\mathrm{Newt}}_n^{(1)}$
zonotopes
conv(two zonotopes)
$\overline{\mathrm{Newt}}_n^{(2)}$

Figure 3.3.: Set of polytopes that can arise as Newton polytopes of convex CPWL functions computed by (parts of) a 2-hidden-layer NN.

Now, this construction can be repeated layerwise, making use of Proposition 3.34: in each hidden layer, we can compute the maximum of two functions computed by the previous layers, which translates to obtaining the new Newton polytope as a convex hull of the union of the two original Newton polytopes. In addition, the linear combinations between layers translate to scaling and taking Minkowski sums of Newton polytopes.

This intuition motivates the following definition. Let $\overline{\mathrm{Newt}}_n^{(0)}$ be the set of all polytopes in $\mathbb{R}^n$ that consist only of a single point. Then, for each $k \geq 1$, we recursively define

$$\overline{\mathrm{Newt}}_n^{(k)} := \left\{ \sum_{i=1}^{p} \mathrm{conv}(P_i, Q_i) \;\middle|\; P_i, Q_i \in \overline{\mathrm{Newt}}_n^{(k-1)},\ p \in \mathbb{N} \right\},$$

where the sum is a Minkowski sum of polytopes. A first, but not precisely accurate interpretation is as follows: the set $\overline{\mathrm{Newt}}_n^{(k)}$ contains the Newton polytopes of positively homogeneous, convex CPWL functions representable with a k-hidden-layer NN. See Figure 3.3 for an illustration of the case $k = 2$.

Unfortunately, this interpretation is not accurate for the following reason: our NNs are allowed to have negative weights, which cannot be fully captured by Minkowski sums as introduced above. Therefore, it might be possible that a k-hidden-layer NN can compute a function that is not contained in $\overline{\mathrm{Newt}}_n^{(k)}$. Luckily, one can remedy this shortcoming, and even extend the interpretation to the non-convex case, by extending the intuition as follows.

Theorem 3.35. *Any positively homogeneous (not necessarily convex) CPWL function can be computed by a k-hidden-layer NN if and only if it can be written as the difference of two positively homogeneous, convex CPWL functions with Newton polytopes in $\overline{\mathrm{Newt}}_n^{(k)}$.*

Proof. We use induction on k. For $k = 0$, the statement is clear since it holds precisely for linear functions. For the induction step, suppose that, for some $k \geq 1$, the equivalence is valid up to $k - 1$ hidden layers. We prove that it is also valid for k hidden layers.

We need to show two directions. First, assume that f is an arbitrary, positively homogeneous CPWL function that can be written as $f = g - h$ with $\mathcal{N}(g), \mathcal{N}(h) \in \overline{\mathrm{Newt}}_n^{(k)}$. We need to show that a k-hidden-layer NN can compute f. We show that this is even true for g and h, and hence, also for f. By definition of $\overline{\mathrm{Newt}}_n^{(k)}$, there exist a finite number $p \in \mathbb{N}$ and polytopes $P_i, Q_i \in \overline{\mathrm{Newt}}_n^{(k-1)}$, $i \in [p]$, such that $\overline{\mathcal{N}}(g) = \sum_{i=1}^p \operatorname{conv}(P_i, Q_i)$. By Proposition 3.34, we have $g = \sum_{i=1}^p \max\{\overline{\mathcal{F}}(P_i), \overline{\mathcal{F}}(Q_i)\}$. By induction, $\overline{\mathcal{F}}(P_i)$ and $\overline{\mathcal{F}}(Q_i)$ can be computed by NNs with $k-1$ hidden layers. Using Proposition 2.2, a k-th hidden layer is then also sufficient to compute g. An analogous argument applies to h. Thus, f is computable with k hidden layers, completing the first direction.

For the other direction, suppose that f is an arbitrary, positively homogeneous CPWL function that can be computed by a k-hidden-layer NN. Let us separately consider the n_k neurons in the k-th hidden layer of the NN. Let a_i, $i \in [n_k]$, be the weight of the connection from the i-th neuron in that layer to the output. Without loss of generality, we have $a_i \in \{\pm 1\}$, because otherwise we can normalize it and multiply the weights of the incoming connections to the i-th neuron with $|a_i|$ instead. Moreover, let us assume that, by potential reordering, there is some $m \le n_k$ such that $a_i = 1$ for $i \le m$ and $a_i = -1$ for $i > m$. With these assumptions, we can write

$$f = \sum_{i=1}^{m} \max\{0, f_i\} - \sum_{i=m+1}^{n_k} \max\{0, f_i\}, \tag{3.12}$$

where each f_i is computable by a $(k-1)$-hidden-layer NN, namely the sub-NN computing the input to the i-th neuron in the k-th hidden layer.

By induction, we obtain $f_i = g_i - h_i$ for some positively homogeneous, convex functions g_i, h_i with $\overline{\mathcal{N}}(g_i), \overline{\mathcal{N}}(h_i) \in \overline{\mathrm{Newt}}_n^{(k-1)}$. We then have

$$\max\{0, f_i\} = \max\{g_i, h_i\} - h_i \tag{3.13}$$

We define

$$g \coloneqq \sum_{i=1}^{m} \max\{g_i, h_i\} + \sum_{i=m+1}^{n_k} h_i$$

and

$$h \coloneqq \sum_{i=1}^{m} h_i + \sum_{i=m+1}^{n_k} \max\{g_i, h_i\}.$$

Note that g and h are convex by construction as a sum of convex functions and that (3.12) and (3.13) imply $f = g - h$. Moreover, by Proposition 3.34,

$$\overline{\mathcal{N}}(g) = \sum_{i=1}^{m} \operatorname{conv}(\overline{\mathcal{N}}(g_i), \overline{\mathcal{N}}(h_i)) + \sum_{i=m+1}^{n_k} \operatorname{conv}(\overline{\mathcal{N}}(h_i), \overline{\mathcal{N}}(h_i)) \in \overline{\mathrm{Newt}}_n^{(k)}$$

and

$$\overline{\mathcal{N}}(h) = \sum_{i=1}^{m} \operatorname{conv}(\overline{\mathcal{N}}(h_i), \overline{\mathcal{N}}(h_i)) + \sum_{i=m+1}^{n_k} \operatorname{conv}(\overline{\mathcal{N}}(g_i), \overline{\mathcal{N}}(h_i)) \in \overline{\mathrm{Newt}}_n^{(k)}.$$

Hence, f can be represented as desired, completing also the other direction. $\square$

The power of Theorem 3.35 lies in the fact that it provides a purely geometric characterization of the class ReLU(k). The classes of polytopes $\overline{\mathrm{Newt}}_n^{(k)}$ are solely defined by the two simple geometric operations Minkowski sum and convex hull of the union.

Therefore, understanding the class ReLU(k) is equivalent to understanding what polytopes one can generate by iterative application of these geometric operations.

In particular, we can give yet another equivalent reformulation of our main conjecture. To this end, let the simplex $\Delta_n \coloneqq \operatorname{conv}\{0, e_1, \ldots, e_n\} \subseteq \mathbb{R}^n$ denote the Newton polytope of the function $f_n = \max\{0, x_1, \ldots, x_n\}$ for each $n \in \mathbb{N}$.

Conjecture 3.36. *For any $k \in \mathbb{N}$, $n = 2^k$, there do not exist polytopes $P, Q \in \overline{\mathrm{Newt}}_n^{(k)}$ such that P is the Minkowski sum of Δ_n and Q.*

Theorem 3.37. *Conjecture 3.36 is equivalent to Conjecture 3.1 and Conjecture 3.2.*

Proof. By Proposition 3.3, it suffices to show equivalence between Conjecture 3.36 and Conjecture 3.2. By Theorem 3.35, f_n can be represented with k hidden layers if and only if there are functions g and h with Newton polytopes in $\overline{\mathrm{Newt}}_n^{(k)}$ satisfying $f_n + h = g$. By Proposition 3.34, this happens if and only if there are polytopes $P, Q \in \overline{\mathrm{Newt}}_n^{(k)}$ with $\Delta_n + Q = P$. □

It is particularly interesting to look at special cases with small k. For $k = 1$, the set $\overline{\mathrm{Newt}}_n^{(1)}$ is the set of all zonotopes. Hence, the (known) statement that $\max\{0, x_1, x_2\}$ cannot be computed with one hidden layer [MB17] is equivalent to the fact that the Minkowski sum of a zonotope and a triangle can never be a zonotope.

The first open case is the case $k = 2$. An unconditional proof that two hidden layers do not suffice to compute the maximum of five numbers is highly desired. In the regime of Newton polytopes, this means to understand the class $\overline{\mathrm{Newt}}_n^{(2)}$. It consists of finite Minkowski sums of polytopes that arise as the convex hull of the union of two zonotopes. Hence, the major open question here is to classify this set of polytopes.

Finally, let us remark that there exists a generalization of the concept of polytopes, known as *virtual polytopes* [PS15], that makes it possible to assign a Newton polytope also to non-convex CPWL functions. This makes use of the fact that every (non-convex) CPWL function is a difference of two convex ones. Consequently, a virtual polytope is a formal Minkowski difference of two ordinary polytopes. Using this concept, Theorem 3.35 and Conjecture 3.36 can be phrased in a simpler way, replacing the pair of polytopes with a single virtual polytope.

3.6. Future Research

The most obvious and, at the same time, most exciting open research question is to prove or disprove Conjecture 3.1, or equivalently Conjecture 3.2 or Conjecture 3.36. The first step could be to prove Assumption 3.7. The assumption is intuitive because every breakpoint introduced at any place outside the hyperplanes H_{ij} needs to be canceled out later. Therefore, it is natural to assume that these breakpoints do not have to be introduced in the first place. However, this intuition does not seem to be enough for a formal proof because it could occur that additional breakpoints in intermediate steps, which are canceled out later, also influence the behavior of the function at other places where we allow breakpoints in the end.

Another step towards resolving our conjecture may be to find an alternative proof of Theorem 3.8, not using Assumption 3.7. This might also be beneficial for generalizing our techniques to more hidden layers, since, while theoretically possible, a direct generalization of the MIP approach is infeasible due to computational limitations. For example, it might be particularly promising to use a tropical approach as described in Section 3.5 and apply methods from discrete geometry to prove Conjecture 3.36.

In light of our results from Section 3.3, it would be desirable to provide a complete characterization of the functions contained in $\mathrm{ReLU}(k)$. Another potential research goal is improving our upper bounds on the width from Section 3.4 and/or proving matching lower bounds as discussed in Section 3.4.5.

Some more interesting research directions are the following:

- establishing or strengthening our results for special classes of NNs like recurrent neural networks (RNNs) or convolutional neural networks (CNNs),
- using exact representation results to show more drastic depth-width trade-offs compared to existing results in the literature,
- understanding how the class $\mathrm{ReLU}(k)$ changes when a polynomial upper bound is imposed on the width of the NN; see related work by Vardi et al. [Var+21].

4 Computational Power

Which network size is sufficient to solve (combinatorial optimization) problems?

The results in this chapter are based on a preprint with Leon Sering [HS21a] and a paper with Martin Skutella [HS21b] that appeared in the proceedings of the AAAI 2021 conference.

4.1. Introduction

Traditionally, neural networks (NNs) have been primarily used for "soft" tasks. For example, in character recognition, there is no clear mathematical definition of when a certain pixel array represents a certain digit. For tasks of that flavor, NNs seem to be particularly well-suited due to their ability to model highly complex decision boundaries that somehow resemble our intuitive understanding of such soft tasks.

Combinatorial optimization (CO) problems do not belong to this class of soft tasks. They have a clear mathematical definition. Consequently, exact combinatorial methods or mathematical programming have been the classical tools to tackle these problems. For hard problems in practice, traditional heuristics without learning components have been the method of choice.

However, in recent years, a lot of research has emerged that utilizes NNs for exact tasks including CO problems. Depending on the use-case, it turns out that NN approaches are in fact a promising way to practically solve CO problems or to enhance classical solution methods. We refer to Bengio, Lodi, and Prouvost [BLP21] for a survey of this stream of research.

Still, the vast majority of these approaches suffers from the same problem as other NN algorithms do: it is hard to give mathematical guarantees concerning the solution quality or the required computing resources. Also, it seems to be impossible to truly understand how and why these algorithms work.

In order to enhance our understanding of (ReLU) NNs' computational power, that is, their theoretical ability to solve difficult problems, we propose to view them as a model of computation operating on real numbers. This is similar to Boolean circuits, or, even closer, arithmetic circuits, which are well-studied objects in complexity theory and computer science in general. We then investigate the computational complexity of various problems, including the Maximum Flow Problem and the Knapsack Problem, within that model.

4.1.1. Neural Networks as a Model of Computation

From the definition of NNs, they are basically *arithmetic circuits* with gates that can compute (weighted) sums and maxima. This has the following two natural implications.

Firstly, as in other arithmetic models of computation, all operations take "unit cost", that is, our complexity analysis is completely independent from the encoding size of the input numbers involved.

Secondly, note that a (feedforward) NN has a fixed number of input neurons. If we would like to solve a CO problem (e.g., on a weighted graph) then the number of required input neurons grows with the size of the instance (e.g., the number of vertices or edges of the input graph). Therefore, as usual in the theory of circuit complexity [AB09], we do not consider single NNs, but so-called *families* of NNs, parameterized by the number of inputs of the considered optimization problem (e.g., number of vertices or edges of the input graph). Then, the question of interest is how the size (depth, width) of the NNs in that family grows with the number of inputs.

Even though NNs are naturally a model of real computation, it is worth to have a look at their computational power with respect to Boolean inputs. It is easy to see that ReLU NNs can directly simulate AND-, OR-, and NOT-gates, and thus also any Boolean circuit; see [MB17]. Hence, in Boolean arithmetics, any problem in P can be solved with polynomial-size NNs. However, to grasp the full nature of NNs, one needs to consider real instead of Boolean arithmetics. In this context, things are much less clear.

Of course, if there are polynomial-size NNs to solve a certain problem, then there exists a strongly polynomial time algorithm for that problem, simply by executing the NN. However, the converse might not be true. This is due to the fact that ReLU NNs only allow a very limited set of possible operations, namely affine combinations and maxima computations. In particular, any function computed by such NNs is continuous, making it impossible to realize instructions like a simple **if**-branching based on a comparison of real numbers. In fact, for some models of computation, the use of branchings is exponentially powerful [JS82].

An example of such a conditional branching is to decide whether an arc is part of the *residual network*, which is a crucial step in classical maximum flow algorithms. Therefore, these algorithms cannot be implemented with NNs and, without our results in Section 4.5, it remains unclear whether polynomially sized NNs to solve the Maximum Flow Problem exist at all.

4.1.2. Neural Networks and Arithmetic Circuits

In a broad sense, *arithmetic circuits* are directed acyclic graphs where each node computes some expression from the outputs of all its predecessors. Then, the full graph defines a function mapping the inputs, located at the sources of the graph, to the outputs, located at the sinks of the graph. In this sense, ReLU feedforward NNs are a special type of arithmetic circuits. In a narrow sense, the inner nodes of an arithmetic circuit are either sum or product nodes, which output the sum or product of all the outputs of their predecessors. This is, of course, different from NNs, which cannot compute products, but still closely related. There has been a lot of research about the complexity of arithmetic circuits [SY10]. Particularly relevant to our work, there is a special kind of arithmetic circuits called *tropical circuits* [Juk15]. In contrast to ordinary arithmetic circuits, they only contain maximum (or minimum) gates instead of sum gates and sum gates instead of product gates. Thus, they are arithmetic circuits in max-plus arithmetics.

Obviously, a tropical circuit can be simulated by an NN of roughly the same size since NNs can compute maxima and sums. Thus, NNs are at least as powerful as tropical circuits. However, NNs are strictly more powerful. In particular, lower bounds on the size of tropical circuits do not apply to NNs. A particular example is the computation of the value of a minimum spanning tree. By Jukna and Seiwert [JS19], no polynomial-size tropical circuit can do this. However, as we show in Section 4.4 as a consequence of a result by Fomin et al. [FGK16], an NN of cubic size (in the number of nodes of the input graph) is sufficient for this task.

The reason for this exponential gap is that, by using negative weights, NNs can realize subtractions (that is, tropical division), which is not possible with tropical circuits; compare the discussion by Jukna and Seiwert [JS19]. However, this is not the only feature that makes NNs more powerful than tropical circuits. In addition, NNs can realize scalar multiplication (tropical exponentiation) with arbitrary real numbers via their weights, which is impossible with tropical circuits. It is unclear to what extent this feature increases the computational power of NNs compared to tropical circuits.

For these and similar reasons, lower bounds from arithmetic circuit complexity do not transfer to NNs. Therefore, we identify it as a major challenge to provide meaningful lower bounds of any kind for the computational model of NNs.

4.1.3. Exact Neural Networks for CO Problems

Next, we discuss some preliminary observations concerning the ability of NNs to compute exact solutions for combinatorial optimization problems.

By Theorem 2.1, the set of functions computable by ReLU NNs is precisely the set of continuous, piecewise linear (CPWL) functions. Even though the proof shows that logarithmic depth (in the input dimension) is always sufficient, these networks might have huge width such that no bounds on the total network size can be obtained.

Many functions related to CO problems are CPWL and can thus be represented by an NN (of any size). In fact, for a feasible set $\mathcal{X} \subseteq \{0,1\}^n$ consider the generic CO problem $\min_{x\in\mathcal{X}} c^T x$. This formulation covers a broad range of CO problems, among them efficiently solvable problems like Shortest Path and Matching Problems, the Minimum Spanning Tree Problem, the Maximum Flow Problem (via duality to the Minimum Cut Problem), and also NP-hard problems like the Knapsack or the Traveling Salesperson Problem. When considering the cost vector c to be a variable, the function $c \mapsto \min_{x\in\mathcal{X}} c^T x$ mapping c to the objective value is a minimum of finitely many linear functions and thus CPWL. For NP-hard problems it is unlikely that polynomial-size NNs can compute this function. However, even for problems in P, the existence of polynomial-size NNs is unclear due to the limited set of operations available.

Note that, due to their continuous nature, ReLU NNs (without threshold gates or similar) that output the discrete solution vector $x \in \{0,1\}^n$ (and not only the objective value) cannot exist. Therefore, if we say an NN family *solves* a certain CO problem, we usually refer to computing the objective value instead of the discrete solution vector. However, for the Maximum Flow Problem the situation is slightly different since it is not the primal Maximum Flow Problem but the dual Minimum Cut Problem that is represented in the generic form above. In fact, using an appropriate tie-breaking rule between equally good flows, the vector denoting a maximum flow is a CPWL function of the arc capacities. Hence, Theorem 2.1 guarantees that a ReLU NN computing this maximum flow function of type $\mathbb{R}^m \to \mathbb{R}^m$ (m being the number of arcs in the network) is possible.

4.1.4. Neural Networks and Parallel Computation

Similar to Boolean circuits, NNs can be seen as a model of parallel computation since the execution of all neurons of one layer can be done in parallel. Without going into detail here, the depth of an NN is related to the running time of a parallel algorithm, its width is related to the required number of processing units, and its size to the total amount of work conducted by the algorithm. Against this background, a natural goal is to design NNs as shallow as possible in order to make maximal use of parallelization. However, several results in the area of NN expressivity state that decreasing the depth is often only possible at the cost of an *exponential* increase in width; see [Aro+18; ES16; LS17; SS17; Tel15; Tel16; Yar17].

Interestingly, a related observation can be made for the Maximum Flow Problem using complexity theory. As argued above, Theorem 2.1 implies the existence of a ReLU NN to solve the Maximum Flow Problem. In fact, Theorem 2.1 even ensures that a logarithmic depth (in the input dimension) is sufficient for that task. It arises the question whether such shallow NNs are also possible while maintaining polynomial total size.

The answer is most likely "no" because it has been shown that the Maximum Flow Problem is *P-complete* [GSS82]. P-complete problems are those problems in P that are *inherently sequential*, meaning that there cannot exist a parallel algorithm with polylogarithmic running time using a polynomial number of processors unless the complexity classes P and NC coincide, which is conjectured to be not the case [GHR95]. NNs with polylogarithmic depth and polynomial total size that solve the Maximum Flow Problem, however, would translate to such an algorithm (under mild additional conditions, such as, that the weights of the NN can be computed in polynomial time). Therefore, we conclude that it is unlikely to obtain NNs for the Maximum Flow Problem that make significant use of parallelization. In other words, NNs with polylogarithmic depth and polynomial width solving the Maximum Flow Problem probably do not exist.

4.1.5. Further Related Work

NNs and circuit complexity. NNs have been studied from a circuit complexity point of view before [BT96; PGM94; STAK92]. However, these works focus on Boolean circuit complexity of NNs with sigmoid or threshold activation functions. We are not aware of previous work investigating the computational power of ReLU NNs as arithmetic circuits operating on the real numbers.

Early NN approaches for CO problems. The idea of using NNs to practically solve CO problems became popular with so-called *Hopfield networks* [HT85] and related architectures in the 1980s and has been extended to general nonlinear programming problems later on [KC88]. Smith [Smi99] surveys these early approaches. Hopfield NNs are special versions of recurrent neural networks (RNNs) that find solutions to optimization problems by converging towards a minimum of an energy function. While most authors mainly focus on the Traveling Salesperson Problem (TSP), Ohlsson, Peterson, and Söderberg [OPS93] study a so-called mean field NN for (generalizations of) the Knapsack Problem and empirically assess the quality of its solutions. Also, specific NNs to solve the Maximum Flow Problem have been developed before [AK91; ER08; NO12].

However, the NNs used in these early works are conceptually very different from modern feedforward NNs that are considered in this thesis.

Modern NN approaches for CO problems. While there has been less research at the intersection of CO and NNs in the 2000s, modern advances in the area of deep learning have boosted the interest in this direction again; see Bengio, Lodi, and Prouvost [BLP21] for a general review and Cappart et al. [Cap+21] for a focused review on graph neural networks. Common applications include speeding up solvers for mixed-integer linear programs, for instance, by automatically learning on which variables to branch in branch-and-bound algorithms; see Lodi and Zarpellon [LZ17] for a survey. Machine learning has also been applied to modeling aspects of CO, as reviewed by Lombardi and Milano [LM18], and to several specific CO problems, where the TSP is often one of them [Bel+16; ER18; Kha+17a; KHW19; Now+17; VFJ15]. The different methods used by these authors include feedforward and recurrent neural networks, reinforcement learning, attention mechanisms, pointer networks, graph embeddings, and graph neural networks.

Modern NN approaches for the Knapsack Problem. There have also been specific applications of neural networks to the Knapsack Problem. For example, Bello et al. [Bel+16] utilize an RNN trained by reinforcement learning and Gu and Hao [GH18] use a pointer network for to find empirically good knapsack solutions. Li et al. [Li+21] derived heuristics inspired by game theory to solve a non-linear knapsack version whose objective function is given by a neural network.

NNs and dynamic programming. Particularly related to our work are interactions between neural networks and dynamic programming algorithms. For example, Yang et al. [Yan+18] and Xu et al. [Xu+20b] use NNs to speed up dynamic programming algorithms for CO problems. The key difference to our work, however, is that NNs are used as heuristics in these papers, making it virtually impossible to give any meaningful worst-case performance guarantees. Another interesting research stream deals with the learnability of algorithms. In this context, Xu et al. [Xu+20a] have developed the concept of *algorithmic alignment* and show that dynamic programming algorithms align well with graph neural networks. A more empirical study in this direction was performed by Veličković et al. [Vel+20]. These results concentrate on learnability in contrast to our focus on expressivity. Still, they agree with the message of our work that dynamic programming is a good paradigm for bringing classical algorithms and neural networks closer together.

Expressivity of NNs. Studying the computational power of NNs, that is, their ability to precisely solve difficult problems, is somehow a special way of investigating their expressivity; compare Chapter 3. Therefore, we recall some of the key results about NN expressivity from the literature. So-called *universal approximation theorems* [AB99; Cyb89; Hor91] state that only one hidden layer is already sufficient to approximate any continuous function on a compact domain arbitrarily well. Unfortunately, no insights concerning exact representability can be obtained from that. Various trade-offs between depth and width of NNs [Aro+18; ES16; Han19; HS17; LS17; NMH18; Rag+17; SS17;

Tel15; Tel16; Yar17] and approaches to count and bound the number of linear regions of a ReLU NN [HR19; Mon+14; PMB14; Rag+17; STR18] have been found. Mukherjee and Basu [MB17] proved size lower bounds to represent Boolean functions with NNs of limited depth.

Textbooks on combinatorial optimization. For an introduction to classical combinatorial optimization, we refer to the textbook by Korte and Vygen [KV08]. Theory of polyhedra as well as linear and integer optimization can be found in the textbook by Schrijver [Sch86]. For specific introductions to network flow problems we recommend the textbooks by Ahuja, Magnanti, and Orlin [AMO93] and Williamson [Wil19].

4.1.6. Overview and Main Results

The overall contribution of this chapter is the idea to analyze the computational power of ReLU NNs by viewing them as a model of real-valued computation that is related to arithmetic/tropical circuits. Our specific results are of the following flavor: For a bunch of CO problems we show that (comparably) small NNs solving these problems exist. We achieve this by providing explicit manual constructions without too much loss in efficiency compared to algorithms in other models of computation.

In order to make it possible to think about NNs in an algorithmic way, we introduce the pseudo-code language *Max-Affine Arithmetic Programs* (MAAPs) in Section 4.2. We show that MAAPs and NNs are basically equivalent (up to constant factors) concerning three basic complexity measures corresponding to depth, width, and overall size of NNs. Hence, MAAPs serve as a convenient tool for the manual construction of NNs with bounded size and could be useful for further research concerning NNs' computational power far beyond the scope of this thesis.

Turning towards specific CO problems, we first showcase some problems for which MAAPs, and hence NNs, can be constructed in a straight-forward way from well-known algorithms, in particular, from dynamic programs in Section 4.3. This includes the Longest Common Subsequence Problem, various variants of Shortest Path Problems, as well as the Traveling Salesperson Problem.

Afterwards, in Section 4.4, we turn to the Minimum Spanning Tree Problem, where we demonstrate how a result from arithmetic circuit complexity translates to an NN of size $\mathcal{O}(n^3)$ mapping edge weights of an n-vertex graph to the value of a minimum spanning tree.

Then, in Section 4.5, we study the Maximum Flow Problem, for which all attempts to model classical algorithms as NNs seem to fail. For that reason, we develop a completely new maximum flow algorithm in the form of a MAAP, which can be transformed to an NN of size $\mathcal{O}(m^2n^2)$ that, given a directed graph with n nodes and m arcs, computes a maximum flow from any possible real-valued arc capacities as input. We would like to mention that this result also has an interesting interpretation from the perspective of parametric algorithms. There exists a variety of literature concerning the question how one could represent solutions to the maximum flow problem if the input depends on one or several unknown parameters; see, e.g., the works by Gallo, Grigoriadis, and Tarjan [GGT89] and McCormick [McC99]. An NN mapping arc capacities to a maximum flow can be seen as such a representation for the most general form of parametric maximum flow problems, namely the one where *all* arc capacities are independent free parameters. We believe it is an interesting and useful observation that such a representation of polynomial size exists.

Afterwards, in Section 4.6, we focus on the Knapsack Problem. We show that a recurrent neural network (RNN) of depth four and width depending quadratically on the profit of an optimum Knapsack solution is sufficient to find optimum knapsack solutions. Again, the idea behind this construction is to mimic a classical dynamic program for the Knapsack Problem. However, in order to always filter out the correct entries of the previous state space in each step, additional technical difficulties need to be overcome. We also prove the following trade-off between the size of an RNN and the quality of the computed knapsack solution: for knapsack instances consisting of n items, an RNN of depth five and width w computes a solution of value at least $1 - \mathcal{O}(n^2/\sqrt{w})$ times the optimum solution value. This result relies on a subtle variant of the rounding procedure that turns the pseudo-polynomial dynamic program into a fully polynomial-time approximation scheme for the Knapsack Problem. Additionally, we provide a carefully conducted computational study to qualitatively support our theoretical size bounds.

Finally, we outline how the results about the Knapsack Problem can be generalized to other CO problems, specifically the Constrained Shortest Path Problem.

At the end of the chapter, in Section 4.8, we provide an overview of related open problems.

4.1.7. Chapter-Specific Notation: Bold Symbols

In this chapter, we deal with manual NN constructions and associated pseudo-code programs (MAAPs). In such a construction, the parameters of the NNs, like architectural details, weights, and biases, as well as constants in MAAPs, are fixed, while inputs and outputs to NNs and MAAPs, intermediate activation values of NNs and variables of MAAPs depend on the concrete input values. To understand the constructions in this chapter, it is crucial to distinguish these two types of values (fixed and input-dependent) from each other. In order to make the difference visible, we denote all input-dependent values by bold symbols.

Note that this is in contrast to Chapter 5, where bold symbols are used to distinguish vectors from scalars.

4.2. Max-Affine Arithmetic Programs

One way of specifying an NN is by explicitly writing down the network architecture, weights, and biases, that is, the affine transformations of each layer. However, for NNs that mimic the execution of an algorithm this is very unhandy and not well readable. For the purpose of an easier notation we introduce a pseudo-code language, called *Max-Affine Arithmetic Programs (MAAPs)*, and prove that it is essentially equivalent to NNs.

MAAPs perform arithmetic operations on real-valued *variables*. They consist of different kinds of *instructions* and may also involve real-valued *constants*. In order to distinguish constants from variables, the latter will be denoted by bold symbols. Each MAAP consists of a fixed number of input and output variables, as well as a sequence of (possibly nested) instructions.

In order to describe an algorithm with an arbitrary number of input variables and to be able to measure asymptotic complexity, we specify a *family* of MAAPs that is parametrized by a natural number that determines the number of input variables and is treated like a constant in each single MAAP of the family. A MAAP family then corresponds to a family of NNs; see Section 4.1.1.

MAAPs consist of the following types of instructions:

1. Assignment: this instruction assigns an expression to an old or new variable. The only two types of allowed expressions are affine combinations or maxima of affine combinations of variables: $b + \sum_j c_j \mathbf{v}_j$ and $\max \left\{ b^{(i)} + \sum_j c_j^{(i)} \mathbf{v}_j^{(i)} \;\middle|\; i = 1, \ldots, n \right\}$, where $n \in \mathbb{N}$, $b, b^{(i)}, c_j, c_j^{(i)} \in \mathbb{R}$ are constants and $\mathbf{v}_j, \mathbf{v}_j^{(i)} \in \mathbb{R}$ are variables. Without loss of generality minima are also allowed.

2. **Do-Parallel**: this instruction contains a constant number of blocks of instruction sequences, each separated by an **and**. These blocks must be executable in parallel, meaning that each variable that is assigned in one block cannot appear in any other block.[1]

3. **For-Do** loop: this is a standard for-loop with a *constant number of iterations* that are executed sequentially.

4. **For-Do-Parallel** loop: this is a for-loop with a *constant number of iterations* in which the iterations are executed in parallel. Therefore, variables assigned in one iteration cannot be used in any other iteration.[2]

Algorithm 1 shows an example MAAP to illustrate the possible instructions.

Note that we do not allow any **if**-statements or other branching operations. In other words, the number of executed instructions of an algorithm is always the same independent of the input variables.

In order to connect MAAPs with NNs, we introduce three complexity measures $d(A)$, $w(A)$, and $s(A)$ for a MAAP A. We will then see that they yield a direct correspondence to depth, width, and size of a corresponding NN.

For these complexity measures for MAAPs, assignments of affine transformations come "for free" since in an NN this can be realized "between layers". This is a major difference to other (parallel) models of computation, e.g., the parallel random access machine (PRAM) [GHR95]. Apart from that, the complexity measures are recursively defined as follows.

- For an assignment A with a maximum or minimum expression of $k \geq 2$ terms we have $d(A) := \lceil \log_2 k \rceil$, $w(A) := 2k$, and $s(A) := 3k$.[3]
- For a sequence A of instruction blocks $B_1, B_2, \ldots, B_k$ we have $d(A) := \sum_{i=1}^{k} d(B_i)$, $w(A) := \max_{i=1}^{k} w(B_i)$, and $s(A) := \sum_{i=1}^{k} s(B_i)$.

[1]Local variables in different blocks may have the same name if their scope is limited to their block.

[2]Again, local variables within different iterations may have the same name if their scope is limited to their iteration.

[3]These definitions are aligned with the depth, width, and size bounds of Proposition 2.2, ceiled for more convenient writing.

Algorithm 1: Example MAAP to illustrate the possible instructions.

```
   Input: Input variables $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$.
   // Assignments and Expressions:
1  $\mathbf{x}_1 \leftarrow 4 + \sum_{i=1}^{n} (-1)^i \cdot \mathbf{v}_i$
2  $\mathbf{x}_2 \leftarrow \max\{3 \cdot \mathbf{v}_1, -1.5 \cdot \mathbf{v}_n, \mathbf{x}_1, 5\}$
   // For-Do loop:
3  for $k = 1, \ldots, n-1$ do
4      $\mathbf{v}_{k+1} \leftarrow \mathbf{v}_k + \mathbf{v}_{k+1}$
   // Do-Parallel:
5  do parallel
6      $\mathbf{y}_1 \leftarrow \max\{\mathbf{x}_1, \mathbf{x}_2\}$
7  and
8      $\mathbf{y}_2 \leftarrow 7$
9  and
10     $\mathbf{y}_3 \leftarrow \sum_{i=1}^{n} \mathbf{v}_i$
   // For-Do-Parallel loop:
11 for each $k = 4, \ldots, n$ do parallel
12     $\mathbf{y}_k \leftarrow \mathbf{v}_{k-1} - \mathbf{v}_k$
13     $\mathbf{y}_k \leftarrow \max\{\mathbf{y}_k, 0\}$
14 return $(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n)$
```

- For a **Do-Parallel** instruction A consisting of blocks $B_1, B_2, \ldots, B_k$ we have $d(A) \coloneqq \max_{i=1}^{k} d(B_i)$, $w(A) \coloneqq \sum_{i=1}^{k} w(B_i)$, and $s(A) \coloneqq \sum_{i=1}^{k} s(B_i)$.
- For a **For-Do** loop A with k iterations that executes block B_i in iteration i we have $d(A) \coloneqq \sum_{i=1}^{k} d(B_i)$, $w(A) \coloneqq \max_{i=1}^{k} w(B_i)$, and $s(A) \coloneqq \sum_{i=1}^{k} s(B_i)$.
- For a **For-Do-Parallel** loop with k iterations that executes block B_i in iteration i we have $d(A) \coloneqq \max_{i=1}^{k} d(B_i)$, $w(A) \coloneqq \sum_{i=1}^{k} w(B_i)$, and $s(A) \coloneqq \sum_{i=1}^{k} s(B_i)$.

The following proposition establishes the desired correspondence between the complexities of MAAPs and NNs.

Proposition 4.1. *For a function $f \colon \mathbb{R}^n \to \mathbb{R}^m$ the following is true.*

(i) If f can be computed by a MAAP A, then it can also be computed by an NN with depth $d(A) + 1$, width $w(A)$, and size $s(A)$.

(ii) If f can be computed by an NN with depth $d + 1$, width w, and size s, then it can also be computed by a MAAP A with $d(A) = d$, $w(A) = 2w$, and $s(A) = 3s$.

Proof.

(i) First note that we can assume without loss of generality that A does not contain **For-Do** or **For-Do-Parallel** loops. Indeed, since only a constant number of iterations is allowed in both cases, we can write them as a sequence of blocks or a

Algorithm 2: A generic MAAP to execute a given NN.

```
Input: Input variables $\mathbf{o}(v)$ for $v \in V_0$.
// For each hidden layer:
1 for $\ell = 1, \dots, d$ do
      // For each neuron in the layer:
2     for each $v \in V_\ell$ do parallel
3         $\mathbf{a}(v) \leftarrow b_v + \sum_{u\colon (u,v)\in E} w_{uv}\mathbf{o}(u)$
4         $\mathbf{o}(v) \leftarrow \max\{0, \mathbf{a}(v)\}$
// For each output neuron:
5 for each $v \in V_{d+1}$ do parallel
6     $\mathbf{a}(v) \leftarrow b_v + \sum_{u\colon (u,v)\in E} w_{uv}\mathbf{o}(u)$
7 return $(\mathbf{a}(v))_{v\in V_{d+1}}$.
```

Do-Parallel instruction, respectively. Note that this also does not alter the complexity measures $d(A)$, $w(A)$, and $s(A)$ by their definition. Hence, suppose for the remainder of the proof that A consists only of assignments and (possibly nested) **Do-Parallel** instructions.

The statement is proven by an induction on the number of lines of A. For the induction base suppose A consists of a single assignment. If this is an affine expression, then an NN without hidden units (and hence with depth 1, width 0, and size 0) can compute f. If this is a maximum (or minimum) expression of k terms, then, using Proposition 2.2, an NN with depth $\lceil \log_2 k \rceil + 1$, width $2k$, and size $3k$ can compute f, which settles the induction base.

For the induction step we consider two cases. If A can be written as a sequence of two blocks B_1 and B_2, then, by induction, there are two NNs representing B_1 and B_2 with depth $d(B_i) + 1$, width $w(B_i)$, and size $s(B_i)$ for $i = 1, 2$, respectively. An NN representing A can be obtained by concatenating these two NNs in series, yielding an NN with depth $d(B_1) + d(B_2) + 1 = d(A) + 1$, width $\max\{w(B_1), w(B_2)\} = w(A)$, and size $s(B_1) + s(B_2) = s(A)$. Otherwise, A consists of a unique outermost **Do-Parallel** instruction with blocks $B_1, B_2, \dots, B_k$. By induction, there are k NNs representing B_i with depth $d(B_i) + 1$, width $w(B_i)$, and size $s(B_i)$, $i \in [k]$, respectively. An NN representing A can be obtained by plugging all these NNs in parallel next to each other, resulting in an NN of depth $\max_{i=1}^k d(B_i) + 1 = d(A) + 1$, width $\sum_{i=1}^k w(B_i) = w(A)$, and size $\sum_{i=1}^k s(B_i) = s(A)$. This completes the induction.

(ii) Suppose the NN is given by a directed graph $G = (V, E)$ as defined in Section 2.1. It is easy to verify that Algorithm 2 computes f with the claimed complexity measures. □

Algorithm 3: MAAP to find the length of the longest common subsequence for given lengths m and n of two integer sequences.

```
Input: Integer sequences $\mathbf{x}_1, \dots, \mathbf{x}_m$ and $\mathbf{y}_1, \dots, \mathbf{y}_n$.
// Initialization:
1 for $i = 0, \dots, m$ do
2     $\mathbf{f}(i, 0) \leftarrow 0$
3 for $j = 1, \dots, n$ do
4     $\mathbf{f}(0, j) \leftarrow 0$
// Recursion (can be done in parallel for all index pairs with same
   sum $i + j$):
5 for $s = 2, \dots, m + n$ do
6     for each pair $(i, j)$ with $i + j = s$, $i \in [m]$, $j \in [n]$ do parallel
7         $\mathbf{d} \leftarrow \max\{\mathbf{x}_i - \mathbf{y}_j, \mathbf{y}_j - \mathbf{x}_i\}$
8         $\mathbf{f}(i, j) \leftarrow \max\{\mathbf{f}(i-1, j-1) + 1 - \mathbf{d}, \mathbf{f}(i-1, j), \mathbf{f}(i, j-1)\}$
9 return $\mathbf{f}(m, n)$.
```

4.3. Some Starting Examples

In this section we provide some computational problems for which standard algorithms can be written in the form of a MAAP and, hence, implemented on an NN. In these cases, the running time of the corresponding algorithm aligns with the size of the NN.

4.3.1. The Longest Common Subsequence Problem

We start with the problem of finding the length of the longest common subsequence of two finite integer sequences $x_1, \dots, x_m$ and $y_1, \dots, y_n$. A standard dynamic programming procedure, see, e.g., Cormen et al. [Cor+01, Section 15.4], computes values $f(i, j)$ equal to the length of the longest common subsequence of the partial sequences $x_1, x_2, \dots, x_i$ and $y_1, y_2, \dots, y_j$ by applying the recursion

$$f(i, j) = \begin{cases} f(i-1, j-1) + 1 & \text{if } x_i = y_j, \\ \max\{f(i-1, j), f(i, j-1)\} & \text{if } x_i \neq y_j. \end{cases} \tag{4.1}$$

Note that this dynamic program contains some if-condition. Hence, we need a trick to avoid the conditional branching in order to turn the dynamic program into a MAAP. This is established in Algorithm 3.

Proposition 4.2. *Algorithm 3 correctly returns the length of the longest common subsequence for two integer sequences* $\mathbf{x}_1, \dots, \mathbf{x}_m$ *and* $\mathbf{y}_1, \dots, \mathbf{y}_n$.

Proof. We need to verify that the computation in lines 7 and 8 is equivalent to the dynamic programming recursion (4.1). We use induction on $s = i + j$ to verify that $\mathbf{f}(i, j)$ is correctly computed. The induction start is settled by the initialization. For the induction step we distinguish two cases.

First, consider the case $\mathbf{x}_i = \mathbf{y}_i$. This implies $\mathbf{d} = 0$. Since by induction $\mathbf{f}(i-1, j-1)$ is at most by one smaller than $\mathbf{f}(i-1, j)$ or $\mathbf{f}(i, j-1)$, the maximum in line 8 is attained by the first term and the recursion is correctly computed in this case.

Otherwise, we are in the case $\mathbf{x}_i \neq \mathbf{y}_i$. By integrality, this implies $\mathbf{d} \geq 1$. Since by induction $\mathbf{f}(i-1, j-1)$ is not larger than $\mathbf{f}(i-1, j)$ or $\mathbf{f}(i, j-1)$, this implies that the maximum in line 8 is attained by one of the two latter terms and the recursion is correctly computed in this case as well. □

Having the correctness, we turn towards the complexity measures of the MAAP, which yields the following result about an NN computing the length of the longest common subsequence.

Proposition 4.3. *For given numbers m and n, there exists an NN of depth and width $\mathcal{O}(n+m)$ and size $\mathcal{O}(mn)$ mapping two integer sequences with lengths m and n to the length of the longest common subsequence.*

Proof. This follows by applying the definitions of the complexity measures d, w, and s to Algorithm 3 and using Proposition 4.1. □

We would like to remark that the NN corresponding to the MAAP in Algorithm 3 can be seen as a two-dimensional RNN in an m by n grid structure, an architecture introduced by Graves, Fernández, and Schmidhuber [GFS07]. Each basic unit of the RNN is of constant size and computes $\mathbf{f}(i, j)$ from $\mathbf{f}(i-1, j-1)$, $\mathbf{f}(i-1, j)$, $\mathbf{f}(i, j-1)$, $\mathbf{x}_i$, and $\mathbf{y}_j$.

4.3.2. The Single-Source Shortest Path Problem

As a second example, we consider the Bellman-Ford algorithm for the Single-Source Shortest Path Problem, see, for example, Kleinberg and Tardos [KT06, Section 6.8]. Let $(c_{uv})_{u,v \in V}$ be the length matrix of a graph with vertex set V and $s \in V$ is the source vertex. For simplicity, assume that $c_{vv} = 0$ for all $v \in V$ and that the graph does not contain negative cycles. The Bellman-Ford algorithm recursively computes values $f(i, v)$ determining the shortest possible length of a path from s to v using at most i arcs by

$$f(i, v) = \min_{u \in V} \{f(i-1, u) + c_{uv}\}.$$

For a fixed number of vertices $|V| = n$, this can directly translated to a MAAP, see Algorithm 4.

Proposition 4.4. *For a graph with a given number of vertices n, there exists an NN of depth $\mathcal{O}(n \log n)$, width $\mathcal{O}(n^2)$, and size $\mathcal{O}(n^3)$ mapping arc lengths to the shortest path distances from a single source to all other vertices.*

Proof. The MAAP in Algorithm 4 is a direct implementation of the Bellman-Ford algorithm and therefore correctly computes the shortest path distances. The claim follows by applying the definitions of the complexity measures d, w, and s to Algorithm 4 and using Proposition 4.1. □

Algorithm 4: MAAP to find the length of the shortest path from $s \in V$ to any other vertex in a graph with n vertices.

```
  Input: Length matrix $(\mathbf{c}_{uv})_{u,v\in V}$ of size $n$ by $n$.
  // Initialization:
1 for each $v \in V$ do parallel
2   | $\mathbf{f}(1, v) \leftarrow \mathbf{c}_{sv}$
  // Recursion:
3 for $i = 2, \dots, n-1$ do
4   | for each $v \in V$ do parallel
5   |   | $\mathbf{f}(i, v) \leftarrow \min_{u\in V}\{\mathbf{f}(i-1, u) + \mathbf{c}_{uv}\}$
6 return $(\mathbf{f}(n-1, v))_{v\in V}$.
```

Algorithm 5: MAAP to find the length of the shortest path between any two vertices in a graph with n vertices.

```
  Input: Length matrix $(\mathbf{c}_{uv})_{u,v\in V}$ of size $n$ by $n$.
  // Initialization:
1 for each $u, v \in V$ do parallel
2   | $\mathbf{d}_{uv}^{(0)} \leftarrow \mathbf{c}_{uv}$
  // Repeated Squaring:
3 for $i = 1, \dots, \lceil \log_2(n-1) \rceil$ do
4   | for each $u, v \in V$ do parallel
5   |   | $\mathbf{d}_{uv}^{(i)} \leftarrow \min_{w\in V}\{\mathbf{d}_{uw}^{(i-1)} + \mathbf{d}_{wv}^{(i-1)}\}$
6 return $(\mathbf{d}_{uv}^{(\lceil \log_2(n-1)\rceil)})_{u,v\in V}$.
```

Again, we would like to remark that the resulting NN can be implemented as an RNN, where this time one cell corresponds to one iteration of the parallel **for** loop. This cell has depth $\mathcal{O}(\log n)$ and width and size $\mathcal{O}(n^2)$. It is applied $\mathcal{O}(n)$ times to obtain the final shortest path distances.

4.3.3. The All-Pairs Shortest Path Problem

Third, recall that the All-Pairs Shortest Path Problem can be solved by computing the $(n-1)$-th min-plus matrix power of the length matrix $(c_{uv})_{u,v\in V}$, see, e.g., Leighton [Lei91, Section 2.5.4]. By repeated squaring, this can be achieved with only $\mathcal{O}(\log n)$ min-plus matrix multiplications. Again, for simplicity, let us assume that $c_{vv} = 0$ for all vertices $v \in V$ and that the graph does not contain negative cycles. The resulting procedure is realized as a MAAP in Algorithm 5.

Analyzing the MAAP with respect to our complexity measures and applying Proposition 4.1 yields the following result.

Algorithm 6: MAAP to find the length of the shortest traveling salesperson tour in a graph with n vertices.

```
  Input: Length matrix $(\mathbf{c}_{uv})_{u,v \in V}$ of size $n$ by $n$.
  // Initialization:
1 for each $v \in V \setminus \{s\}$ do parallel
2     $\mathbf{f}(\{v\}, v) \leftarrow \mathbf{c}_{sv}$
  // Recursion (can be done in parallel for sets of equal
     cardinality):
3 for $i = 2, \ldots, n-1$ do
4     for each $T \subseteq V \setminus \{s\}$ with $|T| = i$ do parallel
5         for each $v \in T$ do parallel
6             $\mathbf{f}(T, v) = \min_{u \in T \setminus \{v\}} \{\mathbf{f}(T \setminus \{v\}, u) + \mathbf{c}_{uv}\}$
7 return $\min_{u \in V \setminus \{s\}} \{\mathbf{f}(V \setminus \{s\}, u) + \mathbf{c}_{us}\}$.
```

Proposition 4.5. *For a graph with a given number of vertices n, there exists an NN of depth $\mathcal{O}(\log^2 n)$, width $\mathcal{O}(n^3)$, and size $\mathcal{O}(n^3 \log n)$ mapping arc lengths to the shortest path distances between all vertices.*

Again note that this NN can be viewed as an RNN where the RNN cell is repeatedly applied $\mathcal{O}(\log n)$ times and has depth $\mathcal{O}(\log n)$, width $\mathcal{O}(n^3)$, and size $\mathcal{O}(n^3)$.

4.3.4. Traveling Salesperson Problem

As a final example, let us consider the Bellman-Held-Karp algorithm for solving the NP-hard (asymmetric) Traveling Salesperson Problem (TSP); see Bellman [Bel62] and Held and Karp [HK62]. Given a (complete, directed) graph with vertex set V and distances c_{uv} from vertex $u \in V$ to vertex $v \in V$, the TSP asks for the shortest round-trip visiting each vertex exactly once. Choosing an arbitrary starting vertex $s \in V$, the Bellman-Held-Karp algorithm recursively computes values $f(T, v)$ for each $T \subseteq V \setminus \{s\}$, $v \in T$, corresponding to the length of the shortest s-v-path visiting exactly the nodes in $T \cup \{s\}$ by the formula

$$f(T, v) = \min_{u \in T \setminus \{v\}} \{f(T \setminus \{v\}, u) + c_{uv}\}.$$

The length of the shortest TSP tour is then given by $\min_{u \in V \setminus \{s\}} \{f(V \setminus \{s\}, u) + c_{us}\}$.

We provide a MAAP for executing this dynamic program in Algorithm 6. Analyzing the MAAP with respect to our complexity measures and applying Proposition 4.1 yields the following result.

Proposition 4.6. *For a graph with a given number of vertices n, there exists an NN of depth $\mathcal{O}(n \log n)$ and width and size $\mathcal{O}(n^2 2^n)$ mapping arc lengths to the length of the shortest TSP tour.*

In particular, a polynomially deep NN suffices to solve the NP-hard (asymmetric) TSP, while the total size is still in the order of the Bellman-Held-Karp algorithm.

4.4. The Minimum Spanning Tree Problem

After seeing a couple of examples where it was more or less straightforward to obtain MAAPs, and hence NNs, from classical algorithms, let us now consider the first non-trivial case: the Minimum Spanning Tree (MST) Problem.

A spanning tree in an undirected graph is a set of edges that is connected, spans all vertices, and does not contain any cycle. For given edge weights, the MST Problem is to find a spanning tree with the least possible total edge weight.

Most common algorithms, for example Kruskal's or Prim's algorithm, compare the edge weights and use conditional branchings to determine the order in which edges are included in the solution. Therefore, they cannot be written as a MAAP or implemented as an NN. Nevertheless, by "tropicalizing" a result by Fomin, Grigoriev, and Koshevoy [FGK16] from arithmetic circuit complexity, we obtain a recursive MAAP to compute the weight of an MST from all edge weights.

To be more precise, Fomin, Grigoriev, and Koshevoy [FGK16] provide a construction of a polynomial-size *subtraction-free arithmetic circuit* (with standard addition, multiplication, and division, but without subtractions) to compute the so-called *spanning tree polynomial* of a graph (V, E), that is, the polynomial

$$\sum_{T \text{ spanning tree}} \prod_{e \in T} x_e$$

defined over $|E|$ many variables x_e associated with the edges of the graph.

Tropicalizing this polynomial (to min-plus algebra) results precisely in the tropical polynomial mapping edge weights to the value of an MST:

$$\min_{T \text{ spanning tree}} \sum_{e \in T} x_e.$$

In the same way, one can tropicalize the arithmetic circuit provided by Fomin, Grigoriev, and Koshevoy [FGK16]. In fact, every sum gate is just replaced with a small NN computing the minimum of its inputs according to Proposition 2.2, every product with a summation, and every division with a subtraction (realized using negative weights). That way, we obtain a polynomial-size NN to compute the value of an MST from any given edge weights. Note that it is crucial that the circuit is *subtraction-free* because there is no inverse with respect to tropical addition.

While this argument is already sufficient to justify the existence of polynomial-size NNs to compute the value of an MST, we now give a completely combinatorial proof of this argument. The resulting NN, however, is basically equivalent to the NN obtained by tropicalizing the circuit of Fomin, Grigoriev, and Koshevoy [FGK16].

The rough idea of our proof is to use a recursive MAAP, which then translates to an NN of the required size. Let us mention that the use of a recursion within the statement of the MAAP is just a matter of notation and unrolling the recursion results in a MAAP family without recursions matching the way we defined MAAPs before. In each step of the recursion, one node of the graph is deleted and all remaining edge weights are updated in such a way that the objective value of the minimum spanning tree problem in the original graph can be calculated from the objective value in the

Algorithm 7: MST_n: Compute the value of a minimum spanning tree for the complete graph on $n \geq 3$ vertices.

```
  Input: Edge weights $(\mathbf{x}_{ij})_{1\leq i<j\leq n}$.
1 $\mathbf{y}_n \leftarrow \min_{i\in[n-1]} \mathbf{x}_{in}$
2 for each $1 \leq i < j \leq n-1$ do parallel
3     $\mathbf{x}'_{ij} \leftarrow \min\{\mathbf{x}_{ij},\ \mathbf{x}_{in} + \mathbf{x}_{jn} - \mathbf{y}_n\}$
4 return $\mathbf{y}_n + \mathrm{MST}_{n-1}\left((\mathbf{x}'_{ij})_{1\leq i<j\leq n-1}\right)$
```

smaller graph. The recursion used in the MAAP emerges from tropicalizing the so-called *star-mesh transformations* used by Fomin, Grigoriev, and Koshevoy [FGK16] into the combinatorial world.

Without loss of generality, we restrict ourselves to complete graphs. Edges missing in the actual input graph can be represented with large weights such that they will never be included in an MST.

For $n = 2$ vertices, the MAAP simply returns the weight of the only edge of the graph. For $n \geq 3$, our MAAP is given in Algorithm 7.

Proposition 4.7. *Algorithm 7 correctly computes the value of a minimum spanning tree in the complete graph on n vertices.*

Proof. We use induction on n. The trivial case $n = 2$ settles the induction start. Now suppose that the subroutine MST_{n-1} correctly computes the value of an MST for $n-1$ vertices. We need to show that the returned value $\mathbf{y}_n + \mathrm{MST}_{n-1}\left((\mathbf{x}'_{ij})_{1\leq i<j\leq n-1}\right)$ is indeed the MST value for n vertices.

First, we show that the value computed by Algorithm 7 is not larger than the correct objective value. For this purpose, let T be the set of edges corresponding to an MST of G. By potential relabeling of the vertices, assume that $\mathbf{y}_n = \mathbf{x}_{1n}$. Note that we may assume without loss of generality that $v_1v_n \in T$: if this is not the case, adding it to T creates a cycle in T involving a second neighbor $v_i \neq v_1$ of v_n. Removing v_iv_n from T results again in a spanning tree with total weight at most the original weight.

We construct a spanning tree T' of the subgraph spanned by the first $n-1$ vertices as follows: T' contains all edges of T that are not incident with v_n. Additionally, for each $v_iv_n \in T$, except for v_1v_n, we add the edge v_1v_i to T'. It is immediate to verify that this construction results in fact in a spanning tree. We then obtain

$$\begin{aligned}\sum_{v_iv_j\in T} \mathbf{x}_{ij} &= \mathbf{x}_{1n} + \sum_{v_iv_n\in T,\ i>1} \mathbf{x}_{in} + \sum_{v_iv_j\in T,\ i,j<n} \mathbf{x}_{ij}\\ &= \mathbf{y}_n + \sum_{v_iv_n\in T,\ i>1} (\mathbf{x}_{in} + \mathbf{x}_{1n} - \mathbf{y}_n) + \sum_{v_iv_j\in T,\ i,j<n} \mathbf{x}_{ij}\\ &\geq \mathbf{y}_n + \sum_{v_iv_n\in T,\ i>1} \mathbf{x}'_{1i} + \sum_{v_iv_j\in T,\ i,j<n} \mathbf{x}'_{ij}\end{aligned}$$

$$
\begin{aligned}
&= \mathbf{y}_n + \sum_{v_i v_j \in T'} \mathbf{x}'_{ij} \\
&\geq \mathbf{y}_n + \mathrm{MST}_{n-1}\left((\mathbf{x}'_{ij})_{1 \leq i < j \leq n-1}\right).
\end{aligned}
$$

Here, the first inequality follows by the way how the values of $\mathbf{x}'$ are defined in line 3 and the second inequality follows since T' is a spanning tree of the first $n-1$ vertices and, by induction, the MAAP is correct for up to $n-1$ vertices. This completes the proof that the MAAP does not overestimate the objective value.

In order to show that the MAAP does not underestimate the true objective value, let T' be the set of edges of a minimum spanning tree of the first $n-1$ vertices with respect to the updated costs $\mathbf{x}'$. Let $E^* \subseteq T'$ be the set of edges $v_i v_j$, $1 \leq i < j \leq n-1$, in T' that satisfy $\mathbf{x}'_{ij} = \mathbf{x}_{in} + \mathbf{x}_{jn} - \mathbf{y}_n$. Note that, in particular, we have $\mathbf{x}'_{ij} = \mathbf{x}_{ij}$ for all $v_i v_j \in T' \setminus E^*$, which will become important later. We show that we may assume without loss of generality that E^* only contains edges incident with v_1. To do so, suppose there is an edge $v_i v_j \in E^*$ with $2 \leq i < j \leq n-1$. Removing that edge from T' disconnects exactly one of the two vertices v_i and v_j from v_1; say, it disconnects v_j. We then can add $v_1 v_j$ to T' and obtain another spanning tree in G'. Moreover, by the definition of the weights $\mathbf{x}'$ and the choice of v_1, we obtain $\mathbf{x}'_{1j} \leq \mathbf{x}_{1n} + \mathbf{x}_{jn} - \mathbf{y}_n \leq \mathbf{x}_{in} + \mathbf{x}_{jn} - \mathbf{y}_n = \mathbf{x}'_{ij}$. Hence, the new spanning tree is still minimal. This procedure can be repeated until every edge in E^* is incident with v_1.

Now, we construct a spanning tree T in G from T' as follows: T contains all edges of $T' \setminus E^*$. Additionally, for every $v_1 v_i \in E^*$, we add the edge $v_i v_n$ to T. Finally, we also add $v_1 v_n$ to T. Again it is immediate to verify that this construction results in fact in a spanning tree, and we obtain

$$
\begin{aligned}
\sum_{v_i v_j \in T} \mathbf{x}_{ij} &= \mathbf{x}_{1n} + \sum_{v_1 v_i \in E^*} \mathbf{x}_{in} + \sum_{v_i v_j \in T' \setminus E^*} \mathbf{x}_{ij} \\
&= \mathbf{y}_n + \sum_{v_1 v_i \in E^*} (\mathbf{x}_{in} + \mathbf{x}_{1n} - \mathbf{y}_n) + \sum_{v_i v_j \in T' \setminus E^*} \mathbf{x}_{ij} \\
&= \mathbf{y}_n + \sum_{v_1 v_i \in E^*} \mathbf{x}'_{1i} + \sum_{v_i v_j \in T' \setminus E^*} \mathbf{x}'_{ij} \\
&= \mathbf{y}_n + \sum_{v_i v_j \in T'} \mathbf{x}'_{ij} \\
&= \mathbf{y}_n + \mathrm{MST}_{n-1}\left((\mathbf{x}'_{ij})_{1 \leq i < j \leq n-1}\right).
\end{aligned}
$$

This shows that the MAAP returns precisely the value of the spanning tree T. Hence, its output is at least as large as the value of an MST, completing the second direction. □

Finally, we prove complexity bounds for the MAAP, allowing us to bound the size of the corresponding NN.

Theorem 4.8. *For a fixed graph with n vertices, there exists an NN of depth $\mathcal{O}(n \log n)$, width $\mathcal{O}(n^2)$, and size $\mathcal{O}(n^3)$ that correctly maps a vector of edge weights to the value of a minimum spanning tree.*

Proof. In Proposition 4.7, we have seen that Algorithm 7 performs the required computation. We show that $d(\mathrm{MST}_n) = \mathcal{O}(n \log n)$, $w(\mathrm{MST}_n) = \mathcal{O}(n^2)$, and $s(\mathrm{MST}_n) = \mathcal{O}(n^3)$. Then, the claim follows by Proposition 4.1.

Concerning the complexity measure d, observe that in each recursion the bottleneck is to compute the minimum in line 1. This is of logarithmic order. Since we have n recursions, it follows that $d(\mathrm{MST}_n) = \mathcal{O}(n \log n)$.

Concerning the complexity measure w, observe that the bottleneck is to compute the parallel **for** loop in line 3. This is of quadratic order, resulting in $w(\mathrm{MST}_n) = \mathcal{O}(n^2)$.

Finally, concerning the complexity measure s, the bottleneck is also the parallel **for** loop in line 3. Again, this is of quadratic order and since we have n recursions, we arrive at $s(\mathrm{MST}_n) = \mathcal{O}(n^3)$. □

4.5. The Maximum Flow Problem

Similar to what we encountered for the MST Problem, in case of the Maximum Flow Problem, all popular algorithms make use of conditional branchings. For example, whether the flow along certain arcs is increased or not, usually depends on whether certain arcs are contained in the residual network. Therefore, in order to construct a polynomial-size NN solving the Maximum Flow Problem, we design a completely new algorithm that works without conditional branchings. In contrast to our previous examples, this time, we even compute the solution itself, that is, the flow, and not only the objective value. To be more precise, we show that, given a fixed directed graph with n nodes and m edges, there exists a polynomial-size NN that computes a function of type $\mathbb{R}^m \to \mathbb{R}^m$ that maps arc capacities to a corresponding maximum flow. Note that the objective value can be computed from the solution by simply adding up all flow values of arcs leaving the source node s. Therefore, this can be done by an NN with the same asymptotic size bounds, too.

4.5.1. Preliminaries on the Maximum Flow Problem

Let $G = (V, E)$ be a directed graph with a finite node set $V = \{v_1, \dots, v_n\}$, $n \in \mathbb{N}$, a source $s = v_1$, a sink $t = v_n$, and an arc set $E \subseteq V^2 \setminus \{vv \mid v \in V\}$ in which each arc $e \in E$ is equipped with a capacity $\nu_e \geq 0$. We write $m = |E|$ for the number of arcs, δ_v^+ and δ_v^- for the sets of outgoing and incoming arcs of node v, as well as N_v^+ and N_v^- for the sets of successor and predecessor nodes of v in G, respectively. The distance $\mathrm{dist}_G(v, w)$ denotes the minimum number of arcs on any path from v to w in G.

In this setting, the *Maximum Flow Problem* consists of finding an s-t-flow $(y_e)_{e \in E}$ satisfying $0 \leq y_e \leq \nu_e$ and $\sum_{e \in \delta_v^-} y_e = \sum_{e \in \delta_v^+} y_e$ for all $v \in V \setminus \{s, t\}$ such that the *flow value* $\sum_{e \in \delta_s^+} y_e - \sum_{e \in \delta_s^-} y_e$ is maximal.

For the sake of an easier notation we assume for each arc $e = uv \in E$ that its reverse arc vu is also contained in E. This is without loss of generality because we can use capacity $\nu_e = 0$ for arcs that are not part of the original set E. In order to avoid redundancy we represent flow only in one arc direction. More precisely, with $\vec{E} = \{v_i v_j \in E \mid i < j\}$ being the set of *forward arcs*, we denote a flow by $(y_e)_{e \in \vec{E}}$. The capacity constraints therefore state that $-\nu_{vu} \leq y_{uv} \leq \nu_{uv}$. Hence, a negative flow value on a forward arc $uv \in \vec{E}$ denotes a positive flow on the corresponding *backward arc vu*.

A crucial construction for maximum flow algorithms is the *residual network*. For a given s-t-flow $(y_e)_{e\in\vec{E}}$, the *residual capacities* are defined as follows. For an arc $uv \in \vec{E}$ the *residual forward capacity* is given by $c_{uv} := \nu_{uv} - y_{uv}$ and the *residual backward capacity* by $c_{vu} := \nu_{vu} + y_{uv}$. The *residual network* consists of all directed arcs with positive residual capacity. Hence, it is given by $G^* = (V, E^*)$ with $E^* := \{e \in E \mid c_e > 0\}$.

The asymptotically fastest known maximum flow algorithm runs in $\mathcal{O}(nm)$ time for networks with n nodes and m arcs [Orl13]. However, as outlined above, classical algorithms are not applicable for direct implementation on an NN because they require conditional branchings. Nevertheless, our approach uses ideas of the famous algorithms by Edmonds and Karp [EK72] and Dinic [Din70].

4.5.2. A MAAP to solve the Maximum Flow Problem

The rough idea of our MAAP, given in Algorithm 8 and Algorithm 9, is as follows. We start with the zero flow and consider the residual network G^*. In each iteration we augment flow along arcs of the residual network that lie on a shortest path from the source s to the sink t until they become disconnected in G^*. However, in contrast to the classical algorithms, finding an appropriate augmenting flow is much more technically involved, due to the limited set of operations allowed. This is accomplished by the FindAugmentingFlow$_k$ subroutine, which we define and analyze next. Its key feature is to return a flow that has positive flow values only on arcs that lie on a path of length k from s to t in the current residual network. Moreover, if such a path still exists, at least one arc of the residual network is saturated by that flow. Of course, all this needs to happen without explicitly knowing the arcs contained in the current residual network since this would involve conditional branchings.

4.5.3. FindAugmentingFlow$_k$ Subroutine

The key component of our MAAP to solve the Maximum Flow Problem is the subroutine, given by Algorithm 9, that returns an augmenting flow using only paths of at most a fixed length k and, in addition, saturates at least one arc of the residual network.

The first step of this subroutine is to determine for each node v and each $i \in \mathbb{N}_0$ the maximal flow value $\mathbf{a}_{i,v}$ that can be sent from v to t on a single path of length exactly i in the residual network. We call such a path a *fattest path* of length i from v to t. The value $\mathbf{a}_{k,s}$ of a fattest path from s to t of length k is of particular interest as the algorithm greedily pushes this value from s to t within k iterations. This means that, at each node v, flow is pushed into arcs $vw \in \delta_v^+$ in the order given by the node indices of the successors w. Here, the flow value that is pushed into a node w should not exceed $\mathbf{a}_{i,w}$, where i is the current iteration. This way, all flow that is pushed into a node could in principle reach t within i steps. However, it can happen that not all the flow that is pushed into a node v can be pushed out of v since the next nodes might be loaded with flow already. Therefore, the inflow at some nodes might be larger than the outflow after the pushing procedure; see Figure 4.1 for an example where this happens.

In order to restore flow conservation, the algorithm performs a clean-up in which it iterates over the nodes in reverse order and reduces the incoming flow by the remaining excess flow at node v. Hence, we obtain a feasible augmenting s-t-flow in the end. In

Algorithm 8: Computing a maximum flow for a fixed graph $G = (V, E)$.

```
Input: Capacities $(\boldsymbol{\nu}_e)_{e \in E}$.
// Initializing:
1  for each $uv \in \vec{E}$ do parallel
2      $\mathbf{x}_{uv} \leftarrow 0$  // flow; negative value correspond to flow on $vu$
3      $\mathbf{c}_{uv} \leftarrow \boldsymbol{\nu}_{uv}$ // residual forward capacities
4      $\mathbf{c}_{vu} \leftarrow \boldsymbol{\nu}_{vu}$ // residual backward capacities
// Main part:
5  for $k = 1, \dots, n-1$ do
6      for $i = 1, \dots, m$ do
7          $(\mathbf{y}_e)_{e \in \vec{E}} \leftarrow$ FindAugmentingFlow$_k((\mathbf{c}_e)_{e \in E})$
           /* Returns an augmenting flow (respecting the residual
              capacities) that only uses paths of length exactly $k$ and
              saturates at least one arc.                              */
           // Augmenting:
8          for each $uv \in \vec{E}$ do parallel
9              $\mathbf{x}_{uv} \leftarrow \mathbf{x}_{uv} + \mathbf{y}_{uv}$
10             $\mathbf{c}_{uv} \leftarrow \mathbf{c}_{uv} - \mathbf{y}_{uv}$
11             $\mathbf{c}_{vu} \leftarrow \mathbf{c}_{vu} + \mathbf{y}_{uv}$
12 return $(\mathbf{x}_e)_{e \in \vec{E}}$
```

Algorithm 9, we give a formal description of this subroutine and an example is given in Figure 4.1.

The following theorem states that the subroutine indeed computes an augmenting flow fulfilling all required properties.

Theorem 4.9. *Let $(\mathbf{c}_e)_{e \in E}$ be residual capacities such that the distance between s and t in the residual network $G^* = (V, E^*)$ is at least k. Then the MAAP given in Algorithm 9 returns an s-t-flow $\mathbf{y} = (\mathbf{y}_{uv})_{uv \in \vec{E}}$ with $-\mathbf{c}_{vu} \leq \mathbf{y}_{uv} \leq \mathbf{c}_{uv}$ such that we have positive flow only on arcs that lie on an s-t-path of length exactly k in G^*. If the distance of s and t in the residual network is exactly k, then $\mathbf{y}$ has a strictly positive flow value and there exists at least one saturated arc, i.e., one arc $e \in E^*$ with $\mathbf{y}_e = \mathbf{c}_e$.*

The proof idea is as follows. For the flow conservation we first show that the $\mathbf{Y}_u$ variables do indeed track the excessive flow at node u. More precisely, we show that after the pushing procedure we have $\sum_{e \in \delta_u^-} \mathbf{z}_e - \sum_{e \in \delta_u^+} \mathbf{z}_e = \mathbf{Y}_u^{k - \mathrm{dist}_{G^*}(s,u)}$ and $\mathbf{Y}_u^i = 0$ for all $i \neq k - \mathrm{dist}_{G^*}(s,u)$. The clean-up reduces the excessive flow to zero, i.e., in the end it holds that $\mathbf{Y}_u^i = 0$ for all i including $i = k - \mathrm{dist}_{G^*}(s,u)$.

In order to show that at least one residual arc is saturated we consider a node v^* that has positive excess flow after the pushing phase. Among these, v^* is chosen as one of the closest nodes to t in G^*. From all shortest v^*-t-paths in G^* we pick the path P that has lexicographically the smallest string of node indices. As the fattest path from v^* to t has at least the residual capacity of P (given by the minimal residual capacity of all arcs along P), the pushing procedure has pushed at least this value along P. Hence,

Algorithm 9: FindAugmentingFlow$_k$ subroutine for a fixed graph $G = (V, E)$ and a fixed length k.

```
    Input: Residual capacities $(\mathbf{c}_e)_{e\in E}$.
    // Initializing:
 1  for each $vw \in \vec{E}$ do parallel
 2      $\mathbf{z}_{vw} \leftarrow 0$   // flow in residual network
 3      $\mathbf{z}_{wv} \leftarrow 0$
 4  for each $(i, v) \in [k] \times (V \setminus \{t\})$ do parallel
 5      $\mathbf{Y}^i_v \leftarrow 0$ // excessive flow at $v$ in iteration $i$ (from $k$ to 1)
 6      $\mathbf{a}_{i,v} \leftarrow 0$ // initialize fattest path values
    // Determining the fattest path values:
 7  for each $v \in N^-_t$ do parallel
 8      $\mathbf{a}_{1,v} \leftarrow \mathbf{c}_{vt}$
 9  for $i = 2, 3, \dots, k$ do
10      for each $v \in V \setminus \{t\}$ do parallel
11          $\mathbf{a}_{i,v} \leftarrow \max_{w\in N^+_v\setminus\{t\}} \min\{\mathbf{a}_{i-1,w}, \mathbf{c}_{vw}\}$
    // Pushing flow of value $\mathbf{a}_{k,s}$ from $s$ to $t$:
12  $\mathbf{Y}^k_s \leftarrow \mathbf{a}_{k,s}$ // excessive flow at $s$
13  for $i = k, k-1, \dots, 2$ do
14      for $v \in V \setminus \{t\}$ in index order do
15          for $w \in N^+_v \setminus \{t\}$ in index order do
                // Push flow out of $v$ and into $w$:
16              $\mathbf{f} \leftarrow \min\{\mathbf{Y}^i_v, \mathbf{c}_{vw}, \mathbf{a}_{i-1,w} - \mathbf{Y}^{i-1}_w\}$ // value we can push over $vw$
                    such that this flow can still arrive at $t$
17              $\mathbf{z}_{vw} \leftarrow \mathbf{z}_{vw} + \mathbf{f}$
18              $\mathbf{Y}^i_v \leftarrow \mathbf{Y}^i_v - \mathbf{f}$
19              $\mathbf{Y}^{i-1}_w \leftarrow \mathbf{Y}^{i-1}_w + \mathbf{f}$
20  for each $v \in N^-_t$ do parallel
        // Push flow out of $v$ and into $t$:
21      $\mathbf{z}_{vt} \leftarrow \mathbf{Y}^1_v$
22      $\mathbf{Y}^1_v \leftarrow 0$
    // Clean-up by bounding:
23  for $i = 2, 3, \dots, k-1$ do
24      for $w \in V \setminus \{t\}$ in reverse index order do
25          for $v \in N^-_w \setminus \{t\}$ in reverse index order do
26              $\mathbf{b} \leftarrow \min\{\mathbf{Y}^i_w, \mathbf{z}_{vw}\}$ // value we can push backwards along $vw$
27              $\mathbf{z}_{vw} \leftarrow \mathbf{z}_{vw} - \mathbf{b}$
28              $\mathbf{Y}^i_w \leftarrow \mathbf{Y}^i_w - \mathbf{b}$
29              $\mathbf{Y}^{i+1}_v \leftarrow \mathbf{Y}^{i+1}_v + \mathbf{b}$
30  for each $uv \in \vec{E}$ do parallel
31      $\mathbf{y}_{vw} \leftarrow \mathbf{z}_{vw} - \mathbf{z}_{wv}$
32  return $(\mathbf{y}_e)_{e\in\vec{E}}$
```

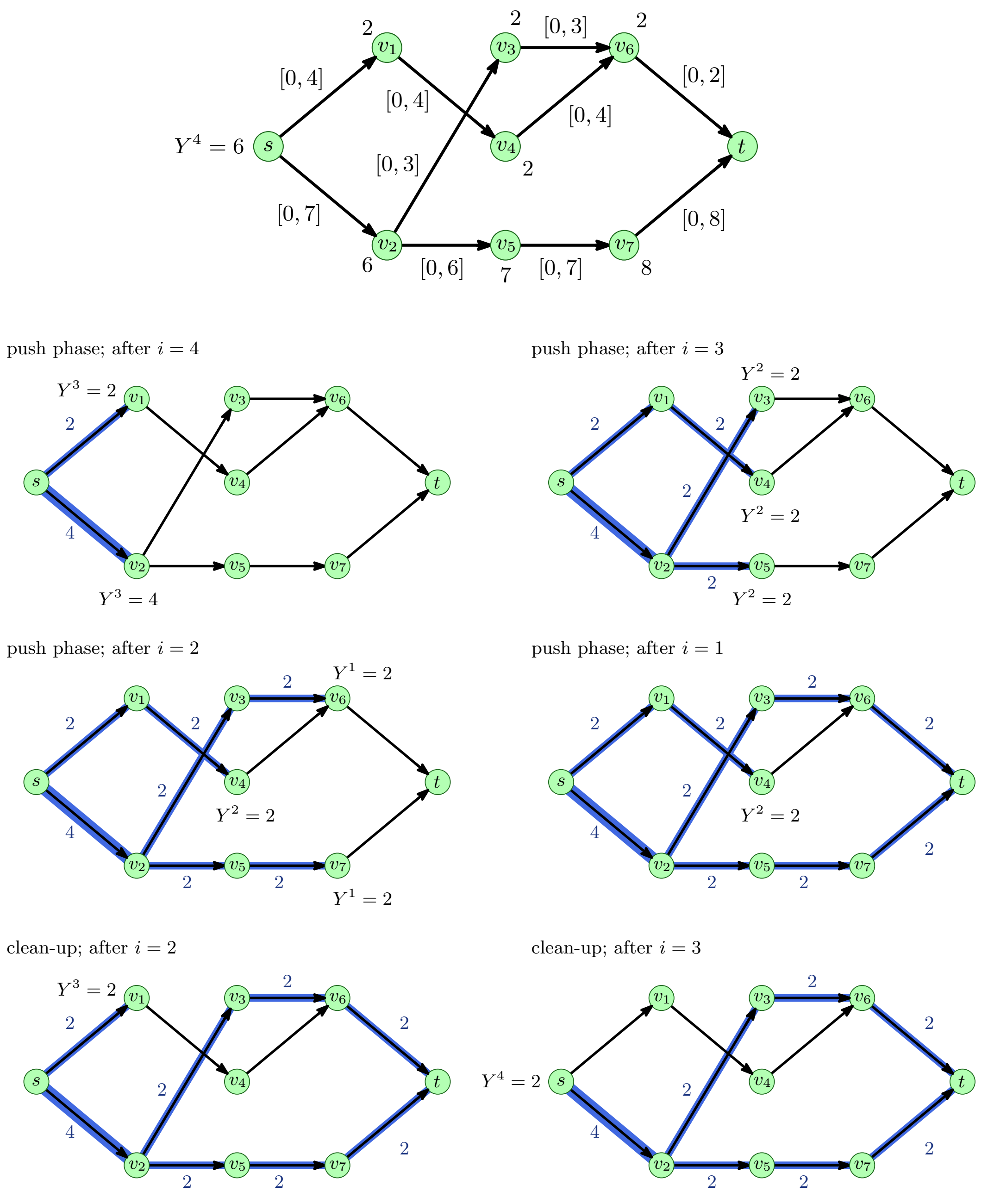


Figure 4.1.: Example of the FindAugmentingFlow Subroutine. The network on the top depicts the residual capacity bounds $[-\mathbf{c}_{wv}, \mathbf{c}_{vw}]$ for all arcs $vw \in \vec{E}$ and the node labels $\mathbf{a}_{i,v}$ for the fattest path from v to t within i steps. The four figures in the middle show the states of the flow $\mathbf{z}_{vw}$ and the excessive flow $\mathbf{Y}_v^i$ at the end of a push iteration. The bottom two figures depict the states after the clean-up iterations. All values that are not displayed are zero. Observe that the result is an s-t-flow that is feasible with respect to the residual capacities, uses only paths of length $k = 4$, and saturates the arc v_6t.

the arc on P with minimal capacity has to be saturated. It is then easy to show that the clean-up does not reduce the value along P.

Keeping this idea in mind, we now provide the full proof with all details.

Proof of Theorem 4.9. It is easy to check that lines 7 to 11 from the algorithm do indeed compute the maximal flow value $\mathbf{a}_{i,v}$ that can be send from v to t along a single path (which we call the *fattest path*) of length exactly i.

In the following we show that in line 31 the arc vector $\mathbf{z} = (\mathbf{z}_e)_{e\in E}$ forms an s-t-flow in the residual network $G^* = (V, E^*)$ that satisfies $0 \leq \mathbf{z}_e \leq \mathbf{c}_e$. For this, recall that $\mathrm{dist}_{G^*}(s,u)$ denotes the distance from s to u in the residual network.

In order to prove flow conservation of $\mathbf{z}$ at all vertices except for s and t, we fix some node $u \in V \setminus \{t\}$ and show that

$$Y_u^j = \begin{cases} \sum_{e\in\delta_u^-} \mathbf{z}_e - \sum_{e\in\delta_u^+} \mathbf{z}_e & \text{if } j = k - \mathrm{dist}_{G^*}(s,u), \\ 0 & \text{otherwise,} \end{cases} \tag{4.2}$$

holds throughout the execution of the subroutine.

Claim 4.10. *Equation* (4.2) *holds after the pushing procedure (lines 12 to 22).*

Proof of Claim 4.10. For $j < k - \mathrm{dist}_{G^*}(s,u)$, there does not exist any u-t-path of length j (since $j + \mathrm{dist}_{G^*}(s,u) < k \leq \mathrm{dist}_{G^*}(s,t)$). Hence, the fattest u-t-path of length exactly j has capacity $\mathbf{a}_{j,u} = 0$. In any iteration that might increase $\mathbf{Y}_u^j$, that is, for $i = j+1$, $v \in V \setminus \{u,t\}$, and $w = u$, we have $\mathbf{f} = 0$. This implies that $\mathbf{Y}_u^j$ remains 0.

For $j > k - \mathrm{dist}_{G^*}(s,u)$, there is no s-u-path of length $k-j$ in the residual graph as $k - j < \mathrm{dist}_{G^*}(s,u)$. The statement $\mathbf{Y}_u^j = 0$ then follows by an induction on $\mathrm{dist}_{G^*}(s,u)$, as we show in the following:

For the base case of $\mathrm{dist}_{G^*}(s,u) = 1$ we have that $\mathbf{Y}_u^j = \mathbf{Y}_u^k = 0$ since for $u \neq s$ it holds that $\mathbf{Y}_u^k$ stays 0 during the whole algorithm.

The induction step follows because for iteration $i = j+1$, $v \in V \setminus \{u,t\}$ and $w = u$ it holds that either $\mathbf{c}_{vw} = 0$ (i.e., arc vw is not part of the residual network) or $\mathbf{Y}_v^i = 0$ (inductively as $\mathrm{dist}_{G^*}(s,v) \geq \mathrm{dist}_{G^*}(s,u) - 1$ and $i = j+1 > k - \mathrm{dist}_{G^*}(s,u) + 1 \geq k - \mathrm{dist}_{G^*}(s,v)$). Either way we have $\mathbf{f} = 0$ implying that $\mathbf{Y}_u^j = \mathbf{Y}_w^{i-1}$ stays 0.

In conclusion, we obtain that $\mathbf{Y}_u^j$ can only be non-zero for $j = k - \mathrm{dist}_{G^*}(s,u)$. In each iteration with $i = k - \mathrm{dist}_{G^*}(s,u) + 1$ and $w = u$ we add $\mathbf{f}$ to the flow value $\mathbf{z}_{vu}$ and the same to $\mathbf{Y}_u^{k-\mathrm{dist}_{G^*}(s,u)}$ and in each iteration with $i = k - \mathrm{dist}_{G^*}(s,u)$ and $v = u$ we add $\mathbf{f}$ to the flow value $\mathbf{z}_{uw}$ and subtract $\mathbf{f}$ from $\mathbf{Y}_u^{k-\mathrm{dist}_{G^*}(s,u)}$. Hence, $\mathbf{Y}_u^{k-\mathrm{dist}_{G^*}(s,u)}$ denotes exactly the excessive flow after the pushing procedure as stated in (4.2). ■

This claim already shows that $\mathbf{z}_e$ can only be positive if e lies on an s-t-path of length exactly k, which is a shortest path in the residual network. To see this, let vw be an arc that is not on such a path. In line 16, it either holds that $\mathbf{Y}_v^i = 0$ (if $i \neq k - \mathrm{dist}_{G^*}(s,u)$) or $\mathbf{a}_{i-1,w} = 0$ because for $i = k - \mathrm{dist}_{G^*}(s,u)$ there is no w-t-path of length $i - 1 = k - \mathrm{dist}_{G^*}(s,u) - 1$ (since otherwise vw would lie on an s-t-path of length k). Thus, $\mathbf{z}_{vw}$ will never be increased. As the clean-up only reduces the flow values, $\mathbf{z}_{vw}$ will still be 0 at the end (line 31).

Claim 4.11. *Equation* (4.2) *holds in each iteration of the clean-up (lines 23 to 29).*

Proof of Claim 4.11. First, we show that $\mathbf{Y}_u^j$ stays 0 for $j \neq k - \text{dist}_{G^*}(s,u)$ by induction over $j = 2, 3, \ldots, k$. The base case follows immediately as we only subtract $\mathbf{b} \geq 0$ from $\mathbf{Y}_u^2$. For the induction step we have to show that $\mathbf{b} = 0$ whenever we add $\mathbf{b}$ to $\mathbf{Y}_u^j$ in line 29. In all iterations with $i = j-1$ and $v = u$ we either have $\mathbf{z}_{uw} = 0$ or $\mathbf{Y}_w^i = 0$. The reason for this is that $\mathbf{z}_{uw} > 0$ implies that uw lies on a shortest s-t-path, which means that $\text{dist}_{G^*}(s,w) = \text{dist}_{G^*}(s,u) + 1$, and hence, $i = j - 1 \neq k - \text{dist}_{G^*}(s,u) - 1 = k - \text{dist}_{G^*}(s,w)$. By induction this means that $\mathbf{Y}_w^i = 0$. Either way this implies $\mathbf{b} = 0$.

Equation (4.2) holds for $j = k - \text{dist}_{G^*}(s,u)$ since for $e \in \delta_u^-$ the value $\mathbf{b}$ is only possibly positive for $i = k - \text{dist}_{G^*}(s,u)$ and then it is subtracted from $\mathbf{z}_e$ as well as from $\mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)}$. For $e \in \delta_u^+$, the value $\mathbf{b}$ can only be positive for $i = k - \text{dist}_{G^*}(s,u) + 1$, and hence, $\mathbf{b}$ is subtracted from $\mathbf{z}_e$ exactly when it is added to $\mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)}$. ■

Next, we show that at the end of the subroutine it holds that $\mathbf{Y}_u^j = 0$ for all j, in particular also for $j = k - \text{dist}_{G^*}(s,u)$. The only exception of this is $\mathbf{Y}_s^k$. To see this, first observe that during the clean-up, $\mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)}$ is maximal after iteration $i = k - \text{dist}_{G^*}(s,u) - 1$ and does not increase anymore for $i \geq k - \text{dist}_{G^*}(s,u)$. At the start of iteration $i = k - \text{dist}_{G^*}(s,u)$ it holds due to (4.2) that

$$\sum_{e \in \delta_u^-} \mathbf{z}_e \geq \mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)}.$$

Hence, for $i = k - \text{dist}_{G^*}(s,u)$ and $w = u$ there is one iteration for all $e \in \delta_u^-$ and within this iteration $\mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)}$ is reduced by $\mathbf{z}_e$ until $\mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)} = 0$. This shows that after all iterations with $i = k - \text{dist}_{G^*}(s,u)$ it holds that $\mathbf{Y}_u^{k-\text{dist}_{G^*}(s,u)} = 0$. Together with (4.2), this immediately implies flow conservation of $(\mathbf{z}_e)_{e \in E}$.

Finally, in order to show that $0 \leq \mathbf{z}_e \leq \mathbf{c}_e$, note that $\mathbf{z}_e$ is initialized with 0 and it is only increased in line 17 of the unique iteration with $vw = e$ and $i = k - \text{dist}_{G^*}(s,v)$, as we have argued in the proof of Claim 4.10. In this iteration we have that $\mathbf{f} \leq \mathbf{c}_e$, which immediately shows that $0 \leq \mathbf{z}_e \leq \mathbf{c}_e$.

It only remains to show that at least one residual arc is saturated. To this end, suppose that the distance of s and t in G^* is k, which means that there exists at least one s-t-path of length exactly k with a strictly positive residual capacity on all arc along this path.

Let us consider the set $\{(v,i) \mid \mathbf{Y}_v^i > 0 \text{ after the pushing procedure}\}$. These are all nodes that need to be cleaned up in order to restore flow conservation, paired with their distance to t. Let (v^*, i^*) be a tuple of this set such that i^* is minimal. In other words, v^* is a node that is closest to t among theses nodes. We manually set (v^*, i^*) to (s, k) in the case that the set is empty.

Claim 4.12. *Some arc on a shortest path from v^* to t in G^* is saturated by $\mathbf{y}_e$.*

Proof of Claim 4.12. Among all these paths between v^* and t of length i^* we consider the path P which has lexicographically the smallest string of node indices. Let $\mathbf{c}_{\min}$ be the minimal residual capacity along this path P.

For all nodes v along P (including v^*) we have $\mathbf{a}_{i,v} \geq \mathbf{c}_{\min}$, where i is the distance from v to t along P, since the fattest path from v to t has to have at least the residual capacity of P.

After the pushing procedure it holds that $\mathbf{z}_e \geq \mathbf{c}_{\min}$ for all $e \in P$. This is true for the first arc on P, since we have excess flow at node v^* remaining (after pushing), hence, we certainly pushed at least $\mathbf{c}_{\min} \leq \mathbf{a}_{i^*-1,w}$ into the first arc v^*w of P. (This is also true if $v^* = s$.) For the remaining arcs of P it is true, because by the lexicographical minimality of P, the algorithm always pushes a flow value that is greater or equal to $\mathbf{c}_{\min}$ first along the next arc on P.

During the clean-up, this property remains valid as we only reduce flow on arcs that have a distance of more than i^* from t.

Hence, an arc $e \in P$ with $\mathbf{c}_e = \mathbf{c}_{\min}$ is saturated at the very end of the subroutine. ■

In conclusion, $\mathbf{y}$ is a feasible s-t-flow in the residual network that has positive value only on paths of length k and saturates at least one arc. This finalizes the proof of Theorem 4.9. □

4.5.4. Correctness of the Main Routine

The following theorem states that Algorithm 8 correctly computes a maximum flow. Using the correctness of the technically involved subroutine `FindAugmentingFlow`$_k$, the remaining proof is actually similar to textbook proofs for the algorithms by Edmonds-Karp and Dinic; see, e.g., Korte and Vygen [KV08].

Theorem 4.13. *Let $G = (V, E)$ be a fixed directed graph with $s, t \in V$. For capacities $(\boldsymbol{\nu}_e)_{e \in E}$ as input, the MAAP given by Algorithm 8 returns a maximum s-t-flow $(\mathbf{x}_e)_{e \in \vec{E}}$.*

Proof. It is a well-known fact that a feasible s-t-flow is maximum if and only if the corresponding residual network does not contain any s-t-path, see, e.g., Korte and Vygen [KV08, Theorem 8.5]. Since any simple path has length at most $n-1$, it suffices to show the following claim.

Claim 4.14. *After iteration k of the* **for** *loop in line 5 of Algorithm 8, $\mathbf{x}$ is a feasible s-t-flow with corresponding residual capacities $\mathbf{c}$ such that no s-t-path of length at most k remains in the residual network.*

Given a residual network (V, E^*), let E^*_k be the set of arcs that lie on an s-t-path of length exactly k. If the distance from s to t is exactly k, then these arcs coincide with the arcs of the so-called *level graph* used in Dinic's algorithm, compare [Din70; KV08].

We will show Claim 4.14 about the outer **for** loop by induction on k using a similar claim about the inner **for** loop.

Claim 4.15. *Suppose, at the beginning of an iteration of the* **for** *loop in line 6, it holds that*

(i) $\mathbf{x}$ is a feasible s-t-flow with corresponding residual capacities $\mathbf{c}$, and

(ii) the length of the shortest s-t-path in the residual network is at least k.

Then, after that iteration, properties (i) *and* (ii) *do still hold. Moreover, if E^*_k is nonempty, then its cardinality is strictly reduced by that iteration.*

Proof of Claim 4.15. Since (i) and (ii) hold at the beginning of the iteration, Theorem 4.9 implies that the flow $\mathbf{y}$ found in line 7 fulfills flow conservation and is bounded by $-\mathbf{c}_{vu} \leq \mathbf{y}_{uv} \leq \mathbf{c}_{uv}$ for each $uv \in \vec{E}$. Hence, we obtain that, after updating $\mathbf{x}$ and $\mathbf{c}$ in lines 8 to 11, $\mathbf{x}$ is still a feasible flow that respects flow conservation and capacities, and **c** are the corresponding new residual capacities. Thus, property (i) is also true at the end of the iteration.

Let $G^* = (V, E^*)$ and $\tilde{G}^* = (V, \tilde{E}^*)$ be the residual graphs before and after the iteration, respectively. Let E^*_k and $\tilde{E}^*_k$ be the set of arcs on s-t-paths of length k in G^* and $\tilde{G}^*$, respectively. Finally, let E' be the union of E^* with the reverse arcs of E^*_k and let $G' = (V, E')$.

Since, by Theorem 4.9, we only augment along arcs in E^*_k, it follows that $\tilde{E}^* \subseteq E'$. Let P be a shortest s-t-path in G' and suppose for contradiction that P contains an arc that is not in E^*. Let $e = uv$ be the first of all such arcs of P and let P_u be the subpath of P until node u. Then the reverse arc vu must be in E^*_k. In particular, $\text{dist}_{G^*}(s, v) < \text{dist}_{G^*}(s, u) \leq |E(P_u)|$, where the second inequality follows because P_u uses only arcs in E^*. Hence, replacing the part of P from s to v by a shortest s-v-path in G^* reduces the length of P by at least two, contradicting that P is a shortest path in G'.

Thus, all shortest paths in G' only contain arcs from E^*. In particular, they have length at least k. Hence, all paths in G' that contain an arc that is not in E^* have length larger than k. Since $\tilde{E}^* \subseteq E'$, this also holds for paths in $\tilde{G}^*$, which implies (ii). It also implies that $\tilde{E}^*_k \subseteq E^*_k$. Moreover, by Theorem 4.9, if E^*_k is nonempty, at least one arc of E^*_k is saturated during the iteration, and thus removed from E^*_k. Thus, the cardinality of E^*_k becomes strictly smaller. ■

Using Claim 4.15, we are now able to show Claim 4.14.

Proof of Claim 4.14. We use induction on k. For the induction start, note that before entering the **for** loop in line 5, that is, so to speak, after iteration 0, obviously no s-t-path of length 0 can exist in the residual network. Also note that after the initialization in lines 1 to 4, **x** is the zero flow, which is obviously feasible, and **c** contains the corresponding residual capacities.

For the induction step, consider the k-th iteration. By the induction hypothesis, we know that, at the beginning of the k-th iteration, **x** is a feasible s-t-flow with corresponding residual capacities **c** and the distance from s to t in the residual network is at least k. Observe that by Claim 4.15, these properties are maintained throughout the entire k-th iteration. In addition, observe that at the beginning of the k-th iteration, we have $|E^*_k| \leq m$. Since, due to Claim 4.15, $|E^*_k|$ strictly decreases with each inner iteration until it is zero, it follows that after the m inner iterations, the residual network does not contain an s-t-path of length k any more, which completes the induction. ■

Since any simple path has length at most $n - 1$, Claim 4.14 implies that, at the end of iteration $k = n - 1$, the nodes s and t must be disconnected in the residual network. Hence, Algorithm 8 returns a maximum flow, which concludes the proof of Theorem 4.13. □

4.5.5. Bounding the Complexity

By applying the definition of our complexity measures, we obtain the following bounds.

Theorem 4.16. *The MAAP A defined by Algorithm 8 fulfills $d(A), s(A) \in \mathcal{O}(n^2m^2)$ as well as $w(A) \in \mathcal{O}(n^2)$.*

Proof. We first analyze the MAAP A' given in Algorithm 9. Concerning the complexity measures d and s, the bottleneck of Algorithm 9 is given by the two blocks consisting of lines 13 to 19 as well as lines 23 to 29. Each of these blocks has $\mathcal{O}(km)$ sequential iterations and the body of the innermost **for** loop has constant complexity. Thus, we have $d(A'), s(A') \in \mathcal{O}(km) \subseteq \mathcal{O}(nm)$ for the overall subroutine.

Concerning measure w, the bottleneck is in fact the initialization in lines 1 to 6, such that we have $w(A') \in \mathcal{O}(m + kn) \subseteq \mathcal{O}(n^2)$.

Now consider the main routine in Algorithm 8. For all three complexity measures, the bottleneck is the call of the subroutine in line 7 within the two **for** loops. Since we have a total of $\mathcal{O}(nm)$ sequential iterations, the claimed complexity measures follow. Note that the parallel **for** loops in lines 1 and 8 do not increase the measure $w(A) \in \mathcal{O}(n^2)$. □

Let us give two remarks about the complexity of the MAAP in Algorithm 8.

First, observe that the total computational work that is carried out by the MAAP, represented by $s(A)$, differs only by a factor of n from the standard running time bound $\mathcal{O}(nm^2)$ of the Edmonds-Karp algorithm; see [KV08, Corollary 8.15]. While the number of augmenting steps is in $\mathcal{O}(nm)$ for both algorithms, the difference lies in finding the augmenting flow. While the Edmonds-Karp algorithm finds the shortest path in the residual network in $\mathcal{O}(m)$ time, the subroutine in Algorithm 9 requires $\mathcal{O}(mn)$ computational work.

The second remark is that the reported complexity $w(A) \in \mathcal{O}(n^2)$ in Theorem 4.16 is actually suboptimal and can be replaced by $\mathcal{O}(1)$ instead, if the parallel **for** loops in the MAAP are replaced with sequential ones. The asymptotics of $d(A)$ and $s(A)$ remain unchanged because the bottleneck parts of the MAAP are already of sequential nature. Still, we used parallel **for** loops in Algorithm 8 and Algorithm 9 in order to point out at which points the ability of NNs to parallelize can be used in a straightforward way.

Combining the previous observations with Proposition 4.1 we obtain the following corollary.

Corollary 4.17. *Let $G = (V, E)$ be a fixed directed graph with $s, t \in V$, $|V| = n$, and $|E| = m$. There exists an NN of depth and size $\mathcal{O}(m^2n^2)$ and width $\mathcal{O}(1)$ that correctly maps arc capacities $(\boldsymbol{\nu}_e)_{e\in E}$ to a maximum s-t-flow $(\mathbf{x}_e)_{e\in\vec{E}}$.*

4.6. The Knapsack Problem

In the previous two sections we have seen two examples of efficiently solvable problems for which it was non-trivial to obtain NNs of polynomial size. In this section, we go one step further and consider the NP-hard Knapsack Problem. Of course, we cannot expect polynomial-size NNs that provide exact solutions. Instead, we strive for pseudo-polynomial size in the exact case and smaller NNs providing solutions with provable approximation guarantees. We also provide a small computational study supporting our theoretical size bounds.

On the methodological side, in this section, we go back to the classical paradigm of dynamic programming, which has also been the key tool to obtain NNs in a straightforward way in Section 4.3. However, in case of the Knapsack Problem, it is a bit more involved to turn dynamic programming recursions into NNs, as we will see.

4.6.1. Preliminaries on the Knapsack Problem

The Knapsack Problem constitutes one of the oldest and most studied problems in combinatorial optimization (CO). Given a set of items with certain profit and size values, as well as a knapsack capacity, the Knapsack Problem asks for a subset of items with maximum total profit such that the total size of the subset does not exceed the capacity.

The Knapsack Problem is one of Karp's 21 original NP-complete problems [Kar72] and has numerous applications in a wide variety of fields, ranging from production and transportation, over finance and investment, to network security and cryptography. It often appears as a subproblem at the core of more complex problems; see, e.g., Kellerer, Pferschy, and Pisinger [KPP04] and Martello and Toth [MT90]. This fact substantiates the Knapsack Problem's prominent importance as one of the key problems in CO. In particular, the Knapsack Problem is frequently being used as a testbed for measuring the progress of various exact and heuristic solution approaches and computational methods such as, e.g., integer programming, constraint programming, or evolutionary algorithms. In integer programming, for example, the Knapsack Problem and so-called 'Knapsack Inequalities' play a central role, both with respect to theory as well as in the development of modern computational methods; see, e.g., Bertsimas and Weismantel [BW05] and Fischetti and Lodi [FL10].

An instance of the Knapsack Problem consists of n items $1, 2, \dots, n$, where each item $i \in [n]$ comes with a given profit $p_i \in \mathbb{N}$ and size $s_i \in]0, 1]$, together with a Knapsack that can hold any subset $M \subseteq [n]$ of items of total size $\sum_{i \in M} s_i$ at most 1. The task is to find such a subset $M \subseteq [n]$ that maximizes the total profit $\sum_{i \in M} p_i$.

We outline a classical dynamic programming formulation for the Knapsack Problem. Let $p^* \in \mathbb{N}$ be an upper bound on the optimum solution value, e.g., $p^* = \sum_{i=1}^{n} p_i$. For $i \in [n]$ and $p \in [p^*]$, let

$$f(p,i) := \min\left\{\sum\nolimits_{j \in M} s_j \;\middle|\; M \subseteq [i],\ \sum\nolimits_{j \in M} p_j \geq p\right\}$$

be the minimum size of a subset of the first i items with total profit at least p. With $f(p,i) := 0$ for $p \leq 0$ and $f(p,0) := +\infty$ for $p \in [p^*]$, the values of f can be computed recursively by

$$f(p,i) = \min\{f(p,i-1), f(p-p_i, i-1) + s_i\} \tag{4.3}$$

for $i \in [n]$, $p \in [p^*]$, where the first option corresponds to not using the i-th item, while the second option corresponds to using it. The optimum solution value is then given by $\max\{p \in [p^*] \mid f(p,n) \leq 1\}$, and the optimum subset can easily be found by backtracking. The runtime of the dynamic program is $\mathcal{O}(np^*)$, thus pseudo-polynomial in the input size.

Due to NP-hardness of the Knapsack Problem, one cannot expect to find an exact algorithm with polynomial running time. However, by carefully downscaling and rounding the profit values in the dynamic program, one can obtain a *fully polynomial-time approximation scheme* (FPTAS). That is, for each $\varepsilon > 0$, one can compute a feasible solution with guaranteed profit of at least $1 - \varepsilon$ times the optimal profit, with running time polynomial in the input size and $1/\varepsilon$. For more details, we refer to the books by Hochbaum [Hoc97], Vazirani [Vaz01], or Williamson and Shmoys [WS11].

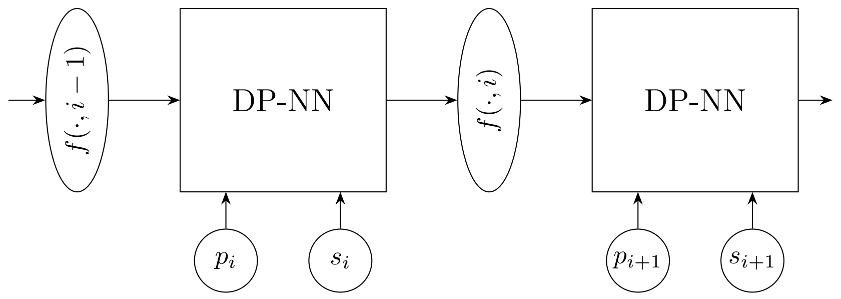


Figure 4.2.: Recurrent structure of the DP-NN to solve the Knapsack Problem.

Usually, the Knapsack Problem is defined with integer size values $s_i \in \mathbb{N}$ and some Knapsack capacity $S \in \mathbb{N}$, bounding the total size of chosen items. Dividing all item sizes by S transforms such an instance into an instance of the type considered here. For the case of integral item sizes, there is also a pseudo-polynomial dynamic programming formulation parameterized by the size instead of the profit values; see, e.g., Kleinberg and Tardos [KT06, Section 6.4]. Our construction in Section 4.6.2 can analogously be applied to this formulation. This variant, however, does not easily extend to an FPTAS. We therefore stick to the variant parametrized by the profit values as introduced above.

4.6.2. An Exact RNN for the Knapsack Problem

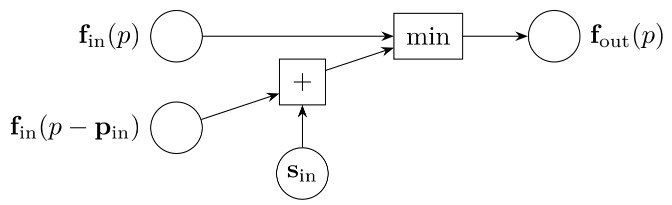
We now present the *DP-NN*, an NN of depth $\mathcal{O}(n)$ and width $\mathcal{O}((p^*)^2)$ that executes the dynamic program (4.3) to find the exact value of an optimum knapsack solution. Here, n is the number of items in the Knapsack instance, and p^* is an a priori known upper bound on the value of an optimum solution. More precisely, the DP-NN is an RNN of depth four and width $\mathcal{O}((p^*)^2)$ that is iteratively applied to the n items of a Knapsack instance. As p^* can, e.g., be chosen as the total size of all items, the DP-NN's width is pseudo-polynomially bounded in the input size of the Knapsack instance. Due to the Knapsack Problem's NP-hardness, however, there is no polynomial-size NN that always finds the optimum solution value, unless P = NP. We first explain the high-level idea of the network structure, before formally defining the DP-NN as a MAAP and proving its correctness.

High-level idea of the construction

The idea behind our construction is that the output neurons of the RNN can be seen as elements of the dynamic programming state space while the hidden neurons and the network itself implement the recursive dynamic programming formula (4.3). Here, the main technical difficulty is to always filter out the correct entries of the previous state space (input neurons) needed in the recursive formula.

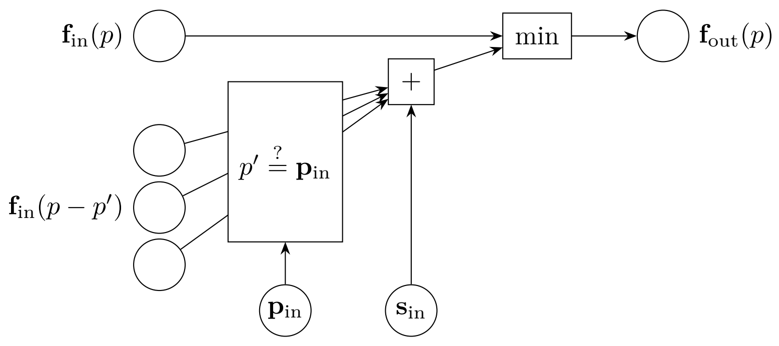
In the i-th step, the DP-NN receives $p^* + 2$ inputs, namely $f(p, i-1)$ for $p \in [p^*]$, as well as p_i and s_i. It computes p^* output values, namely $f(p, i)$ for $p \in [p^*]$. Hence, in total, it has $p^* + 2$ input neurons and p^* output neurons. Figure 4.2 illustrates the recurrent structure of the NN, which computes the state space of the dynamic program.

In order to make the recurrent structure of our NN obvious, we do not use the index i in the following description of the network. Instead, we denote the $p^* + 2$ input values



$\mathbf{f}_{\text{in}}(p)$
min
$\mathbf{f}_{\text{out}}(p)$
+
$\mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}})$
$\mathbf{s}_{\text{in}}$

Figure 4.3.: Desirable architecture for computing $\mathbf{f}_{\text{out}}(p)$, $p \in [p^*]$, from the inputs. However, the existence of an edge (nonzero weight) critically depends on the input value $\mathbf{p}_{\text{in}}$, which is not allowed.



$\mathbf{f}_{\text{in}}(p)$
min
$\mathbf{f}_{\text{out}}(p)$
+
$p' \overset{?}{=} \mathbf{p}_{\text{in}}$
$\mathbf{f}_{\text{in}}(p - p')$
$\mathbf{p}_{\text{in}}$
$\mathbf{s}_{\text{in}}$

Figure 4.4.: High-level idea how the DP-NN computes $\mathbf{f}_{\text{out}}(p)$ for $p \in [p^*]$ from the inputs.

by $\mathbf{f}_{\text{in}}(p)$ for $p \in [p^*]$, as well as $\mathbf{p}_{\text{in}}$ and $\mathbf{s}_{\text{in}}$. The p^* output values are denoted by $\mathbf{f}_{\text{out}}(p)$ for $p \in [p^*]$. The goal is to implement the recursion

$$\mathbf{f}_{\text{out}}(p) = \min\{\mathbf{f}_{\text{in}}(p),\ \mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}}) + \mathbf{s}_{\text{in}}\} \quad \text{for } p \in [p^*]$$

in an NN; cp. (4.3). It consists of an addition and taking a minimum, which are both simple operations for an NN. Hence, ideally, we would like to have an architecture as depicted in Figure 4.3 for computing $\mathbf{f}_{\text{out}}(p)$ for every $p \in [p^*]$. The problem with this is, however, that the decision which component of $\mathbf{f}_{\text{in}}$ is accessed in order to compute the sum with $\mathbf{s}_{\text{in}}$ depends on the input value $\mathbf{p}_{\text{in}}$. Since we aim for an architecture that is fixed and works for general input values $\mathbf{p}_{\text{in}}$, we have to extend our construction as depicted in Figure 4.4. As we do not know the value of $\mathbf{p}_{\text{in}}$ in advance, we connect every input neuron $\mathbf{f}_{\text{in}}(p-p')$, $p' \in [p-1]$, to the unit that computes the sum $\mathbf{f}_{\text{in}}(p-\mathbf{p}_{\text{in}})+\mathbf{s}_{\text{in}}$. Since we only want to take the value $\mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}})$ into account, we need to add an additional unit that disables those connections with $p' \neq \mathbf{p}_{\text{in}}$.

Due to the integrality of the profit values, this additional unit can be realized with two hidden layers and a constant number of neurons for every value of $p \in [p^*]$ and $p' \in [p-1]$, as we show in a moment. Computing the minimum adds a third hidden layer. Hence, the DP-NN has depth four while width and size are in $\mathcal{O}((p^*)^2)$. Unfolding the RNN and viewing it as a single feedforward NN executing the whole dynamic program results in depth $\mathcal{O}(n)$ and size $\mathcal{O}(n(p^*)^2)$. In the next subsection, we provide a detailed construction of the DP-NN as a MAAP and prove the following theorem.

Algorithm 10: MAAP corresponding to the DP-NN.

```
Input: $\mathbf{f}_{\text{in}}(p)$ for $p \in [p^*]$, as well as $\mathbf{p}_{\text{in}}$ and $\mathbf{s}_{\text{in}}$.
  // Compute helper variables to select correct input entry:
1 for each $k \in [p^*]$ do parallel
2     $\mathbf{h}(k) \leftarrow \max\{2(\mathbf{p}_{\text{in}} - k), 2(k - \mathbf{p}_{\text{in}})\}$
  // Execute recursion (4.4) for each $p \in [p^*]$ in parallel:
3 for each $p \in [p^*]$ do parallel
4     for each $k \in [p-1]$ do parallel
          // Select correct input entry (non-zero only for $k = \mathbf{p}_{\text{in}}$):
5         $\mathbf{g}(p,k) \leftarrow \max\{0, \mathbf{f}_{\text{in}}(p-k) - \mathbf{h}(k)\}$
6     $\mathbf{f}_{\text{out}}(p) \leftarrow \min\{\mathbf{f}_{\text{in}}(p), \mathbf{s}_{\text{in}} + \sum_{k=1}^{p-1} \mathbf{g}(p,k)\}$
7 return $\mathbf{f}_{\text{out}}(p)$ for $p \in [p^*]$.
```

Theorem 4.18. *With the DP-NN, there exists an NN of depth four and width and size $\mathcal{O}((p^*)^2)$ that correctly computes the dynamic programming recursion* (4.3) *for knapsack instances with capacity $S = 1$, $s_i \in]0,1]$, and $p_i \in \mathbb{N}$, for $i \in [n]$, where p^* is an upper bound on the optimal solution value.*

Observe that due to the NP-hardness of the Knapsack Problem, the dependence of the network size on p^* cannot be avoided unless P=NP, if exact results are desired.

Details of the construction and correctness

Note that for size values larger than the Knapsack capacity, which is equal to 1 by our definition, we do not really care how large they actually are. Therefore, we define $\tilde{f}(p,i) = \min\{f(p,i), 2\}$ to be the values of the dynamic program truncated at 2. In other words, we replace all values in the interval $[2, +\infty]$ with the constant 2. Note that the recursion

$$\tilde{f}(p,i) = \min\{\tilde{f}(p,i-1), \tilde{f}(p-p_i, i-1) + s_i\} \tag{4.4}$$

is still valid with starting values $\tilde{f}(p,i) = 0$ for $p \leq 0$ and $\tilde{f}(p,0) = 2$ for $p \in [p^*]$. Instead of computing the actual values of f, the DP-NN computes the values of $\tilde{f}$. A MAAP to accomplish this task is given in Algorithm 10. The DP-NN is then defined as the NN emerging from applying Proposition 4.1 to Algorithm 10.

Our next goal is to prove Theorem 4.18, that is, correctness of the DP-NN.

Proof of Theorem 4.18. We prove the correctness of Algorithm 10. The correctness of the DP-NN as well as the claimed size bounds follow by Proposition 4.1.

First observe that $\mathbf{h}(k)$ will be zero if and only if $k = \mathbf{p}_{\text{in}}$ and at least two otherwise. Therefore, due to $\mathbf{f}_{\text{in}}(p-k) \leq 2$, we obtain $\mathbf{g}(p, \mathbf{p}_{\text{in}}) = \mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}})$ and $\mathbf{g}(p,k) = 0$ for all $k \neq \mathbf{p}_{\text{in}}$. This implies $\sum_{k=1}^{p-1} \mathbf{g}(p,k) = \mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}})$. Hence, the assignment in line 6 correctly implements the desired recursion.

Note that, if $p \leq \mathbf{p}_{\text{in}}$, then the sum $\sum_{k=1}^{p-1} \mathbf{g}(p,k)$ is zero. In this case, the result is as desired, too, because taking only the current item into the knapsack solution provides profit at least p with size $\mathbf{s}_{\text{in}}$. □

4.6.3. Smaller RNNs with Provable Approximation Guarantees

In order to overcome the drawback due to the dependence of the network width on p^*, we provide a construction, called *FPTAS-NN*, that uses less neurons, at the cost of losing optimality, while still obtaining solution values of provable quality in the worst case.

The *FPTAS-NN* is an RNN of depth five and fixed width w which, when applied iteratively to the n items of a Knapsack instance, always produces a solution value of at least $1 - \mathcal{O}(n^2/\sqrt{w})$ times the optimum solution value. In particular, an ε-approximate solution value can be guaranteed by choosing width $w \in \mathcal{O}(n^4/\varepsilon^2)$. The dependence of the width on ε is unavoidable, unless $\mathrm{P} = \mathrm{NP}$. To the best of our knowledge, our results establish the first rigorous trade-off between the size of neural networks for CO problems and their worst-case solution quality.

As in the standard Knapsack FPTAS [Hoc97; Vaz01; WS11], the idea of this construction is to round the profit values if p^* becomes too large for an exact computation.

As for the DP-NN, we first give the high-level idea of the network structure and rounding procedure, before formally defining the FPTAS-NN as a MAAP and proving its correctness.

High-level idea of the construction

Let $P \in \mathbb{N}$ be a fixed number. The FPTAS-NN computes values $g(p, i)$ for every $p \in [P]$ and $i \in [n]$. These values are similar to the values $f(p, i)$ of the previous section, there is, however, one major difference. Let $p_i^* = \sum_{j=1}^{i} p_j$ be the total profit of the first i items. As soon as p_i^* exceeds P, we can no longer store the best possible size value for every possible profit value but have to round profits instead. The granularity we want to use for rounding is $d_i := \max\{1, p_i^*/P\}$. We construct the FPTAS-NN to compute values $g(p, i)$, $p \in [P]$, $i \in [n]$, such that we can guarantee the existence of a subset of $[i]$ that has size at most $g(p, i)$ and profit at least $p\, d_i$. Moreover, this is done in such a way that the optimal solution cannot have a considerably higher profit value. That is, we prove a worst-case approximation guarantee for the solution found by the FPTAS-NN.

In addition to the values of g, the FPTAS-NN must also propagate the current total profit value p_i^* through the network in order to determine the rounding granularity in each step. Hence, in the i-th step, it receives $P + 3$ inputs, namely $g(p, i - 1)$ for $p \in [P]$, p_{i-1}^*, p_i, and s_i. It computes $P + 1$ outputs, namely $g(p, i)$ for $p \in [P]$ and p_i^*. Figure 4.5 illustrates the recurrent structure of this NN.

As previously, we again drop the index i in order to make the recurrent structure obvious. We denote the $n_0 = P + 3$ input parameters by $\mathbf{g}_{\mathrm{in}}(p)$, for $p \in [P]$, as well as $\mathbf{p}_{\mathrm{in}}^*$, $\mathbf{p}_{\mathrm{in}}$, and $\mathbf{s}_{\mathrm{in}}$. The $P+1$ output values are denoted by $\mathbf{g}_{\mathrm{out}}(p)$, for $p \in [P]$, and $\mathbf{p}_{\mathrm{out}}^*$.

Similar to the DP-NN in Section 4.6.2, the basic idea is to implement a recursion of the type

$$\mathbf{g}_{\mathrm{out}}(p) = \min\{\mathbf{g}_{\mathrm{in}}(p^{(1)}), \mathbf{g}_{\mathrm{in}}(p^{(2)}) + \mathbf{s}_{\mathrm{in}}\} \quad \text{for } p \in [P],$$

where the first argument of the minimum represents the option of not using item i, while the second one corresponds to using it. Notice, however, that $p^{(1)}$ and $p^{(2)}$ cannot simply be calculated as p and $p - \mathbf{p}_{\mathrm{in}}$, respectively, since we may have to round with different granularities in two successive steps. Therefore, the rough structure of the FPTAS-NN is as follows: first, $\mathbf{p}_{\mathrm{in}}^*$ and $\mathbf{p}_{\mathrm{in}}$ are used in order to calculate the old and new rounding granularities $\mathbf{d}_{\mathrm{old}} = \max\{1, \mathbf{p}_{\mathrm{in}}^*/P\}$, as well as $\mathbf{d}_{\mathrm{new}} = \max\{1, (\mathbf{p}_{\mathrm{in}}^* + \mathbf{p}_{\mathrm{in}})/P\}$. Since this

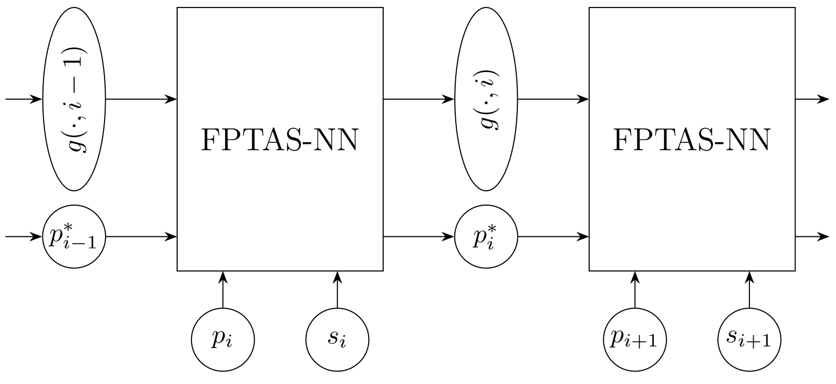
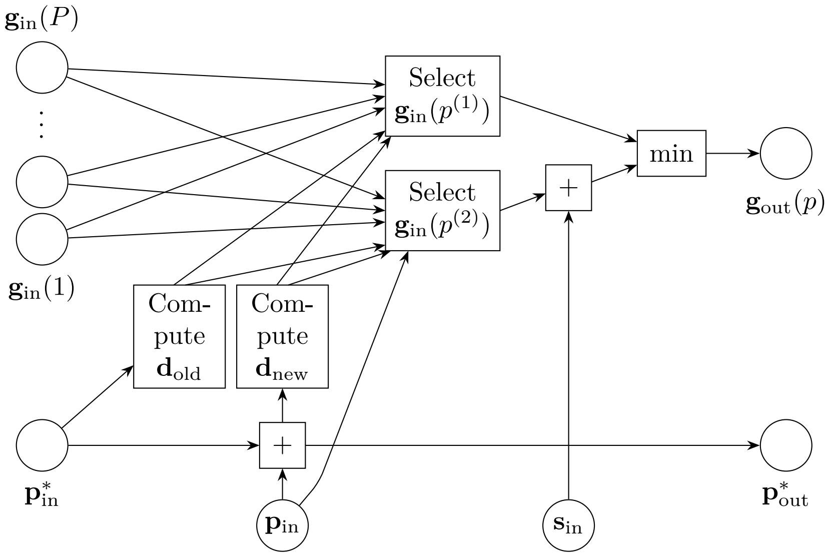


Figure 4.5.: Recurrent structure of the FPTAS-NN for the Knapsack Problem.



$\mathbf{g}_{\text{in}}(P)$
$\mathbf{g}_{\text{in}}(1)$
Select $\mathbf{g}_{\text{in}}(p^{(1)})$
Select $\mathbf{g}_{\text{in}}(p^{(2)})$
min
$+$
$\mathbf{g}_{\text{out}}(p)$
Compute $\mathbf{d}_{\text{old}}$
Compute $\mathbf{d}_{\text{new}}$
$+$
$\mathbf{p}^*_{\text{in}}$
$\mathbf{p}_{\text{in}}$
$\mathbf{s}_{\text{in}}$
$\mathbf{p}^*_{\text{out}}$

Figure 4.6.: High-level idea how the FPTAS-NN computes $\mathbf{g}_{\text{out}}(p)$, $p \in [P]$, and $\mathbf{p}^*_{\text{out}}$ from the inputs.

computation consists of maxima and weighted sums only, it can easily be achieved by an NN with one hidden layer. Second, the granularities are used in order to select $\mathbf{g}_{\text{in}}(p^{(1)})$ and $\mathbf{g}_{\text{in}}(p^{(2)})$ from the inputs. Below we give some more details on how this is being done. The value of $p^{(2)}$ also depends on $\mathbf{p}_{\text{in}}$. Third, the final recursion is established as in the DP-NN. In addition to $\mathbf{g}_{\text{out}}(p)$, for $p \in [P]$, we also output $\mathbf{p}^*_{\text{out}} = \mathbf{p}^*_{\text{in}} + \mathbf{p}_{\text{in}}$ in order to keep track of the rounding granularities in successive steps. An overview of the entire network structure is given in Figure 4.6.

Suppose we use the network for processing the i-th item. For each $p \in [P]$ we want to determine a (preferably small) value $\mathbf{g}_{\text{out}}(p)$ such that there is a subset of $[i]$ of total profit at least $p\,\mathbf{d}_{\text{new}}$ and total size at most $\mathbf{g}_{\text{out}}(p)$. For each $p' \in [P]$, we know that there is a subset of $[i-1]$ of total profit at least $p'\mathbf{d}_{\text{old}}$ and total size at most $\mathbf{g}_{\text{in}}(p')$. We have two options: ignoring item i or using it. If we ignore it, then each $p^{(1)}$ with $p^{(1)}\mathbf{d}_{\text{old}} \geq p\,\mathbf{d}_{\text{new}}$ allows us to choose $\mathbf{g}_{\text{out}}(p) = \mathbf{g}_{\text{in}}(p^{(1)})$. If we do use the i-th item, however, then each $p^{(2)}$ with the property $p^{(2)}\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} \geq p\,\mathbf{d}_{\text{new}}$ allows us to choose $\mathbf{g}_{\text{out}}(p) = \mathbf{g}_{\text{in}}(p^{(2)}) + \mathbf{s}_{\text{in}}$.

Hence, we want to choose $p^{(1)}$ and $p^{(2)}$ as small as possible such that these properties are fulfilled. Therefore, the units labeled 'Select $\mathbf{g}_{\text{in}}(p^{(1)})$' and 'Select $\mathbf{g}_{\text{in}}(p^{(2)})$' in Figure 4.6 are constructed by setting all other connections to zero except for those belonging to the smallest values of $p^{(1)}$ and $p^{(2)}$ satisfying the above properties. Similar to how we computed $\mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}})$ in the previous section, this requires two hidden layers and $\mathcal{O}(P^2)$ neurons in total.

In total, the FPTAS-NN has depth 5. The first hidden layer computes the rounding granularities, two hidden layers are required to select $\mathbf{g}_{\text{in}}(p^{(1)})$ and $\mathbf{g}_{\text{in}}(p^{(2)})$ and a final hidden layer computes the minimum in the actual recursion. The width and size of the FPTAS-NN are in the order of $\mathcal{O}(P^2)$. Unfolding the RNN and viewing it as a single feedforward NN executing the whole FPTAS results in depth $\mathcal{O}(n)$ and size $\mathcal{O}(nP^2)$.

In the next subsection, we provide a formal description of the FPTAS-NN as a MAAP and prove the following theorem. Part one ensures that the FPTAS-NN produces only feasible Knapsack solutions, while part two shows that the FPTAS-NN indeed provides a fully polynomial-time approximation scheme to solve the Knapsack Problem.

Theorem 4.19. *For each $n \in \mathbb{N}$ and each $\varepsilon \in \,]0,1]$, there exists an RNN, namely the FPTAS-NN, with depth five and width and size $\mathcal{O}(n^4/\varepsilon^2)$ such that, when iteratively applied to n items of a Knapsack instance with capacity $S = 1$, $s_i \in \,]0,1]$, and $p_i \in \mathbb{N}$, for $i \in [n]$, it computes values $g(p,i)$ with the following two properties:*

(i) For every $i \in [n]$ and every $p \in [P]$ with $g(p,i) \leq 1$, there exists a subset of $[i]$ with profit at least pd_i and size at most $g(p,i)$.

(ii) If p^{OPT} is the profit of the optimal solution and $p^{\text{NN}} = \max\{pd_n \mid g(p,n) \leq 1\}$ is the best possible profit found by the FPTAS-NN, then $p^{\text{NN}} \geq (1-\varepsilon)p^{\text{OPT}}$.

Theorem 4.19 implies a trade-off between the width of the NN and the precision of the Knapsack solution in the following sense. For achieving an approximation ratio of $1-\varepsilon$, an NN of width $\mathcal{O}(n^4/\varepsilon^2)$ is required. In other words, the FPTAS-NN with fixed width w achieves a worst-case approximation ratio of $1 - \mathcal{O}(n^2/\sqrt{w})$.

Observe that, assuming P $\neq$ NP, it is clear that the size of the NN must grow if ε tends to zero. Hence, complexity theory implies that a width-quality trade-off cannot be avoided. Still, it remains as an open question whether the growth rates implied by our construction are best possible.

Details of the construction and correctness

We now formally describe the FPTAS-NN as a MAAP and prove that it provides strong worst-case approximation guarantees.

As for the DP-NN, we truncate the values of g at 2, that is, instead of any value larger than 2 including $+\infty$, we just use the value 2. The FPTAS-NN is applied to a Knapsack instance in the following way. Using the initialization $g(p,0) = 2$ for $p \in [P]$ and $p_0^* = 0$, for each item $i = 1,\ldots,n$, we feed the inputs $\mathbf{g}_{\text{in}}(p) = g(p,i-1)$ for $p \in [P]$, $\mathbf{p}_{\text{in}}^* = p_{i-1}^*$, $\mathbf{p}_{\text{in}} = p_i$, and $\mathbf{s}_{\text{in}} = s_i$ into the network and store the outputs as $g(p,i) = \mathbf{g}_{\text{out}}(p)$ for $p \in [P]$ and $p_i^* = \mathbf{p}_{\text{out}}^*$. The FPTAS-NN itself is defined as the NN emerging from applying Proposition 4.1 to the MAAP given in Algorithm 11.

Let us now show Theorem 4.19.

Algorithm 11: MAAP corresponding to the FPTAS-NN for a given, fixed parameter $P \in \mathbb{N}$.

```
   Input: $\mathbf{g}_{\text{in}}(p)$ for $p \in [P]$, as well as $\mathbf{p}_{\text{in}}$, $\mathbf{s}_{\text{in}}$, and $\mathbf{p}^*_{\text{in}}$.
   // Compute $\mathbf{p}^*_{\text{out}}$ as well as old and new rounding granularities:
1  $\mathbf{p}^*_{\text{out}} \leftarrow \mathbf{p}^*_{\text{in}} + \mathbf{p}_{\text{in}}$
2  do parallel
3      $\mathbf{d}_{\text{old}} \leftarrow \max\{\mathbf{p}^*_{\text{in}}/P, 1\}$
4  and
5      $\mathbf{d}_{\text{new}} \leftarrow \max\{\mathbf{p}^*_{\text{out}}/P, 1\}$
   // The remaining MAAP is executed in parallel over all $p \in [P]$:
6  for each $p \in [P]$ do parallel
7      do parallel
8          for each $k \in [P]$ with $k \geq p$ do parallel
               // Idea: $\mathbf{h}^{(1)}_{+}(p,k) + \mathbf{h}^{(1)}_{-}(p,k)$ equals 0 iff $k = p^{(1)}$ and is $\geq 2$
                  otherwise.
9              do parallel
10                 $\mathbf{h}^{(1)}_{+}(p,k) \leftarrow \max\{0, 2P(p\mathbf{d}_{\text{new}} - k\mathbf{d}_{\text{old}})\}$
11             and
12                 $\mathbf{h}^{(1)}_{-}(p,k) \leftarrow \max\{0, 2P((k-1)\mathbf{d}_{\text{old}} - p\mathbf{d}_{\text{new}}) + 2\}$
               // Idea: $\mathbf{h}^{(1)}_{\text{select}}(p,k)$ equals $2 - \mathbf{g}_{\text{in}}(k)$ iff $k = p^{(1)}$ and 0
                  otherwise.
13             $\mathbf{h}^{(1)}_{\text{select}}(p,k) \leftarrow \max\{0, 2 - \mathbf{g}_{\text{in}}(k) - \mathbf{h}^{(1)}_{+}(p,k) - \mathbf{h}^{(1)}_{-}(p,k)\}$
14     and
15         for each $k \in [P]$ with $k \leq p$ do parallel
               // Idea: $\mathbf{h}^{(2)}_{+}(p,k) + \mathbf{h}^{(2)}_{-}(p,k)$ equals 0 iff $k = p^{(2)}$ and is $\geq 2$
                  otherwise.
16             do parallel
17                 $\mathbf{h}^{(2)}_{+}(p,k) \leftarrow \max\{0, 2P(p\mathbf{d}_{\text{new}} - k\mathbf{d}_{\text{old}} - \mathbf{p}_{\text{in}})\}$
18             and
19                 $\mathbf{h}^{(2)}_{-}(p,k) \leftarrow \max\{0, 2P((k-1)\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} - p\mathbf{d}_{\text{new}}) + 2\}$
               // Idea: $\mathbf{h}^{(2)}_{\text{select}}(p,k)$ equals $\mathbf{g}_{\text{in}}(k)$ iff $k = p^{(2)}$ and 0
                  otherwise.
20             $\mathbf{h}^{(2)}_{\text{select}}(p,k) \leftarrow \max\{0, \mathbf{g}_{\text{in}}(k) - \mathbf{h}^{(2)}_{+}(p,k) - \mathbf{h}^{(2)}_{-}(p,k)\}$
       // Idea: $\mathbf{h}^{(1)}(p)$ equals $\mathbf{g}_{\text{in}}(p^{(1)})$.
21     $\mathbf{h}^{(1)}(p) \leftarrow 2 - \sum_{k=p}^{P} \mathbf{h}^{(1)}_{\text{select}}(p,k)$
       // Idea: $\mathbf{h}^{(2)}(p)$ equals $\mathbf{g}_{\text{in}}(p^{(2)})$.
22     $\mathbf{h}^{(2)}(p) \leftarrow \sum_{k=1}^{p} \mathbf{h}^{(2)}_{\text{select}}(p,k)$
23     $\mathbf{g}_{\text{out}}(p) \leftarrow \min\{\mathbf{h}^{(1)}(p), \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p)\}$
24 return $\mathbf{g}_{\text{out}}(p)$ for $p \in [P]$ and $\mathbf{p}^*_{\text{out}}$.
```

Proof of Theorem 4.19. For a given instance of the Knapsack Problem with n items and a given $\varepsilon \in]0, 1]$, use the FPTAS-NN with parameter $P := \lceil n^2/\varepsilon \rceil$. For a fixed $p \in [P]$, let $p^{(1)}$ and $p^{(2)}$ be the smallest possible integers that satisfy $p^{(1)}\mathbf{d}_{\text{old}} \geq p\mathbf{d}_{\text{new}}$ and $p^{(2)}\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} \geq p\mathbf{d}_{\text{new}}$, respectively. The proof consists of the following steps: first, we verify that the resulting NN has the claimed depth, width, and size. Then, we go through the MAAP line by line to show that $\mathbf{h}^{(1)}(p)$ and $\mathbf{h}^{(2)}(p)$ indeed basically equal $\mathbf{g}_{\text{in}}(p^{(1)})$ and $\mathbf{g}_{\text{in}}(p^{(2)})$, respectively. Finally, we use this to prove properties (i) and (ii) by induction on the iteration count i.

Concerning the depth, observe that lines 1, 21 and 22 do not contribute to complexity measure d since they are only assignments of affine transformations. Due to parallelization, lines 2 to 5 contribute 1, lines 7 to 20 contribute 2, and line 23 contributes 1. Therefore, for the full MAAP A, we obtain $d(A) = 4$, that is, the FPTAS-NN has depth 5 by Proposition 4.1. Concerning the width and size, the bottleneck is given by the nested parallel **for** loops, resulting in $w(A), s(A) \in \mathcal{O}(P^2) = \mathcal{O}(n^4/\varepsilon^2)$, completing our analysis of depth, width, and size of the FPTAS-NN.

We now go through the MAAP line by line to show that it does what we want. Observe that, in the i-th step, if we feed the inputs $\mathbf{p}^*_{\text{in}} = p^*_{i-1}$ and $\mathbf{p}_{\text{in}} = p_i$ into the network, $\mathbf{d}_{\text{old}}$ and $\mathbf{d}_{\text{new}}$ equal the desired rounding granularities d_{i-1} and d_i, respectively. We show some claims asserting that the intermediate helper variables within the MAAP indeed match the intuition stated in the comment lines of the MAAP.

Claim 4.20. *For each $p, k \in [P]$ with $k \geq p$, we have $\mathbf{h}^{(1)}_+(p,k) + \mathbf{h}^{(1)}_-(p,k) = 0$ if and only if $k = p^{(1)}$. Otherwise, we have $\mathbf{h}^{(1)}_+(p,k) + \mathbf{h}^{(1)}_-(p,k) \geq 2$.*

Proof. Obviously, it holds that $\mathbf{h}^{(1)}_+(p,k) = 0$ if and only if $k\mathbf{d}_{\text{old}} \geq p\mathbf{d}_{\text{new}}$. On the other hand, using that $\mathbf{d}_{\text{old}}$ and $\mathbf{d}_{\text{new}}$ are integer multiples of $\frac{1}{P}$, we obtain

$$\begin{aligned}
& \mathbf{h}^{(1)}_-(p,k) = 0 \\
\Leftrightarrow \quad & (k-1)\mathbf{d}_{\text{old}} \leq p\mathbf{d}_{\text{new}} - \frac{1}{P} \\
\Leftrightarrow \quad & (k-1)\mathbf{d}_{\text{old}} < p\mathbf{d}_{\text{new}} \\
\Leftrightarrow \quad & \text{no integer } k' < k \text{ satisfies } k'\mathbf{d}_{\text{old}} \geq p\mathbf{d}_{\text{new}}.
\end{aligned}$$

This proves the first part of the claim. The second part follows because, again, $\mathbf{d}_{\text{old}}$ and $\mathbf{d}_{\text{new}}$ are integer multiples of $\frac{1}{P}$ and, hence, $\mathbf{h}^{(1)}_+(p,k) + \mathbf{h}^{(1)}_-(p,k)$ is an integer multiple of 2. ■

Claim 4.21. *For each $p, k \in [P]$ with $k \leq p$, we have $\mathbf{h}^{(2)}_+(p,k) + \mathbf{h}^{(2)}_-(p,k) = 0$ if and only if $k = p^{(2)}$. Otherwise, we have $\mathbf{h}^{(2)}_+(p,k) + \mathbf{h}^{(2)}_-(p,k) \geq 2$.*

Proof. Analogous to Claim 4.20 where an additional summand $\mathbf{p}_{\text{in}}$ is inserted at the right places. ■

Claim 4.22. *For each $p \in [P]$, if $p^{(1)} \leq P$, we have $\mathbf{h}^{(1)}(p) = \mathbf{g}_{\text{in}}(p^{(1)})$. If $p^{(1)} > P$, we have $\mathbf{h}^{(1)}(p) = 2$.*

Proof. Note that $p^{(1)}$ is never smaller than p. If $p \le p^{(1)} \le P$, then it holds that $\mathbf{h}^{(1)}_{\text{select}}(p, p^{(1)}) = 2 - \mathbf{g}_{\text{in}}(p^{(1)})$ and $\mathbf{h}^{(1)}_{\text{select}}(p, k) = 0$ for each $k \ne p^{(1)}$ by Claim 4.20. If $p^{(1)} > P$, then $\mathbf{h}^{(1)}_{\text{select}}(p, k) = 0$ for each k. Thus, the claim follows by the definition of $\mathbf{h}^{(1)}$. ∎

Claim 4.23. *For each $p \in [P]$, if $p^{(2)} \ge 1$, we have $\mathbf{h}^{(2)}(p) = \mathbf{g}_{\text{in}}(p^{(2)})$. If $p^{(2)} \le 0$, we have $\mathbf{h}^{(2)}(p) = 0$.*

Proof. We first show that $p^{(2)}$ is never larger than p by proving that $p\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} \ge p\mathbf{d}_{\text{new}}$. If $\mathbf{d}_{\text{new}} = 1$, then also $\mathbf{d}_{\text{old}} = 1$ holds and this statement is true. Otherwise, we have $\mathbf{d}_{\text{new}} = \frac{\mathbf{p}^*_{\text{in}} + \mathbf{p}_{\text{in}}}{P}$ and $\mathbf{d}_{\text{old}} \ge \frac{\mathbf{p}^*_{\text{in}}}{P}$. Hence, we obtain $p(\mathbf{d}_{\text{new}} - \mathbf{d}_{\text{old}}) \le \frac{p\mathbf{p}_{\text{in}}}{P} \le \mathbf{p}_{\text{in}}$. Therefore, in any case, $p\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} \ge p\mathbf{d}_{\text{new}}$ follows, and thus also $p^{(2)} \le p$.

If $1 \le p^{(2)} \le p$, then it follows that $\mathbf{h}^{(2)}_{\text{select}}(p, p^{(2)}) = \mathbf{g}_{\text{in}}(p^{(2)})$ and $\mathbf{h}^{(2)}_{\text{select}}(p, k) = 0$ for each $k \ne p^{(2)}$ by Claim 4.21. If $p^{(2)} \le 0$, then $\mathbf{h}^{(2)}_{\text{select}}(p, k) = 0$ holds for each k. Thus, the claim follows by the definition of $\mathbf{h}^{(2)}$. ∎

Having collected these claims about the functionality of all helper variables in the MAAP, we now turn towards proving the properties (i) and (ii).

(i) We show that the claim even holds for all values of p and i with $g(p, i) < 2$ and not only for those with $g(p, i) \le 1$.

We use induction on i. For the induction start ($i = 0$), nothing is to show due to the initialization $g(p, 0) = 2$ for all $p \in [P]$. For the induction step, suppose the claim is valid for all steps up to $i - 1$.

Fix some $p \in [P]$. By line 23, the output $g(p, i) = \mathbf{g}_{\text{out}}(p)$ in the i-th step equals $\min\{\mathbf{h}^{(1)}(p), \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p)\}$. In the following, we distinguish two cases. Recall that $p^{(1)}$ and $p^{(2)}$ are the smallest possible integers with $p^{(1)}\mathbf{d}_{\text{old}} \ge p\mathbf{d}_{\text{new}}$ and $p^{(2)}\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} \ge p\mathbf{d}_{\text{new}}$, respectively.

Case 1: $\mathbf{h}^{(1)}(p) \le \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p)$. This implies $g(p, i) = \mathbf{h}^{(1)}(p)$. If $\mathbf{h}^{(1)}(p) = 2$, nothing is to show. Otherwise, by Claim 4.22, we have $p^{(1)} \le P$ with $p^{(1)}\mathbf{d}_{\text{old}} \ge p\mathbf{d}_{\text{new}}$ and $g(p, i) = \mathbf{h}^{(1)}(p) = \mathbf{g}_{\text{in}}(p^{(1)}) = g(p^{(1)}, i-1)$. By induction, we obtain that there exists a subset of $[i-1]$ with size at most $g(p, i)$ and profit at least $p^{(1)}\mathbf{d}_{\text{old}}$. Hence, using the same items yields a subset of $[i]$ with size at most $g(p, i)$ and profit at least $p\mathbf{d}_{\text{new}}$. Thus, the claim is proven in this case.

Case 2: $\mathbf{h}^{(1)}(p) > \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p)$. This implies $g(p, i) = \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p)$. Note that this can only happen if $\mathbf{h}^{(2)}(p) < 2$ because $\mathbf{h}^{(1)}(p)$ has at most value 2. First, suppose that $p^{(2)} \le 0$. This implies $p_i = \mathbf{p}_{\text{in}} \ge p\mathbf{d}_{\text{new}}$. Hence, by using just item i, we obtain a subset of profit at least $p\mathbf{d}_{\text{new}}$ and size at most $s_i = \mathbf{s}_{\text{in}} \le \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p) = g(p, i)$, and we are done. Second, if $p^{(2)} \ge 1$, then Claim 4.23 implies that

$$g(p, i) = \mathbf{s}_{\text{in}} + \mathbf{h}^{(2)}(p) = \mathbf{s}_{\text{in}} + \mathbf{g}_{\text{in}}(p^{(2)}) = s_i + g(p^{(2)}, i-1).$$

By induction, we obtain that there exists a subset of $[i-1]$ with size at most $g(p, i) - s_i$ and profit at least $p^{(2)}\mathbf{d}_{\text{old}}$. Hence, adding item i to this subset yields a subset of $[i]$ with size at most $g(p, i)$ and profit at least $p^{(2)}\mathbf{d}_{\text{old}} + \mathbf{p}_{\text{in}} \ge p\mathbf{d}_{\text{new}}$. Thus, the claim is also proven in this case.

(ii) Let M^{OPT} be an optimal solution to the Knapsack instance and $M_i^{\text{OPT}} = M^{\text{OPT}} \cap [i]$ be the subset of $[i]$ chosen by the optimal solution. Let $s_i^{\text{OPT}} = \sum_{j \in M_i^{\text{OPT}}} s_j$ be the size of M_i^{OPT} and $p_i^{\text{OPT}} = \sum_{j \in M_i^{\text{OPT}}} p_j$ be the profit of M_i^{OPT}. The idea of the proof is that in each step, we lose at most a profit of d_i compared to the optimal solution. Formally, we prove the following claim by induction on i.

Claim 4.24. *For every $i \in [n]$ and every $p \leq \left\lceil \frac{p_i^{\text{OPT}}}{d_i} \right\rceil - i$, we have $g(p,i) \leq s_i^{\text{OPT}}$.*

Proof. The induction start is settled by extending the claim to $i = 0$, for which it is trivial. For the induction step, suppose the claim is valid for all steps up to $i-1$. Fix a value $p \leq \left\lceil \frac{p_i^{\text{OPT}}}{d_i} \right\rceil - i$. Let again $p^{(1)}$ and $p^{(2)}$ be the smallest possible integers with $p^{(1)} d_{i-1} \geq p d_i$ and $p^{(2)} d_{i-1} + p_i \geq p d_i$, respectively. We distinguish two cases.

Case 1: $i \notin M^{\text{OPT}}$, i.e., the optimal solution does not use item i. Observe that

$$\begin{aligned} p d_i &\leq \left(\left\lceil \frac{p_i^{\text{OPT}}}{d_i} \right\rceil - i \right) d_i \\ &\leq p_i^{\text{OPT}} - (i-1) d_i \\ &= p_{i-1}^{\text{OPT}} - (i-1) d_i \\ &\leq p_{i-1}^{\text{OPT}} - (i-1) d_{i-1} \\ &\leq \left(\left\lceil \frac{p_{i-1}^{\text{OPT}}}{d_{i-1}} \right\rceil - (i-1) \right) d_{i-1}. \end{aligned}$$

Hence, we obtain

$$p^{(1)} \leq \left\lceil \frac{p_{i-1}^{\text{OPT}}}{d_{i-1}} \right\rceil - (i-1) \tag{4.5}$$

by the definition of $p^{(1)}$. In particular, $p^{(1)} \leq \frac{p_{i-1}^*}{d_{i-1}} \leq P$ by the definition of d_{i-1}. Therefore, Claim 4.22 and line 23 imply $g(p,i) \leq g(p^{(1)}, i-1)$. Due to inequality (4.5), it further follows by induction that $g(p,i) \leq g(p^{(1)}, i-1) \leq s_{i-1}^{\text{OPT}} = s_i^{\text{OPT}}$, which settles the induction step in this case.

Case 2: $i \in M^{\text{OPT}}$, i.e., the optimal solution uses item i. Observe that

$$\begin{aligned} p d_i &\leq \left(\left\lceil \frac{p_i^{\text{OPT}}}{d_i} \right\rceil - i \right) d_i \\ &\leq p_i^{\text{OPT}} - (i-1) d_i \\ &= p_{i-1}^{\text{OPT}} + p_i - (i-1) d_i \\ &\leq p_{i-1}^{\text{OPT}} + p_i - (i-1) d_{i-1} \\ &\leq \left(\left\lceil \frac{p_{i-1}^{\text{OPT}}}{d_{i-1}} \right\rceil - (i-1) \right) d_{i-1} + p_i. \end{aligned}$$

Hence, we obtain

$$p^{(2)} \leq \left\lceil \frac{p_{i-1}^{\mathrm{OPT}}}{d_{i-1}} \right\rceil - (i-1) \tag{4.6}$$

by the definition of $p^{(2)}$.

If $p^{(2)} \leq 0$, Claim 4.23 and line 23 imply $g(p,i) \leq s_i \leq s_i^{\mathrm{OPT}}$. If $p^{(2)} \geq 0$, Claim 4.23 and line 23 imply $g(p,i) \leq g(p^{(2)}, i-1) + s_i$. Due to inequality (4.6), we can further conclude by induction that $g(p,i) \leq g(p^{(2)}, i-1) + s_i \leq s_{i-1}^{\mathrm{OPT}} + s_i = s_i^{\mathrm{OPT}}$, which finalizes the induction step. ∎

Now, making use of Claim 4.24, we obtain that $g\left(\left\lceil \frac{p^{\mathrm{OPT}}}{d_n} \right\rceil - n, n\right) \leq s_n^{\mathrm{OPT}} \leq 1$. Therefore, it holds that

$$p^{\mathrm{NN}} \geq \left(\left\lceil \frac{p^{\mathrm{OPT}}}{d_n} \right\rceil - n\right) d_n \geq p^{\mathrm{OPT}} - n d_n. \tag{4.7}$$

If $d_n = 1$, that is, $p^* \leq P$, then we have that $d_i = 1$ for all $i \in [n]$. Hence, in each step and for each $p \in [P]$, we have $p^{(1)} = p$ and $p^{(2)} = p - p_i$. Therefore, by Claim 4.22, Claim 4.23, and line 23, the FPTAS-NN behaves like the DP-NN from Section 4.6.2 that executes the exact dynamic program and the theorem follows.

Otherwise, if $d_n > 1$, we have $d_n = \frac{p^*}{P}$. Since there must exist one item with profit at least $\frac{p^*}{n}$, we obtain $p^{\mathrm{OPT}} \geq \frac{p^*}{n}$ and, hence, $n d_n = \frac{np^*}{P} \leq \frac{n^2 p^{\mathrm{OPT}}}{P}$. Together with (4.7), this implies $\frac{p^{\mathrm{NN}}}{p^{\mathrm{OPT}}} \geq 1 - \frac{n^2}{P} \geq 1 - \varepsilon$. □

4.6.4. Empirical Evidence for Quadratic Width

While the running time of the classical Knapsack dynamic program depends only linearly on p^*, the width of the DP-NN is $\mathcal{O}((p^*)^2)$. In our construction, the quadratic factor arises from dynamically finding $\mathbf{f}_{\mathrm{in}}(p - \mathbf{p}_{\mathrm{in}})$ in a hard-coded network, as explained in Section 4.6.2. For similar reasons, the width of the FPTAS-NN grows with $1/\varepsilon^2$ instead of $1/\varepsilon$.

The natural question to ask is whether this quadratic dependence can be avoided by a different construction. While this question remains open from a purely theoretical point of view, in this section, we provide empirical evidence that the quadratic factor might indeed be necessary due to inherent properties of ReLU feedforward NNs.

For details about the experimental setup, including used soft- and hardware, random data generation and systematic of seeds, training and testing setup, hyperparameters, as well as the source code, please refer to the corresponding subsection below. At first, we only describe the necessary information to understand the key findings.

Similar to the DP-NN of Section 4.6.2, we train an NN with three hidden layers and variable width to execute one step of the Knapsack dynamic program, that is, to map $\mathbf{f}_{\mathrm{in}}$, $\mathbf{p}_{\mathrm{in}}$, and $\mathbf{s}_{\mathrm{in}}$ to $\mathbf{f}_{\mathrm{out}}$, for random Knapsack instances. For the 25 different values $\{3, 6, 9, \ldots, 75\}$ of p^*, we increase the width in steps of 25 until a mean squared error (MSE) loss of at most 0.005 is reached. The threshold 0.005 is carefully chosen such that NNs with reasonable width are empirically able to achieve it. Below we also

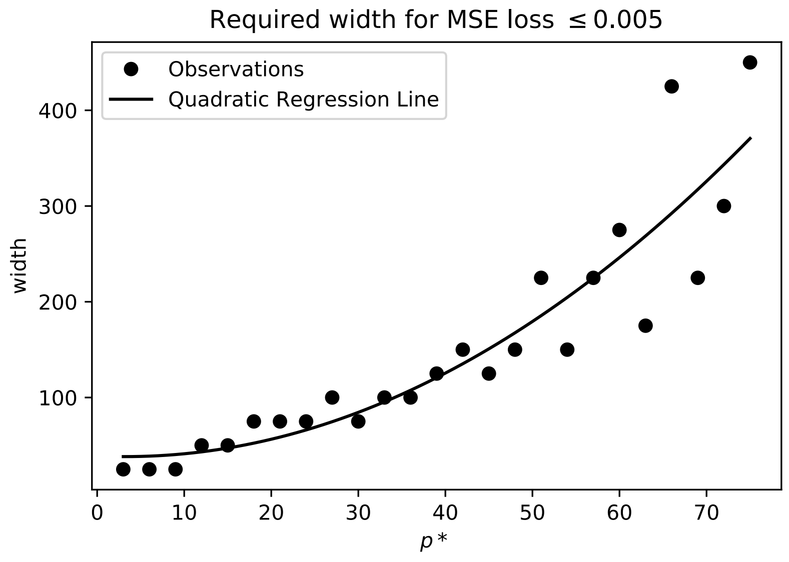


Figure 4.7.: Required width to achieve a mean squared error of at most 0.005 as a function of p^*.

show that other thresholds yield similar results. Figure 4.7 shows for each value of p^* the required width to achieve an MSE of at most 0.005.

In order to statistically test whether a quadratic dependence is more likely than a linear relation, we use linear regression. Assuming the required width is given by a function

$$\text{width} = a_0 + a_1 p^* + a_2 (p^*)^2 + \text{noise},$$

the resulting least squares regression curve can be seen in Figure 4.7. Testing the null hypothesis $a_2 = 0$ against the alternative $a_2 \neq 0$, we obtain a p-value of 1.1 %, which we judge to be significant evidence that a quadratic relation is more likely than a linear one.

In order to show that our experimental results do not depend on the particular choice of the MSE threshold, we conducted the experiments with thresholds other than 0.005 as well. In Figures 4.8 and 4.9 you can see the results for the thresholds 0.00375 and 0.0025, respectively. One can clearly observe that a quadratic dependence seems to be reasonable in these cases, too. Testing the null hypothesis that the dependence is actually linear against the alternative of a quadratic relation yields p-values of 0.0046 % and 0.12 %, respectively, which is, again, a significant indication of a quadratic dependence.

Of course, one should take these result with a grain of salt since the superlinear relation might have multiple reasons. For instance, it is unclear, whether the difficulty to train larger networks has a stronger effect than the expressivity of ReLU NNs. Still, we find that this computational study supports our theoretical size bounds.

Detailed Experimental Setup

In this subsection we describe in detail how we conducted the experiments described above.

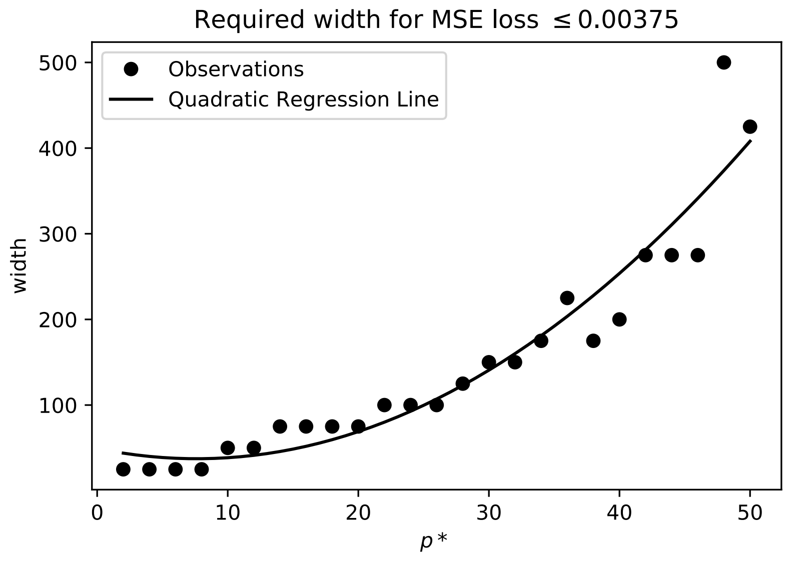


Figure 4.8.: Required width to achieve a mean squared error of at most 0.00375 as a function of p^*.

Hard- and software. All our experiments were conducted on a machine with an Intel Core i5-8500 6-Core 64-bit CPU and 15.5 GB RAM, using the openSUSE Leap 15.1 Linux distribution. We use Python 3.8.5 with Numpy 1.19.1, Tensorflow 2.2.0 in CPU-only mode, and Statsmodels 0.11.1 for regression and statistical tests.

Generation of random Knapsack instances. For a given value of p^* we sample a set of items of total profit $\sum p_i = p^*$ in the following way: the profit of the i-th item is always chosen uniformly at random among all integer values between 1 and $p^* - \sum_{i'=1}^{i-1} p_{i'}$. This is repeated until all profits sum up to p^*. We chose this procedure in order to make it likely to have both, very profitable and less profitable items within one instance. Finally, we shuffle the order of the items. For each item, we then pick a size value uniformly in the interval $[0, 1]$ and normalize these values such that their sum $\sum s_i$ equals a random value chosen uniformly in $]1, 2[$. We certainly want $\sum s_i > 1$, because otherwise all items would fit into the Knapsack. On the other hand, $\sum s_i < 2$ makes sense, because in our DP-NN (compare Section 4.6.2), we use 2 as a placeholder for $+\infty$.

Preparation of the training set. Since we can generate arbitrarily many random Knapsack instances, we use an infinite training set and never train on the same data point twice. A Knapsack instance with n items yields n training points, namely one for each step of the dynamic program. In order to avoid having the n training points belonging to one instance in successive order, we generate training points belonging to several instances and shuffle them.

Neural network architecture. For a given value p^* and width w, the corresponding neural network used consists of an input layer with $p^* + 2$ neurons (corresponding to

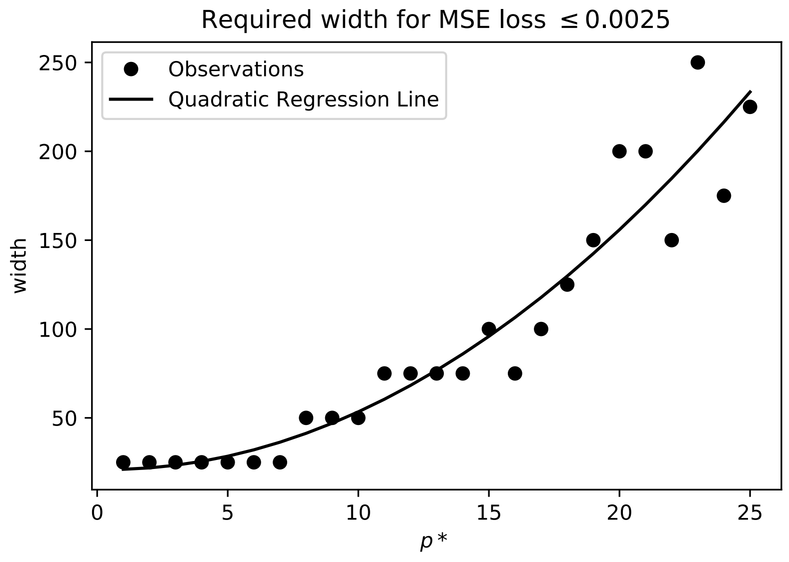


Figure 4.9.: Required width to achieve a mean squared error of at most 0.0025 as a function of p^*.

the p^* values of the previous dynamic programming state, as well as the scalar profit and size values), three hidden layers with w neurons each and ReLU activations, as well as an output layer with p^* neurons (corresponding to the new state of the dynamic program) without further activation function. That way, we choose the same depth as in the DP-NN (Section 4.6.2), but do not employ the specific knowledge about the widths of the three hidden layers. As in the DP-NN, each layer is not only connected to the previous layer, but also receives direct connections from the input layer. In total, by our results of Section 4.6.2, this architecture is theoretically able to exactly represent the dynamic programming transition function if $w \in \mathcal{O}((p^*)^2)$.

Training and testing a specific network. For a given value p^* and width w, we train the neural network architecture described above as follows. We train in epochs of 1000 batches with batch size 32 using mean squared error (MSE) loss and the Adam optimizer, which is a robust standard choice. It makes sense to use MSE loss as it punishes errors in both directions equally hard and large errors harder than small errors. All other (hyper)parameters are left to the default settings of Tensorflow, which empirically works quite well for our problem type and size. It takes between 8 and 30 seconds to train one epoch with our machine setup. We train until there are two successive epochs without improvement in training loss, which empirically happens after 10 to 80 epochs. Using a test set that is randomly generated in the same way as the training set, we evaluate the trained network on 1000 batches of size 32 each. The resulting mean squared error is our final result for the given values of p^* and w.

Finding the required width. For a given value p^* and a given MSE threshold, we always train networks with increasing widths $25, 50, 75, \ldots$ in steps of 25 as described above until a network achieves an MSE less or equal to the threshold on the test set.

Seed generation. In order to ensure reproducibility of our experiments, each time before we train and test an NN with given value p^* and width w, we reset the random seeds of both Numpy and Tensorflow to $257 \cdot p^* + w$, where 257 is just an arbitrary prime number. Note that these packages only guarantee the same result of random experiments if the same package versions are used.

Regression and statistical tests. For each of the three threshold values 0.005, 0.0025, and 0.00375, we find the required width for achieving the respective threshold for 25 different values of p^*. With the help of the ordinary least squares (OLS) regression utility of the Statsmodels package, we find a quadratic regression line for the p^*-width relation in each of the three cases. The output of the OLS automatically contains the reported p-values for testing whether the coefficient of the quadratic term is zero.

Source code. The source code is publicly available at

`https://github.com/ChristophHertrich/neural-knapsack`.

There, the file `README.md` explains how the code can be used to reproduce the experiments described here.

4.7. The Constrained Shortest Path Problem

We close the technical part of this chapter by demonstrating how our results on the Knapsack Problem can be generalized to even more CO problems. More specifically, we consider a common generalization of the Shortest Path Problem and the Knapsack Problem, namely the NP-hard Constrained Shortest Path Problem. We only give an intuition how this can be done based on our results of the previous section, without going into detail.

As for the Shortest Path Problems in Section 4.3, let $G = (V, E)$ be a fixed directed graph and $(c_{uv})_{u,v \in V}$ be a (nonnegative) length matrix. In addition, the input graph is also equipped with a (nonnegative) resource matrix $(r_{uv})_{u,v \in V}$. The task is to find a minimum length path P from a source vertex s to any other vertex, but this time subject to a resource constraint $\sum_{uv \in P} r_{uv} \leq R$ for a given resource limit R. An extensive overview of solution approaches to this problem can be found, e.g., in the dissertation by Ziegelmann [Zie01]. Similar to the Knapsack Problem, there exist two natural pseudo-polynomial dynamic programs, one of them parametrized by length values and the other one by resource values. Both can be implemented on an NN by combining the ideas from Section 4.6.2 with the NN for the Bellmann-Ford algorithm above. We showcase this for the variant parametrized by the length values. This dynamic program recursively calculates values $f(c, v)$ representing the minimum amount of resource needed for a path from s to v with length at most c by

$$f(c, v) = \min\{f(c-1, v), \min_{u \in V \setminus \{v\}}\{f(c - c_{uv}, u) + r_{uv}\}\}.$$

For fixed c, u, and v, the term $f(c - c_{uv}, u) + r_{uv}$ can be calculated by a similar construction as we computed $\mathbf{f}_{\text{in}}(p - \mathbf{p}_{\text{in}}) + \mathbf{s}_{\text{in}}$ in Section 4.6.2. Assuming an upper bound c^* on the optimal objective value, this can be achieved with constant depth and $\mathcal{O}(c^*)$ width. Hence, it remains to compute a minimum of at most n numbers in order to compute $f(c, v)$. Thus, each single value $f(c, v)$ can be computed with depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(nc^*)$. We have to compute $\mathcal{O}(nc^*)$ of these values, but for fixed c, all these values can be computed in parallel. Therefore, the whole dynamic program can be executed on an NN with depth $\mathcal{O}(c^* \log n)$ and a total size of $\mathcal{O}(n^2(c^*)^2 \log n)$. This is pseudo-polynomial, which is the best we can hope for due to the NP-hardness of the problem. Moreover, similar to the Knapsack Problem, this dynamic program can be turned into an FPTAS by downscaling and rounding the length values. This observation can be used to obtain a width-quality trade-off for the Constrained Shortest Path Problem similar to what we have shown in Section 4.6.3.

4.8. Future Research

We conclude this chapter on the computational power of NNs in the context of CO problems by identifying several open research questions in continuation of our findings.

The most exciting and, at the same time, probably most difficult question for further research is to find theoretical lower bounds on the size of NNs for certain problems. We believe the most promising approach towards achieving such bounds would be to use methods from arithmetic circuit complexity, particularly about tropical circuits. Even though NNs are strictly more powerful than these tropical circuits, as we pointed out earlier, it might be worthwhile to deeply investigate relations between these two concepts to find out whether some ideas from arithmetic circuit complexity can be used to prove lower bounds for NNs.

In light of the lack of lower bounds, a natural second direction for further research is to improve the constructions presented in this chapter to obtain smaller NNs. This is particularly promising and interesting in those cases where the size of the resulting NN is larger than the time complexity of the best known algorithms for these problems. For example, in case of the Maximum Flow Problem, our NN has size $\mathcal{O}(n^2m^2)$, compared to the best known running time $\mathcal{O}(nm)$ of a classical algorithm. It would be very interesting to know whether our construction is best possible or whether smaller constructions are conceivable. Similarly, the DP-NN to solve the Knapsack Problem has width $\mathcal{O}((p^*)^2)$, while the running time of the pseudo-polynomial dynamic program depends only linearly on p^*. Despite the empirical evidence we provide that this quadratic dependence might indeed be necessary, a firm theoretical analysis would be desirable.

Closely related is the question whether one can use the ability of NNs to perform computations in parallel more effectively. As an example, our NN to solve the Maximum Flow Problem has depth $\mathcal{O}(n^2m^2)$. Even though, as we argued, highly parallel architectures (polylogarithmic depth) with polynomial width are unlikely, we wonder whether it is possible to achieve a depth that is a polynomial of lower degree than n^2m^2, while still maintaining polynomial total size.

Apart from the problems considered in this chapter, another natural way to continue research concerning the computational power of NNs is to extend our results to more (CO) problems. For example, it is an open question whether polynomial-size NNs exist that compute the objective values of the Assignment Problem, various Weighted Matching Problems, or Minimum Cost Flow Problems for real-valued arc/edge weights as inputs.

Finally, the ultimate goal of a continued study about the computational power of NNs could be a meta-result in the following sense: which properties of a (CO) problem determine whether small NNs are possible to solve this problem? Are there, for example, necessary or sufficient conditions on a dynamic program, which allow to turn it into an NN? Similar in spirit, Woeginger [Woe00] classifies dynamic programs that guarantee the existence of a fully polynomial-time approximation scheme.

We hope that this facet of NN complexity promotes further research on the interplay between NNs and classical (CO) algorithms.

5 Training Complexity

Does a small dimensionality help when fitting neural network weights to data?

The results in this chapter are based on a preprint with Vincent Froese and Rolf Niedermeier [FHN21].

5.1. Introduction

Reducing the dimensionality of the input data while preserving most of the information contained in it is a popular preprocessing step in many machine learning applications [BFN19; MPH09]. Besides many other benefits induced by a smaller input dimension, a natural follow-up question is whether this step also simplifies the process of training the actual machine learning model. In this chapter, we study this question for the model of neural networks (NNs). More precisely, we investigate to what extent a low input dimension can help in lowering the worst-case computational complexity of NN training, that is, fitting NN weights to given training data. To achieve this goal, we employ tools and concepts from parameterized complexity analysis.

In this chapter, we completely focus on 2-layer ReLU NNs. Even though this simplifies the model significantly and many practical applications use more layers, understanding these shallow NNs is a natural first step in gaining a fundamental understanding of general ReLU NNs. Additionally, this case already gives rise to a variety of non-trivial questions and challenges; see, for example, recent works by Bakshi, Jayaram, and Woodruff [BJW19] and Goel et al. [Goe+21].

5.1.1. Chapter-Specific Notation

In contrast to Chapter 4, where bold symbols distinguish variables that depend on the NN input from constants and weights, in this chapter, bold symbols are used to distinguish vectors from scalars.

We also slightly change our notation with respect to specific symbols. In order to align with the literature about parameterized complexity of geometric problems, d always represents the input dimension (and not the depth) of the considered NN. Additionally, k represents the number of hidden neurons and n is used to denote the number of training samples in the training set.

5.1.2. Empirical Risk Minimization for ReLU Neural Networks

Formally, the problem we investigate in this chapter is to minimize the so-called *empirical risk* for a 2-layer ReLU NN with respect to a *loss function* $\ell\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$, mapping the predicted and the true label to a loss value. The problem of training a two-layer ReLU neural network with k hidden neurons and a single output neuron (see Figure 5.1) is defined as follows:

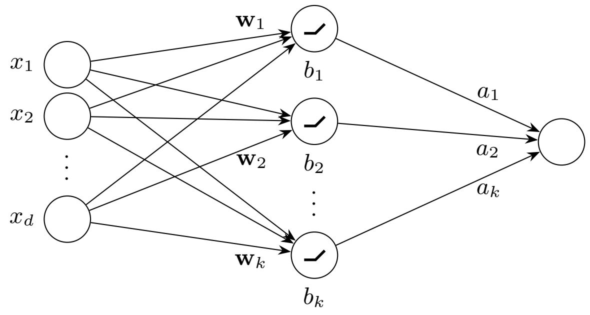


Figure 5.1.: The neural network architecture we study in this chapter: After the input layer (left) with d input neurons, we have one hidden layer with k ReLU neurons and a single output neuron without additional activation function.

k-ReLU(ℓ)
Input: Data points $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$.
Task: Find weight vectors $\mathbf{w}_1, \dots, \mathbf{w}_k \in \mathbb{R}^d$, biases $b_1, \dots, b_k \in \mathbb{R}$, and coefficients $a_1, \dots, a_k \in \{-1, 1\}$ that minimize

$$\sum_{i=1}^{n} \ell\left(\sum_{j=1}^{k} a_j \max\{0, \langle \mathbf{w}_j, \mathbf{x}_i \rangle + b_j\},\ y_i \right).$$

The corresponding decision problem is to decide whether a target training error of at most $\gamma \in \mathbb{R}$ can be achieved. As usual in the context of computational complexity analysis, we consider the decision instead of the optimization version. One of our main contributions is to prove strong computational hardness results that already hold for only a single hidden ReLU neuron, that is, $k = 1$.

We focus on the case of ℓ^p-loss functions, that is, $\ell(\hat{y}, y) = |\hat{y} - y|^p$, for $p \in]0, \infty[$. Note that this includes both convex and concave loss functions (with respect to the absolute error $|\hat{y} - y|$). In addition, we also investigate the limit cases $p = 0$ and $p = \infty$. The ℓ^0-loss (which is completely insensitive to outliers) simply counts the number of points that are not perfectly fitted, while the ℓ^∞-loss only cares about outliers, that is, it measures only the largest error on any data point. Without loss of generality, we always assume that $n \geq d + 1$ because otherwise the problem can be solved in the lower-dimensional affine hull of the input data points.[1]

5.1.3. Parameterized Complexity and the Exponential Time Hypothesis

Parameterized complexity analysis is a methodology to study the impact of certain parameters on the time complexity of computational problems [DF13; Cyg+15]. For that purpose, every classical problem instance I is associated with a parameter value $k \in \mathbb{N}$.

[1]To see this, assume that all $\mathbf{x}_i$ are contained in an affine subspace A with dimension strictly less than d. Using a bijective affine transformation $T\colon A \to \mathbb{R}^{\dim A}$, we now can solve the equivalent k-ReLU(ℓ) problem with the lower-dimensional data points $(T(x_i), y_i)$, $i \in [n]$. The solution weights for the original problem can then be obtained by composing T with the affine transformation given by the weights of the modified problem.

For example, in this chapter, the considered parameter associated with an instance of the ReLU training problem is the dimension d of the input space.

A parameterized problem is *fixed-parameter tractable* (contained in the class FPT) if there exists an algorithm solving any instance (I, k) in $f(k) \cdot \text{poly}(|I|)$ time, where f is a function solely depending on k. Note that fixed-parameter tractability implies polynomial time for constant parameter values where, importantly, the degree of the polynomial is independent from the parameter value.

One way to show parameterized intractability of a problem is to use a *parameterized reduction* to prove that it is *W[1]-hard*. The class W[1] is a superset of FPT that also contains parameterized problems which are conjectured not to be in FPT (for example, Clique parameterized by the size of the requested clique). A parameterized reduction from a problem P to another problem P' is an algorithm that maps an instance (I, k) of P in $f(k) \cdot \text{poly}(|I|)$ time to an instance (I', k') of P' such that $k' \leq g(k)$ for some function g and (I, k) is a yes-instance of P if and only if (I', k') is a yes-instance of P'.

The class XP contains all parameterized problems which can be solved in polynomial time if the parameter is a constant, that is, in time $f(k) \cdot |I|^{g(k)}$. It is known that FPT $\subseteq$ W[1] $\subseteq$ XP and that FPT $\subsetneq$ XP.

The *Exponential Time Hypothesis* (ETH) [IP01] states that 3-SAT cannot be solved in subexponential time in the number n of variables of the Boolean input formula. That is, there exists a constant $c > 0$ such that 3-SAT cannot be solved in $\mathcal{O}(2^{cn})$ time. The ETH implies that FPT $\neq$ W[1] (and P $\neq$ NP) [Cyg+15]. It also implies running time lower bounds, for example, that Clique cannot be solved in $\rho(k) \cdot n^{o(k)}$ time for any function ρ, where k is the size of the sought clique [Cyg+15].

5.1.4. Related Work

The NP-hardness of empirical risk minimization with ℓ^2-loss for a single ReLU was shown independently by Dey, Wang, and Xie [DWX20] and Goel et al. [Goe+21]. The work of Goel et al. [Goe+21] is probably the one closest to the work in this chapter. They provided an in-depth study concerning tight hardness results for 2-layer ReLU networks such as NP-hardness, conditional running time lower bounds, and hardness of approximation. Arora et al. [Aro+18] provided a polynomial-time algorithm for k-ReLU(ℓ) for $d \in \mathcal{O}(1)$ and convex loss ℓ; in terms of parameterized algorithmics, this is an XP-algorithm for parameter d: the degree of the polynomial of the running time (only) depends on d. The underlying idea of their algorithm is to basically iterate over all $\mathcal{O}(n^d)$ hyperplane partitions of the n data points. Indeed, as pointed out by Goel et al. [Goe+21], since there are at most 2^n partitions, the same algorithm implies fixed-parameter tractability for the parameter n. Moreover, Goel et al. [Goe+21] remarked that the well-known Exponential Time Hypothesis (ETH) implies that no $2^{o(n)}$-time algorithm exists. Deciding whether zero error ($\gamma = 0$) is possible (that is, *realizable* data) is polynomial-time solvable for a single ReLU by linear programming [Goe+21] and NP-hard for two ReLUs [Goe+21]. Approximation has been subject to further works [DWX20; Dia+20; Goe+21]. Furthermore, Bakshi, Jayaram, and Woodruff [BJW19] and Chen, Klivans, and Meka [CKM20] showed fixed-parameter tractability results for related but different learning concepts of ReLU networks and Boob, Dey, and Lan [BDL20] studied the computational complexity

Table 5.1.: (Parameterized) computational complexity of training a single ReLU neuron with respect to parameter d (input dimension) for ℓ^p-loss functions.

	Hardness	Algorithm
$p \in [0, 1[$	W[1]-h. + no $n^{o(d)}$-time alg. (Theorem 5.1)	$n^{\mathcal{O}(d)}$ poly(n, d) (Theorem 5.4)
$p \in [1, \infty[$	W[1]-h. + no $n^{o(d)}$-time alg. (Theorem 5.1)	n^d poly(n, d) [Aro+18]
$p = \infty$	-	poly(n, d) (Proposition 5.3)

of ReLU networks where the output neuron is also a ReLU. Abrahamsen, Kleist, and Miltzow [AKM21] showed that neural network training (of more complex architectures) is complete for the class $\exists\mathbb{R}$ (existential theory of the reals), implying that the problem is presumably not contained in NP.

Finally, note that the number of dimensions appears naturally in parameterized complexity studies for geometric problems [GKR09; KKW15]; moreover, it occurs also in recent studies for principal component analysis (PCA) [FGS20; Sim+19] and in computer vision [CCN20].

5.1.5. Overview and Main Results

In the course of this chapter we provide several contributions towards understanding the computational complexity of ReLU NN training parameterized by the input dimension d with respect to various loss functions.

First, in Section 5.2, we prove that training a two-layer ReLU NN is already computationally intractable even for a single hidden neuron and small d (Theorem 5.1), that is, we show W[1]-hardness for parameter d and provide an ETH-based lower bound of $n^{\Omega(d)}$ matching the running time upper bound of $n^{\mathcal{O}(d)}$ of the training algorithm due to Arora et al. [Aro+18]. Hence, our result shows that the combinatorial search among all $\mathcal{O}(n^d)$ possible hyperplane partitions is essentially the best one can do. Notably, our hardness results even hold for very sparse data points with binary labels.

Second, in Section 5.3, we contribute a polynomial-time algorithm (for arbitrary dimension) for training a single-neuron ReLU network when using the ℓ^∞-loss function (Proposition 5.3). This generalizes the polynomial-time result due to Goel et al. [Goe+21] for the zero-error case.

Third, in Section 5.4, we complement the W[1]-hardness result with a matching upper bound as follows. We extend the XP-result for convex loss functions by Arora et al. [Aro+18] to concave loss functions (Theorem 5.4) for arbitrary 2-layer NNs (any $k \geq 1$). Note that for W[1]-hard problems, an XP-algorithm is the best one can hope for. Hence, we completely settle the computational complexity parameterized by dimension d of training a two-layer ReLU NN for any ℓ^p-loss with $p \in [0, \infty[$.

Table 5.1 provides an overview of our results for the special case of a single hidden neuron ($k = 1$).

Finally, we point out open research questions in Section 5.5.

5.1.6. Discussion of other Parameters besides Dimensionality

The core theme of this chapter is to better understand how the dimension parameter d influences the computational complexity of ReLU network training as defined above. To this end, we conduct a parameterized complexity analysis [DF13]. Before moving on to study the key parameter d, let us briefly discuss other parameters occurring in our setting. The most natural other parameters are: the number k of ReLU neurons in the hidden layer, the number n of input points, and the maximum target error γ.

It turns out that the parameterized complexity for these three parameters is already settled due to simple observations. First, the case $k = 1$ (that is, 1-ReLU(ℓ)) is known to be NP-hard for ℓ^2-loss [DWX20; Goe+21] and we even extend the NP-hardness to ℓ^p-loss for arbitrary $p \in [0, \infty[$ (see Theorem 5.1); this renders the parameter k hopeless in terms of getting efficient parameterized algorithms. For the parameter n, fixed-parameter tractability was already observed by Goel et al. [Goe+21] (see also related work). Finally, for $\gamma = 0$, the case $k = 1$ is polynomial-time solvable [Goe+21] and for $k \geq 2$ NP-hardness is known [BJW19; Goe+21]. Thus, the dimension d is clearly the most interesting parameter also from a theoretical point of view.

5.2. Hardness of Training a Single ReLU in Small Dimension

In this section, we show that there is no hope to obtain an FPT algorithm with respect to parameter d for training even the simplest possible architecture consisting of a single neuron with respect to the ℓ^p-loss for any $p \in [0, \infty[$. To this end, we show intractability of the problem 1-ReLU(ℓ^p). For $p = 0$, the problem is to minimize the number of data points that are not perfectly fitted.

Theorem 5.1. *For every $p \in [0, \infty[$, 1-ReLU(ℓ^p) is NP-hard, W[1]-hard with respect to dimension d, and it cannot be solved in $\rho(d) \cdot n^{o(d)}$ time for any computable function ρ unless the Exponential Time Hypothesis fails.*

Note that Goel et al. [Goe+21] proved NP-hardness and running time lower bounds for additive approximation (with ℓ^2-loss). Their results are based on gap reductions from problems such as finding dense subgraphs and coloring hypergraphs. The reductions are focused on providing a gap which guarantees the approximation hardness. They achieve this by using a "large" number d of dimensions (typically equal to the number of vertices of the input (hyper)graph). For our purpose, however, we need a more fine-grained parameterized reduction where d is "small". To this end, we reduce from a colored variant of Clique such that d is bounded linearly in the size of the clique.

Proof of Theorem 5.1. We reduce from the following problem.

Multicolored Clique

Input:	An undirected graph $G = (V, E)$ where the vertices are colored with k colors.
Question:	Does G contain a k-clique (a complete subgraph with k vertices) with exactly one vertex from each color?

Multicolored Clique is NP-hard, W[1]-hard with respect to k, and not solvable in time $\rho(k) \cdot |V|^{o(k)}$ for any computable function ρ unless the Exponential Time Hypothesis fails [Cyg+15]. We give a parameterized reduction (which is indeed also a polynomial-time reduction) from Multicolored Clique to 1-ReLU(ℓ^p) where the dimension of the data points is $d = 2k$. Hence, the theorem follows.

Let $G = (V, E)$ be an undirected graph with $N := |V|$ vertices and let $c\colon V \to [k]$ be a vertex coloring. We denote by $V_j = \{v_{j,1}, \dots, v_{j,N_j}\}$ the set of vertices with color j, where $N_j := |V_j|$. In the following, we construct a set of data points from $\mathbb{R}^{2k}$ with labels in $\{0, 1\}$, as well as a target error $\gamma \in \mathbb{R}$, such that these data points can be fitted by a ReLU function with ℓ^p-error at most γ if and only if a multicolored k-clique exists in G.

We set the target error to $\gamma := N - k$. Next, we define a small value $0 < \delta < 1$ such that making an absolute error of value $1 - \delta$ on $N - k + 1$ different inputs already exceeds the threshold γ. For $p = 0$, we simply choose $\delta := 0.5$. For $p > 0$, let $\tilde{p} := \max\{p, 1\}$ and $\delta := 1/(2\tilde{p}(N - k + 1))$. This yields

$$\begin{aligned}(1 - \delta)^p (N - k + 1) &\geq (1 - \tilde{p}\delta)(N - k + 1) \\ &> (1 - 2\tilde{p}\delta)(N - k + 1) \\ &= N - k = \gamma, \end{aligned} \tag{5.1}$$

where, in the case $p > 1$, the first inequality follows from Bernoulli's inequality, and in the case $p \leq 1$, it follows from $x^p \geq x$ for all $x \in [0, 1]$.

In addition, we define a large integer M such that making an absolute error of δ on M different inputs also exceeds the threshold γ. For $p = 0$, we choose $M := N - k + 1$. For $p > 0$, we set $M := \lceil \gamma/\delta^p \rceil + 1$, which implies

$$M\delta^p > \gamma. \tag{5.2}$$

Note that $\gamma \in \mathcal{O}(N)$ and $1/\delta \in \mathcal{O}(N)$. Thus, M is polynomially bounded in the size of G.

Let $N_{\max} := \max_{j \in [k]} N_j$ be the maximum number of vertices of one color. For our reduction we need $N_{\max}$ distinct rational points on the unit circle centered at the origin. For example, one can choose

$$\tilde{\mathbf{x}}_i := \left(\frac{1 - i^2}{1 + i^2}, \frac{2i}{1 + i^2} \right) \in \mathbb{Q}^2$$

for each $i = 1, 2, \dots, N_{\max}$ [ST94]. For each vertex $v_{j,i} \in V$, $j \in [k]$, $i \in [N_j]$, let $\mathbf{x}_{j,i} = (\mathbf{0}_{2j-2}, \tilde{\mathbf{x}}_i, \mathbf{0}_{2k-2j}) \in \mathbb{R}^{2k}$ be the point $\tilde{\mathbf{x}}_i$ lifted to $2k$ dimensions, where each color corresponds to two dimensions. Here, we use the notation $\mathbf{0}_r$ for the r-dimensional zero-vector. We add two types of data points to our instance (see Figure 5.2 for an example). First, for each vertex $v_{j,i} \in V$, add the point $(\mathbf{x}_{j,i}, 1) \in \mathbb{R}^{2k} \times \mathbb{R}$. Second, for each pair of distinct vertices $v_{j,i} \neq v_{j',i'} \in V$, if they cannot both be part of a multicolored clique because they are non-adjacent or have the same color, then add M copies of the point $\big((\mathbf{x}_{j,i} + \mathbf{x}_{j',i'})/2, 0\big) \in \mathbb{R}^{2k} \times \mathbb{R}$. This finishes the construction.

We now show that there exists a multicolored clique of size k in G if and only if these data points can be fitted by a ReLU with ℓ^p-error at most γ. First, assume that the vertices $v_{1,i_1}, \dots, v_{k,i_k}$ form a multicolored clique of size k in G. We define $\varepsilon := 1 - \max_{i \neq i' \in [N_{\max}]} \langle \tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_{i'} \rangle$. Observe that it holds $\varepsilon > 0$ since the points $\tilde{\mathbf{x}}_i$,

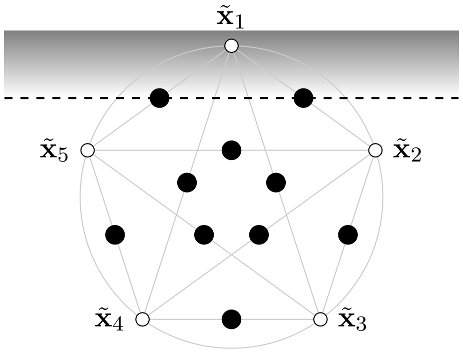


Figure 5.2.: Schematic illustration of the reduction from Multicolored Clique. Shown are two dimensions corresponding to one of the k colors. The white points $\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_5$ correspond to vertices of that color and have label 1. Black points indicate M copies of the corresponding middle point with label 0. The dashed line indicates the hyperplane defined by the weight vector $\mathbf{w}$ of the ReLU neuron and the shaded area indicates the predictions of the neuron (darker means larger values). The idea is that exactly one white point can be predicted correctly (which selects the corresponding vertex to be in the clique) without predicting a black point incorrectly and thereby exceeding the error.

$i \in [N_{\max}]$, are distinct points on the unit circle. Let $\mathbf{w} := 2/\varepsilon \cdot (\tilde{\mathbf{x}}_{i_1}, \tilde{\mathbf{x}}_{i_2}, \dots, \tilde{\mathbf{x}}_{i_k}) \in \mathbb{R}^{2k}$ and $b := 1 - 2/\varepsilon$. We claim that the ReLU function $f(\mathbf{x}) = \max\{0, \langle \mathbf{w}, \mathbf{x} \rangle + b\}$ achieves an ℓ^p-error of exactly $\gamma = N - k$. To see this, first note that for each $j \in [k]$, we have

$$\langle \mathbf{w}, \mathbf{x}_{j,i_j} \rangle + b = 2/\varepsilon \cdot \langle \tilde{\mathbf{x}}_{i_j}, \tilde{\mathbf{x}}_{i_j} \rangle + 1 - 2/\varepsilon = 1,$$

where we used that $\tilde{\mathbf{x}}_{i_j}$ lies on the unit circle. Hence, the k points $\mathbf{x}_{1,i_1}, \dots, \mathbf{x}_{k,i_k}$ are perfectly fitted. Second, for each vertex $v_{j,i} \in V \setminus \{v_{1,i_1}, \dots, v_{k,i_k}\}$ outside the clique, we have

$$\langle \mathbf{w}, \mathbf{x}_{j,i} \rangle + b = 2/\varepsilon \cdot \langle \tilde{\mathbf{x}}_{i_j}, \tilde{\mathbf{x}}_i \rangle + 1 - 2/\varepsilon \le 2/\varepsilon \cdot (1 - \varepsilon) + 1 - 2/\varepsilon = -1,$$

where the inequality follows from our choice of ε. Hence, for each of these $N - k$ points, we have $f(\mathbf{x}_{j,i}) = 0$, that is, we incur an error of 1. Finally, for each pair of distinct vertices $v_{j,i} \neq v_{j',i'} \in V$ that are either non-adjacent or have the same color, note that only one of the two vertices can belong to the clique. Thus, making use of our two calculations above, we obtain

$$\langle \mathbf{w}, (\mathbf{x}_{j,i} + \mathbf{x}_{j',i'})/2 \rangle + b = ((\langle \mathbf{w}, \mathbf{x}_{j,i} \rangle + b) + (\langle \mathbf{w}, \mathbf{x}_{j',i'} \rangle + b))/2 \le (1 - 1)/2 = 0.$$

Hence, all points with label 0 are fitted exactly and the total ℓ^p-error is equal to $\gamma = N - k$.

For the reverse direction, suppose that there exist $\mathbf{w} \in \mathbb{R}^{2k}$ and $b \in \mathbb{R}$ such that the ReLU function $f(\mathbf{x}) = \max\{0, \langle \mathbf{w}, \mathbf{x} \rangle + b\}$ achieves an ℓ^p-error of at most $\gamma = N - k$. We show that the set

$$C := \{v_{j,i} \in V \mid f(\mathbf{x}_{j,i}) > \delta\}$$

forms a multicolored clique in G. First, observe that $|C| \ge k$, because otherwise all the points associated with vertices in $V \setminus C$ would incur a total ℓ^p-error of at least

$(1-\delta)^p(N-k+1)$, which is larger than γ by (5.1). Hence, it remains to show for each pair of vertices $v_{j,i} \neq v_{j',i'} \in C$ that they belong to different color classes and are adjacent. Suppose the contrary. Then, by construction, the 1-ReLU(ℓ^p) instance also contains M copies of the point $\big((\mathbf{x}_{j,i}+\mathbf{x}_{j',i'})/2, 0\big) \in \mathbb{R}^{2k}\times\mathbb{R}$. From $\langle \mathbf{w}, \mathbf{x}_{j,i}\rangle + b > \delta$ and $\langle \mathbf{w}, \mathbf{x}_{j',i'}\rangle + b > \delta$, it follows by linearity that

$$f((\mathbf{x}_{j,i}+\mathbf{x}_{j',i'})/2) \geq \langle \mathbf{w}, (\mathbf{x}_{j,i}+\mathbf{x}_{j',i'})/2\rangle + b > \delta.$$

Thus, we incur an ℓ^p-error of at least $M\delta^p$, which is larger than γ by (5.2). Hence, C is indeed a multicolored k-clique. □

A closer inspection of the above proof reveals that hardness even holds for a more restricted problem.

Corollary 5.2. *For $p \in [0,\infty[$, 1-ReLU(ℓ^p) is NP-hard, W[1]-hard with respect to d, and cannot be solved in $\rho(d)\cdot n^{o(d)}$ time for any computable function ρ (assuming the Exponential Time Hypothesis), even if all input data points contain at most four non-zero entries and have binary labels.*

We further remark that the basic idea of the reduction in the proof of Theorem 5.1 also works for more general loss functions. Essentially, the only necessary condition is that the value M can be chosen such that it is polynomially bounded in the size of the graph G and satisfies an inequality analogous to (5.2) where δ^p is replaced by $\ell(\delta, 0)$.

Our findings tell us that in order to achieve fixed-parameter tractability, one has to consider other parameters to combine with the dimension d. A natural parameter is the target loss γ. However, this is not a promising parameter since it can be made arbitrarily small by scaling all values. If we consider the number σ of different coordinate values of the $\mathbf{x}_i$, then we trivially obtain fixed-parameter tractability in combination with d since the overall number of different data points is at most σ^d. Hence, the algorithm by Arora et al. [Aro+18] runs in $\sigma^{d^2}\cdot \text{poly}(nd)$ time.

To sum up, identifying promising parameters (or parameter combinations) to obtain tractable cases remains a challenge worthwhile further investigation.

5.3. Polynomial-time Algorithm for a Single ReLU with Maximum Norm

As pointed out by Goel et al. [Goe+21], deciding whether given data points are realizable by a single ReLU neuron (that is, whether $\gamma = 0$) can be done in polynomial time via linear programming. In other words, it is possible to check whether the input points can be perfectly fitted by a single ReLU neuron and, in case of a positive answer, to find the corresponding weights in polynomial time. Recall that the same problem is NP-hard in the case of two (or more) neurons by Goel et al. [Goe+21].

In this section, we extend this polynomial-time algorithm to minimizing the ℓ^∞-loss, that is, minimizing the maximum prediction error. In fact, we provide a polynomial-time optimization algorithm (not only decision) for a problem variant that generalizes ℓ^∞-loss minimization. In this variant, the real labels y_i for the data points $\mathbf{x}_i$ are replaced with target intervals $[\alpha_i, \beta_i]$ with $\alpha_i \leq \beta_i$ and we aim to minimize the maximum deviation of a prediction from its corresponding target interval. To this end, we define $\mathrm{dist}_{\alpha,\beta}(t) \coloneqq \max\{\alpha - t, 0, t - \beta\}$ to be the *distance* of $t \in \mathbb{R}$ to the interval $[\alpha, \beta]$.

ReLU(ℓ^∞-Interval)
Input: Data points $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$,
and interval boundaries $\alpha_1 \leq \beta_1, \dots, \alpha_n \leq \beta_n \in \mathbb{R}$.
Task: Find $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ that minimize

$$\max_{i\in[n]} \mathrm{dist}_{\alpha_i,\beta_i}(\max\{0, \langle \mathbf{w}, \mathbf{x}_i \rangle + b\}).$$

Note that we obtain ℓ^∞-loss minimization by setting $\alpha_i = \beta_i = y_i$ for all $i \in [n]$.

Proposition 5.3. *ReLU(ℓ^∞-Interval) can be solved in polynomial time.*

Proof. We show that an optimal solution can be found via solving a series of linear programs. For each $i \in [n]$ with $\alpha_i > 0$, our algorithm finds out whether the optimal objective value γ^* is larger or smaller than α_i. In the first case, the prediction $\langle \mathbf{w}, \mathbf{x}_i \rangle + b$ is allowed to be arbitrarily small, while in the second case we need to ensure the lower bound $\langle \mathbf{w}, \mathbf{x}_i \rangle + b \geq \alpha_i - \gamma^*$. Therefore, we implement a binary search to find an interval in which the optimal objective value γ^* is contained as follows. Let $\{\tilde{\alpha}_1, \tilde{\alpha}_2, \dots, \tilde{\alpha}_r\}$ be the set of all distinct positive α_i-values, $i \in [n]$, sorted by index such that $0 =: \tilde{\alpha}_0 < \tilde{\alpha}_1 < \cdots < \tilde{\alpha}_r < \tilde{\alpha}_{r+1} \coloneqq \infty$. Let $s^* \in [r+1]$ denote the (unknown) index with $\gamma^* \in [\tilde{\alpha}_{s^*-1}, \tilde{\alpha}_{s^*}[$. For each $s \in [r+1]$, we define a linear program denoted by LP(s) which minimizes the maximum deviation under the assumption that only the predictions for data points $\mathbf{x}_i$ with $\alpha_i \geq \tilde{\alpha}_s$ are bounded from below, while all other predictions can be arbitrarily small.

$$\begin{aligned}
\min_{\mathbf{w},b,\gamma}\quad & \gamma && \\
\text{s.t.}\quad & \langle \mathbf{w}, \mathbf{x}_i \rangle + b \in [\alpha_i - \gamma, \beta_i + \gamma], && i \in [n] \text{ with } \alpha_i \geq \tilde{\alpha}_s, \\
& \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq \beta_i + \gamma, && i \in [n] \text{ with } \alpha_i < \tilde{\alpha}_s, \\
& \gamma \geq -\beta_i, && i \in [n], \\
& \gamma \geq 0. &&
\end{aligned} \tag{LP(s)}$$

Here, the constraint $\gamma \geq -\beta_i$ is only relevant if $\beta_i < 0$. In this case, it is needed to ensure that the error is at least $-\beta_i$ because a ReLU unit can only output nonnegative values.

Suppose we already knew the optimal index s^*. Observe that, by construction of LP(s^*), a triplet $(\mathbf{w}, b, \gamma)$ is an optimal solution for LP(s^*) if and only if $(\mathbf{w}, b)$ is optimal for the problem ReLU(ℓ^∞-Interval) with objective value γ. Hence, it only remains to show how s^* can be found. To this end, let $\gamma(s)$ be the objective value of LP(s) for each $s \in [r+1]$. Note that $\gamma(s_1) \geq \gamma(s_2)$ for $s_1 < s_2$ because the set of constraints of LP(s_1) is a superset of the constraints of LP(s_2). Hence, for $s > s^*$, it follows that

$$\gamma(s) \leq \gamma(s^*) = \gamma^* < \tilde{\alpha}_{s^*} \leq \tilde{\alpha}_{s-1}.$$

Similarly, for $s < s^*$, we obtain

$$\gamma(s) \geq \gamma(s^*) = \gamma^* \geq \tilde{\alpha}_{s^*-1} \geq \tilde{\alpha}_s.$$

As a consequence, we can determine whether $s < s^*$, $s = s^*$, or $s \geq s^*$ by solving LP(s) and comparing $\gamma(s)$ with $\tilde{\alpha}_s$ and $\tilde{\alpha}_{s-1}$. Thus, using binary search and solving $\mathcal{O}(\log n)$ linear programs, we can determine s^* and the optimal solution γ^* together with the corresponding weights $\mathbf{w}$ and b. □

We remark that, analogously to the original problem with labels y_i, the zero-error case for the variant with intervals $[\alpha_i, \beta_i]$ can be solved with a single linear program instead of a binary search: It suffices to run LP(1) once. This results in objective value 0 if and only if all data points can be fitted precisely within their intervals.

5.4. Polynomial-time Algorithm for Concave Loss in Fixed Dimension

In this section, we prove that, for any loss function of the form $\ell(\hat{y}, y) = \tilde{\ell}(|\hat{y} - y|)$ where $\tilde{\ell}\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is concave, the problem k-ReLU(ℓ) is polynomial-time solvable for constant d (that is, it is in XP with respect to d). In particular, this covers the case of ℓ^p-loss for $p \in [0, 1[$. Notably, concave loss functions can yield increased robustness by mitigating the influence of outliers. For convex loss functions, in particular for the ℓ^p-loss with $p \geq 1$, an analogous result has already been shown by Arora et al. [Aro+18, Theorem 4.1]. More precisely, they showed that, if ℓ is convex, then k-ReLU(ℓ) can be solved in $\mathcal{O}(2^k n^{dk} \operatorname{poly}(n, d, k))$ time. The idea of their algorithm is essentially to try out all $\mathcal{O}(n^d)$ hyperplane partitions of the n input points for each of the k ReLU neurons and solve a corresponding convex program.

For the concave case, we follow a similar approach. The only but decisive difference is that the occurring subproblems are not convex programs. Instead, we show that they can be written as optimization problems over polyhedra with an objective function that is piecewise concave. It is well-known that global optima of concave problems always occur at a vertex of the feasible polyhedron [Ben95] and that it is possible to enumerate all vertices of the polyhedron in XP-time [KP03]. However, since in our case the objective function is only piecewise concave, it is possible that no vertex is a global optimum. Instead, we need to enumerate all vertices of all concave pieces of the feasible region. We show that this can still be done in XP-time, completing the parameterized complexity classification picture.

Theorem 5.4. *For every loss of the form $\ell(\hat{y}, y) = \tilde{\ell}(|\hat{y} - y|)$, with $\tilde{\ell}\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ being concave, the problem k-ReLU(ℓ) is solvable in time $2^k (nk)^{\mathcal{O}(dk)} \operatorname{poly}(n, d, k)$.*

Proof. Following the approach by Arora et al. [Aro+18, Algorithm 1], for each neuron $j \in [k]$, we consider each coefficient $a_j \in \{-1, 1\}$ and each hyperplane partition $P_+^j \cup P_-^j = [n]$, $P_+^j \cap P_-^j = \emptyset$, of the n (indices of the) data points (that is, there exists a $(d-1)$-dimensional hyperplane, defined by a vector $\mathbf{w}_j$ and a bias b_j, separating P_+^j and P_-^j, compare Figure 5.3). Here, P_+^j is the *active* set, where $\langle \mathbf{w}_j, \mathbf{x}_i \rangle + b_j \geq 0$ shall hold for each $i \in P_+^j$ and $\langle \mathbf{w}_j, \mathbf{x}_i \rangle + b_j \leq 0$ for each $i \in P_-^j$. As in the algorithm by Arora et al. [Aro+18], this results in a total of (at most) $2^k n^{dk}$ subproblems. For fixed coefficients a_j and fixed partitions (P_+^j, P_-^j), $j \in [k]$, the corresponding subproblem (compare Line 8 in Algorithm 1 of Arora et al. [Aro+18]) is the following:

$$\begin{aligned} \min_{\mathbf{w}_1,\dots,\mathbf{w}_k,\, b_1,\dots,b_k} \quad & \sum_{i=1}^{n} \tilde{\ell}\left(\left| y_i - \sum_{j\colon i\in P_+^j} a_j(\langle \mathbf{w}_j, \mathbf{x}_i\rangle + b_j)\right|\right) \\ \text{s.t.} \quad & \langle \mathbf{w}_j, \mathbf{x}_i\rangle + b_j \le 0, \qquad j \in [k], i \in P_-^j, \\ & \langle \mathbf{w}_j, \mathbf{x}_i\rangle + b_j \ge 0, \qquad j \in [k], i \in P_+^j. \end{aligned} \tag{5.3}$$

We now show that this problem can be solved in XP-time with respect to d. As argued in the introduction of this chapter, we may assume without loss of generality that the affine hull of the data points $\mathbf{x}_i$, $i \in [n]$, is the whole space $\mathbb{R}^d$ because, otherwise, we could solve the problem within a lower-dimensional affine subspace. We first show that this implies that the feasible region $P \subseteq \mathbb{R}^{kd+k}$ of (5.3) is *pointed*, that is, it has at least one vertex. More precisely, we show that the zero vector $\mathbf{0}_{kd+k}$ is a vertex of P. To do so, we need to show that it satisfies $kd+k$ linearly independent constraints of (5.3) with equality. Since $\mathbf{0}_{kd+k}$ satisfies every constraint of (5.3) with equality, we only need to show that there exist $kd+k$ linearly independent rows. We write $\mathbf{r}_{ij} := (\mathbf{0}_{d(j-1)}, \mathbf{x}_i, \mathbf{0}_{d(k-j)}, \mathbf{e}_j) \in \mathbb{R}^{kd+k}$, $i \in [n]$, $j \in [k]$, for the kn rows of the constraint matrix, where $\mathbf{e}_j \in \{0,1\}^k$ is the j-th unit vector. By our assumption that the affine hull of the data points is the whole space $\mathbb{R}^d$, there exists a subset $S \subseteq [n]$ of $d+1$ indices such that the $d+1$ vectors $\mathbf{x}_i$, $i \in S$, are affinely independent. This implies that, for each fixed j, the $d+1$ rows $\mathbf{r}_{ij}$ are linearly independent. Moreover, since, for $j_1 \neq j_2$ and arbitrary $i_1, i_2 \in [n]$, two rows $\mathbf{r}_{i_1 j_1}$ and $\mathbf{r}_{i_2 j_2}$ have non-zero entries only in distinct columns, it follows that the $kd+k$ rows $\mathbf{r}_{ij}$, $i \in S$, $j \in [k]$, are linearly independent. Hence, P is pointed.

Next, we divide the feasible region P of (5.3) into several polyhedral pieces, depending on the sign of the prediction error at each data point. Let $\mathbf{s} = (s_i)_{i\in[n]} \in \{-1,1\}^n$ be a sign vector and let

$$P(\mathbf{s}) := \left\{ (\mathbf{w}_1,\dots,\mathbf{w}_k, b_1,\dots,b_k) \in P \;\middle|\; \forall i \in [n]\colon \; s_i\left(y_i - \sum_{j\colon i\in P_+^j} a_j(\langle \mathbf{w}_j, \mathbf{x}_i\rangle + b_j)\right) \ge 0 \right\}$$

be the subset of the feasible region P for which the sign of the prediction error for each data point $\mathbf{x}_i$ coincides with s_i. Since P is pointed, $P(\mathbf{s})$ must be pointed as well. Moreover, by definition, the prediction error of every data point has a fixed sign within $P(\mathbf{s})$, implying that the objective function of (5.3) (as a sum of concave functions) is concave within $P(\mathbf{s})$ (compare Figure 5.4). In addition, the objective value is trivially bounded from below by 0. Since the minimum of a bounded (from below), concave function over a pointed, nonempty polyhedral set is always attained by a vertex [Ben95], it follows that $P(\mathbf{s})$ is either empty or must have a vertex minimizing the loss within $P(\mathbf{s})$. Since $P = \bigcup_{\mathbf{s}\in\{-1,1\}^n} P(\mathbf{s})$, it follows that the optimal solution of (5.3) must be a vertex

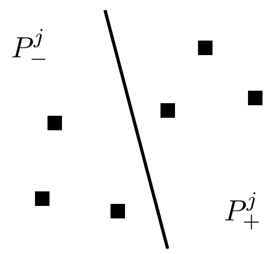


Figure 5.3.: Two-dimensional illustration of a hyperplane partition of the data points into $P_+^j = \{i \in [n] \mid \langle \mathbf{w}_j, \mathbf{x}_i \rangle + b_j \geq 0\}$ and $P_-^j = \{i \in [n] \mid \langle \mathbf{w}_j, \mathbf{x}_i \rangle + b_j < 0\}$.

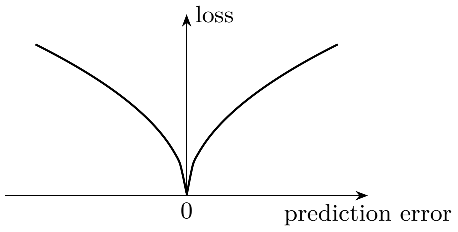


Figure 5.4.: The contribution of a data point $\mathbf{x}_i$ to the objective function is not globally concave. However, it is concave if the sign of the prediction error $y_i - \sum_{j\colon i \in P_+^j} a_j(\langle \mathbf{w}_j, \mathbf{x}_i \rangle + b_j)$ is fixed.

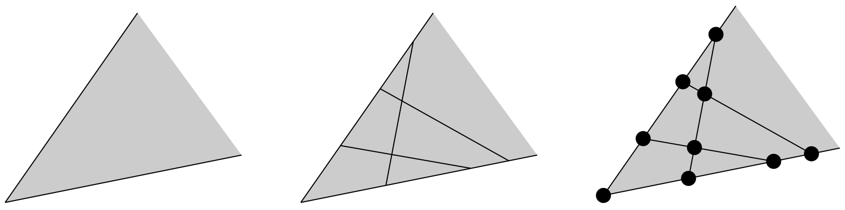


Figure 5.5.: Schematic illustration of how an optimal solution to the subproblem (5.3) can be found. The feasible region is a pointed polyhedral cone (left). The hyperplanes where the prediction error at a certain data point equals zero subdivide P into the regions $P(\mathbf{s})$ (middle). Since the objective function is concave in each of these regions, it suffices to check the vertices of all regions (right).

of one of the polyhedral sets $P(\mathbf{s})$. Hence, it suffices to enumerate all these vertices. Compare Figure 5.5 for a schematic illustration of this idea. Each vertex of one of the polyhedra $P(\mathbf{s})$ is given by $kd + k$ linearly independent inequalities that hold with equality. For selecting these $kd + k$ equations, we have the choice between a total of $kn + n$ equations: the kn constraints of (5.3) as well as the n equations corresponding to the sign constraints defined by $\mathbf{s}$. Note that these n equations are the same for each $\mathbf{s}$ although the inequalities are different.

We conclude that it suffices to check all $\binom{kn+n}{kd+k} \leq (nk)^{\mathcal{O}(dk)}$ possible subsets of $kd + k$ equations. If the chosen equations are linearly independent, then we can determine the corresponding unique solution and check whether it is a feasible solution to (5.3). For each chosen set of equations, these steps can be done in $\operatorname{poly}(n, d, k)$ time. Among all feasible solutions found that way, we take the best one. Consequently, each of the (at most) $2^k n^{dk}$ subproblems can be solved in $(nk)^{\mathcal{O}(dk)} \operatorname{poly}(n, d, k)$ time, resulting in the claimed overall running time. $\square$

In comparison to the algorithm for convex loss functions [Aro+18], our algorithm for concave loss functions requires more time to solve the $\mathcal{O}(2^k n^{dk})$ many subproblems, namely $(nk)^{\mathcal{O}(dk)} \operatorname{poly}(n, d, k)$ instead of $\operatorname{poly}(n, d, k)$ time each. This aligns with the general theme in optimization that convex problems are easier to solve than non-convex problems. However, due to the combinatorial search, both cases result in an XP overall running time.

5.5. Future Research

Summarizing the results of this chapter, we conclude that low data dimensionality is seemingly only of limited use when striving for exact training algorithms. While XP-algorithms are possible for all ℓ^p losses with $p \in [0, \infty]$, there is no hope to obtain FPT-algorithms for $p \in [0, \infty[$. Based on these findings, we identify the following directions for future research.

First, confronting inapproximability results for polynomial-time algorithms (see, e.g., Goel et al. [Goe+21]) and our W[1]-hardness result for exact algorithms (Theorem 5.1), a natural follow-up question is: Can acceptable worst-case approximation ratios be obtained in FPT-time?

Moreover, since FPT results for the parameter dimension alone are out of reach, it would be interesting to identify other suitable parameters for which FPT-algorithms are possible. For example, parameterizing by some "distance from triviality" measure (e.g. assuming specially structured input data) might be an interesting approach [Nie06].

In light of the polynomial-time algorithm for training a single neuron ($k = 1$) in the zero-error case (compare Goel et al. [Goe+21] and Section 5.3), it would be desirable to understand the parameterized complexity of k-ReLU(ℓ) with respect to d for $k \geq 2$ in the zero-error case as well. NP-hardness has been shown by Bakshi, Jayaram, and Woodruff [BJW19] and Goel et al. [Goe+21]. The next step would be to investigate whether FPT-algorithms are possible.

Finally, while it is intuitive that deeper architectures are even more difficult to train, it could be interesting to confirm this intuition from a theoretical perspective by providing a solid study of the (parameterized) complexity of training deeper NNs (≥ 3 layers) with different architectures. A first step in that direction was performed by Boob, Dey, and Lan [BDL20], who investigate NNs with two hidden layers where the second hidden layer consists of only one neuron.

6 Concluding Remarks

"Modelle sind auch nur Meinungen, die sich als Mathematik verkleidet haben."
"Models are just opinions disguised as mathematics."

From "QualityLand" by Marc-Uwe Kling [Kli17]

Nowadays, artificial neural networks are mostly used as a black box and their stunning success in a wide range of application areas often appears to be magic. At the same time, they conquer more and more domains where transparency and regulation of the used algorithms are absolutely necessary. This includes, for example, safety-critical applications in the health sector or in autonomous driving, as well as avoiding biases and discrimination against groups of people. Only a better mathematical understanding of the used technologies can make it possible to exploit the high performances of these methods, while, at the same time, maintaining explainability, safety, and fairness.

In times of massive computing power, state-of-the-art architectures reach an impressive level of complexity. To tackle the challenge of looking behind the scenes of these models, we provided three different perspectives, our *facets of complexity* for ReLU neural networks. We made progress concerning the questions which functions one can represent with certain architectures, how powerful neural networks are as a model of computation, and how difficult it is to train them.

Still, this thesis can only be a modest attempt to uncover some of the magic around the success of modern artificial neural networks. Many exciting open questions remain, both immediately related to our results as discussed at the end of each chapter, as well as further away on the long path towards a solid theoretical understanding of neural networks. What would life be without magic?

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[AB99] M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, 1999.

[AK91] M. M. Ali and F. Kamoun. "A neural network approach to the maximum flow problem". In: *IEEE Global Telecommunications Conference GLOBECOM'91: Countdown to the New Millennium. Conference Record.* 1991, pp. 130–134.

[AKM21] M. Abrahamsen, L. Kleist, and T. Miltzow. "Training Neural Networks is ER-complete". In: *Advances in Neural Information Processing Systems (NeurIPS).* Full version: *arXiv:2102.09798.* 2021.

[Alf+20] M. Alfarra, A. Bibi, H. Hammoud, M. Gaafar, and B. Ghanem. "On the Decision Boundaries of Deep Neural Networks: A Tropical Geometry Perspective". In: *arXiv:2002.08838* (2020).

[ALW17] A. M. Alvarez, Q. Louveaux, and L. Wehenkel. "A machine learning-based approximation of strong branching". In: *INFORMS Journal on Computing* 29.1 (2017), pp. 185–195.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Upper Saddle River, New Jersey, USA: Prentice Hall, 1993.

[And+20] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma. "Strong mixed-integer programming formulations for trained neural networks". In: *Mathematical Programming* 183 (2020), pp. 3–39.

[Aro+18] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee. "Understanding Deep Neural Networks with Rectified Linear Units". In: *International Conference on Learning Representations (ICLR).* 2018.

[Bar93] A. R. Barron. "Universal approximation bounds for superpositions of a sigmoidal function". In: *IEEE Transactions on Information Theory* 39.3 (1993), pp. 930–945.

[Bar94] A. R. Barron. "Approximation and estimation bounds for artificial neural networks". In: *Machine Learning* 14.1 (1994), pp. 115–133.

[BDL20] D. Boob, S. S. Dey, and G. Lan. "Complexity of training ReLU neural network". In: *Discrete Optimization* (2020). In Press.

[Bel+16] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. "Neural Combinatorial Optimization with Reinforcement Learning". In: *arXiv:1611.09940* (2016).

[Bel62] R. Bellman. "Dynamic programming treatment of the travelling salesman problem". In: *Journal of the ACM* 9.1 (1962), pp. 61–63.

[Ben95] H. P. Benson. “Concave Minimization: Theory, Applications and Algorithms”. In: *Handbook of Global Optimization.* Springer, 1995, pp. 43–148.

[BFN19] Y. Bartal, N. Fandina, and O. Neiman. “Dimensionality reduction: theoretical perspective on practical measures”. In: *Advances in Neural Information Processing Systems (NeurIPS).* 2019.

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. “The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems”. In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21.

[BJW19] A. Bakshi, R. Jayaram, and D. P. Woodruff. “Learning Two Layer Rectified Neural Networks in Polynomial Time”. In: *Conference on Learning Theory (COLT).* 2019.

[BLP21] Y. Bengio, A. Lodi, and A. Prouvost. “Machine learning for combinatorial optimization: A methodological tour d’horizon”. In: *European Journal of Operational Research* 290.2 (2021), pp. 405–421.

[BLZ18] P. Bonami, A. Lodi, and G. Zarpellon. “Learning a classification of mixed-integer quadratic programming problems”. In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research.* Springer. 2018, pp. 595–604.

[BT96] V. Beiu and J. G. Taylor. “On the circuit complexity of sigmoid feedforward neural networks”. In: *Neural Networks* 9.7 (1996), pp. 1155–1171.

[BW05] D. Bertsimas and R. Weismantel. *Optimization over Integers.* Belmont, MA: Dynamic Ideas, 2005.

[Cap+21] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković. “Combinatorial optimization and reasoning with graph neural networks”. In: *arXiv:2102.09544* (2021).

[CCN20] T. Chin, Z. Cai, and F. Neumann. “Robust Fitting in Computer Vision: Easy or Hard?” In: *International Journal of Computer Vision* 128.3 (2020), pp. 575–587.

[CKM20] S. Chen, A. R. Klivans, and R. Meka. “Learning Deep ReLU Networks Is Fixed-Parameter Tractable”. In: *arXiv:2009.13512* (2020).

[Cla73] A. C. Clarke. “Hazards of Prophecy: The failure of imagination”. In: *Profiles of the Future: An Enquiry into the Limits of the Possible.* 2nd edition. Popular Library, 1973.

[CM18] V. Charisopoulos and P. Maragos. “A tropical approach to neural networks with piecewise linear activations”. In: *arXiv:1805.08749* (2018).

[Cor+01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* 2nd edition. MIT Press, 2001.

[Cyb89] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[Cyg+15] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms.* Springer, 2015.

[DF13] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity.* Springer, 2013.

[Dia+20] I. Diakonikolas, S. Goel, S. Karmalkar, A. R. Klivans, and M. Soltanolkotabi. "Approximation Schemes for ReLU Regression". In: *Conference on Learning Theory (COLT).* 2020.

[Din70] E. A. Dinic. "Algorithm for solution of a problem of maximum flow in a network with power estimation". In: *Soviet Mathematics Doklady* 11 (1970), pp. 1277–1280.

[DWX20] S. S. Dey, G. Wang, and Y. Xie. "Approximation Algorithms for Training One-Node ReLU Neural Networks". In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 6696–6706.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Science & Business Media, 1987.

[EK72] J. Edmonds and R. M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM* 19.2 (1972), pp. 248–264.

[ER08] S. Effati and M Ranjbar. "Neural network models for solving the maximum flow problem". In: *Applications and Applied Mathematics* 3.3 (2008), pp. 149–162.

[ER18] P. Emami and S. Ranka. "Learning Permutations with Sinkhorn Policy Gradient". In: *arXiv:1805.07010* (2018).

[ES16] R. Eldan and O. Shamir. "The power of depth for feedforward neural networks". In: *Conference on Learning Theory (COLT).* 2016.

[FGK16] S. Fomin, D. Grigoriev, and G. Koshevoy. "Subtraction-free complexity, cluster transformations, and spanning trees". In: *Foundations of Computational Mathematics* 16.1 (2016), pp. 1–31.

[FGS20] F. V. Fomin, P. A. Golovach, and K. Simonov. "Parameterized Complexity of PCA". In: *17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT).* 2020.

[FHN21] V. Froese, C. Hertrich, and R. Niedermeier. "The Computational Complexity of ReLU Network Training Parameterized by Data Dimensionality". In: *arXiv:2105.08675* (2021).

[FJ17] M. Fischetti and J. Jo. "Deep neural networks as 0-1 mixed integer linear programs: A feasibility study". In: *arXiv:1712.06174* (2017).

[FL10] M. Fischetti and A. Lodi. "On the knapsack closure of 0-1 Integer Linear Programs". In: *Electronic Notes in Discrete Mathematics* 36 (2010), pp. 799–804.

[Gas+19] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. "Exact combinatorial optimization with graph convolutional neural networks". In: *arXiv:1906.01629* (2019).

[GBB11] X. Glorot, A. Bordes, and Y. Bengio. “Deep sparse rectifier neural networks”. In: *International Conference on Artificial Intelligence and Statistics (AISTATS).* 2011.

[GEB15] L. A. Gatys, A. S. Ecker, and M. Bethge. “A neural algorithm of artistic style”. In: *arXiv:1508.06576* (2015).

[GFS07] A. Graves, S. Fernández, and J. Schmidhuber. “Multi-dimensional recurrent neural networks”. In: *International Conference on Artificial Neural Networks.* 2007, pp. 549–558.

[GGT89] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. “A fast parametric maximum flow algorithm and applications”. In: *SIAM Journal on Computing* 18.1 (1989), pp. 30–55.

[GH18] S. Gu and T. Hao. “A pointer network based deep learning algorithm for 0–1 Knapsack Problem”. In: *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI).* 2018, pp. 473–477.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[GK19] S. Goel and A. R. Klivans. “Learning neural networks with two nonlinear layers in polynomial time”. In: *Conference on Learning Theory (COLT).* 2019.

[GKM18] S. Goel, A. Klivans, and R. Meka. “Learning one convolutional layer with overlapping patches”. In: *International Conference on Machine Learning (ICML).* 2018.

[GKR09] P. Giannopoulos, C. Knauer, and G. Rote. “The Parameterized Complexity of Some Geometric Problems in Unbounded Dimension”. In: *Parameterized and Exact Computation, 4th International Workshop (IWPEC).* 2009.

[Goe+17] S. Goel, V. Kanade, A. Klivans, and J. Thaler. “Reliably learning the ReLU in polynomial time”. In: *Conference on Learning Theory (COLT).* 2017.

[Goe+21] S. Goel, A. R. Klivans, P. Manurangsi, and D. Reichman. “Tight Hardness Results for Training Depth-2 ReLU Networks”. In: *12th Innovations in Theoretical Computer Science Conference (ITCS).* 2021.

[Gri+21] R. Gribonval, G. Kutyniok, M. Nielsen, and F. Voigtlaender. “Approximation spaces of deep neural networks”. In: *Constructive Approximation* (2021), pp. 1–109.

[GSS82] L. M. Goldschlager, R. A. Shaw, and J. Staples. “The maximum flow problem is log space complete for P”. In: *Theoretical Computer Science* 21.1 (1982), pp. 105–111.

[Gur21] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual.* 2021. URL: `http://www.gurobi.com`.

[Han19] B. Hanin. “Universal function approximation by deep neural nets with bounded width and ReLU activations”. In: *Mathematics* 7.10 (2019), p. 992.

[HDE14] H. He, H. Daumé III, and J. M. Eisner. “Learning to search in branch and bound algorithms”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2014.

[Her+21] C. Hertrich, A. Basu, M. Di Summa, and M. Skutella. “Towards Lower Bounds on the Depth of ReLU Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Full version: *arXiv:2105.14835*. 2021.

[HK62] M. Held and R. M. Karp. “A dynamic programming approach to sequencing problems”. In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), pp. 196–210.

[Hoc97] D. S. Hochbaum. “Various Notions of Approximations: Good, Better, Best, and More”. In: *Approximation Algorithms for NP-hard Problems*. Ed. by D. S. Hochbaum. PWS Publishing Co., 1997, pp. 346–446.

[Hor91] K. Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural networks* 4.2 (1991), pp. 251–257.

[HR19] B. Hanin and D. Rolnick. “Complexity of Linear Regions in Deep Networks”. In: *International Conference on Machine Learning (ICML)*. 2019.

[HS17] B. Hanin and M. Sellke. “Approximating continuous functions by ReLU nets of minimal width”. In: *arXiv:1710.11278* (2017).

[HS21a] C. Hertrich and L. Sering. “ReLU Neural Networks of Polynomial Size for Exact Maximum Flow Computation”. In: *arXiv:2102.06635* (2021).

[HS21b] C. Hertrich and M. Skutella. “Provably Good Solutions to the Knapsack Problem via Neural Networks of Bounded Size”. In: *AAAI Conference on Artificial Intelligence*. Full version: *arXiv:2005.14105*. 2021.

[HT85] J. J. Hopfield and D. W. Tank. ““Neural” computation of decisions in optimization problems”. In: *Biological Cybernetics* 52.3 (1985), pp. 141–152.

[HUL93a] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms I*. Vol. 305. Grundlehren der Mathematischen Wissenschaften. Berlin: Springer-Verlag, 1993.

[HUL93b] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms II*. Vol. 306. Grundlehren der Mathematischen Wissenschaften. Berlin: Springer-Verlag, 1993.

[IP01] R. Impagliazzo and R. Paturi. “On the Complexity of k-SAT”. In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375.

[JGH18] A. Jacot, F. Gabriel, and C. Hongler. “Neural tangent kernel: convergence and generalization in neural networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[Jos21] M. Joswig. *Essentials of tropical combinatorics*. Vol. 219. Graduate Studies in Mathematics. Providence, RI: American Mathematical Society, 2021.

[JS19] S. Jukna and H. Seiwert. “Greedy can beat pure dynamic programming”. In: *Information Processing Letters* 142 (2019), pp. 90–95.

[JS82] M. Jerrum and M. Snir. “Some exact complexity results for straight-line computations over semirings”. In: *Journal of the ACM* 29.3 (1982), pp. 874–897.

[Juk15] S. Jukna. “Lower bounds for tropical circuits and dynamic programs”. In: *Theory of Computing Systems* 57.1 (2015), pp. 160–194.

[Kar72] R. M. Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations*. Ed. by R. E. Miller and J. W. Thatcher. Springer, 1972, pp. 85–103.

[KC88] M. P. Kennedy and L. O. Chua. “Neural networks for nonlinear programming”. In: *IEEE Transactions on Circuits and Systems* 35.5 (1988), pp. 554–562.

[Kha+16] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. “Learning to branch in mixed integer programming”. In: *AAAI Conference on Artificial Intelligence*. 2016.

[Kha+17a] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. “Learning Combinatorial Optimization Algorithms over Graphs”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[Kha+17b] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. “Learning to Run Heuristics in Tree Search.” In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2017.

[KHW19] W. Kool, H. van Hoof, and M. Welling. “Attention, Learn to Solve Routing Problems!” In: *International Conference on Learning Representations (ICLR)*. 2019.

[KKW15] C. Knauer, S. König, and D. Werner. “Fixed-Parameter Complexity and Approximability of Norm Maximization”. In: *Discrete & Computational Geometry* 53.2 (2015), pp. 276–295.

[Kli17] M.-U. Kling. *QualityLand*. Berlin: Ullstein Buchverlage GmbH, 2017.

[KLP17] M. Kruber, M. E. Lübbecke, and A. Parmentier. “Learning when to use a decomposition”. In: *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 2017, pp. 202–210.

[KP03] V. Kaibel and M. E. Pfetsch. “Some Algorithmic Problems in Polytope Theory”. In: *Algebra, Geometry and Software Systems*. Ed. by M. Joswig and N. Takayama. Springer, 2003, pp. 23–47.

[KPP04] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012.

[KT06] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson Education, 2006.

[KV08] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. 4th edition. Springer, 2008.

[Lap+19] S. Lapuschkin, S. Wäldchen, A. Binder, G. Montavon, W. Samek, and K.-R. Müller. "Unmasking Clever Hans predictors and assessing what machines really learn". In: *Nature communications* 10.1 (2019), pp. 1–8.

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521 (2015), pp. 436–444.

[LeC+89] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. "Handwritten digit recognition with a back-propagation network". In: *Advances in Neural Information Processing Systems (NeurIPS).* 1989.

[Lei91] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes.* Morgan-Kaufmann, 1991.

[Li+21] D. Li, J. Liu, D. Lee, A. Seyedmazloom, G. Kaushik, K. Lee, and N. Park. "A Novel Method to Solve Neural Knapsack Problems". In: *International Conference on Machine Learning (ICML).* 2021.

[LM18] M. Lombardi and M. Milano. "Boosting combinatorial problem modeling with machine learning". In: *International Joint Conference on Artificial Intelligence (IJCAI).* 2018.

[LS17] S. Liang and R. Srikant. "Why deep neural networks for function approximation?" In: *International Conference on Learning Representations (ICLR).* 2017.

[LZ17] A. Lodi and G. Zarpellon. "On learning and branching: a survey". In: *TOP* 25.2 (2017), pp. 207–236.

[MB17] A. Mukherjee and A. Basu. "Lower bounds over Boolean inputs for deep neural networks with ReLU gates". In: *arXiv:1711.03073* (2017).

[McC99] S. T. McCormick. "Fast algorithms for parametric scheduling come from extensions to parametric maximum flow". In: *Operations Research* 47.5 (1999), pp. 744–756.

[MCT21] P. Maragos, V. Charisopoulos, and E. Theodosis. "Tropical Geometry and Machine Learning". In: *Proceedings of the IEEE* 109.5 (2021), pp. 728–755.

[Mha96] H. N. Mhaskar. "Neural networks for optimal approximation of smooth and analytic functions". In: *Neural Computation* 8.1 (1996), pp. 164–177.

[MM95] H. N. Mhaskar and C. A. Micchelli. "Degree of approximation by neural and translation networks with a single hidden layer". In: *Advances in Applied Mathematics* 16.2 (1995), pp. 151–183.

[Mon+14] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio. "On the Number of Linear Regions of Deep Neural Networks". In: *Advances in Neural Information Processing Systems (NeurIPS).* 2014.

[MP43] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[MPH09] L. van der Maaten, E. Postma, and J. van der Herik. “Dimensionality Reduction: A Comparative Review”. In: *Tilburg University* TiCC TR 2009-005 (2009).

[MRZ21] G. Montúfar, Y. Ren, and L. Zhang. “Sharp bounds for the number of regions of maxout networks and vertices of Minkowski sums”. In: *arXiv:2104.08135* (2021).

[MS15] D. Maclagan and B. Sturmfels. *Introduction to Tropical Geometry.* Vol. 161. Graduate Studies in Mathematics. American Mathematical Soc., 2015.

[MT90] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, 1990.

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.

[NMH18] Q. Nguyen, M. C. Mukkamala, and M. Hein. “Neural Networks Should Be Wide Enough to Learn Disconnected Decision Regions”. In: *International Conference on Machine Learning (ICML).* 2018.

[NO12] A. Nazemi and F. Omidi. “A capable neural network model for solving the maximum flow problem”. In: *Journal of Computational and Applied Mathematics* 236.14 (2012), pp. 3498–3513.

[Now+17] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. “Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks”. In: *arXiv:1706.07450* (2017).

[OMK20] D. W. Otter, J. R. Medina, and J. K. Kalita. “A survey of the usages of deep learning for natural language processing”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.2 (2020), pp. 604–624.

[OPS93] M. Ohlsson, C. Peterson, and B. Söderberg. “Neural Networks for Optimization Problems with Inequality Constraints: The Knapsack Problem”. In: *Neural Computation* 5.2 (1993), pp. 331–339.

[Orl13] J. B. Orlin. “Max Flows in O(nm) Time, or Better”. In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing (STOC).* 2013.

[PGM94] I. Parberry, M. R. Garey, and A. Meyer. *Circuit Complexity and Neural Networks.* MIT Press, 1994.

[Pin99] A. Pinkus. “Approximation theory of the MLP model”. In: *Acta Numerica* 8 (1999), pp. 143–195.

[PMB14] R. Pascanu, G. Montufar, and Y. Bengio. “On the number of inference regions of deep feed forward networks with piece-wise linear activations”. In: *International Conference on Learning Representations (ICLR).* 2014.

[PS15] G. Y. Panina and I. Streĭnu. “Virtual polytopes”. In: *Uspekhi Mat. Nauk* 70.6(426) (2015), pp. 139–202.

[Rag+17] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein. “On the Expressive Power of Deep Neural Networks”. In: *International Conference on Machine Learning (ICML).* 2017.

[Ros58] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6 (1958), pp. 386–408.

[Sag20] Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*. `https://www.sagemath.org`. 2020.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. New York: John Wiley and Sons, 1986.

[Sim+19] K. Simonov, F. V. Fomin, P. A. Golovach, and F. Panolan. "Refined Complexity of PCA with Outliers". In: *International Conference on Machine Learning (ICML)*. 2019.

[SKR20] T. Serra, A. Kumar, and S. Ramalingam. "Lossless compression of deep neural networks". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2020, pp. 417–430.

[Smi99] K. A. Smith. "Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research". In: *INFORMS Journal on Computing* 11.1 (1999), pp. 15–34.

[SR20] T. Serra and S. Ramalingam. "Empirical bounds on linear regions of deep rectifier networks". In: *AAAI Conference on Artificial Intelligence*. 2020.

[SS17] I. Safran and O. Shamir. "Depth-width tradeoffs in approximating natural functions with neural networks". In: *International Conference on Machine Learning (ICML)*. 2017.

[SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[ST94] J. H. Silverman and J. Tate. *Rational Points on Elliptic Curves*. Springer, 1994.

[STAK92] J. S. Shawe-Taylor, M. H. Anthony, and W. Kern. "Classes of feedforward neural networks and their circuit complexity". In: *Neural Networks* 5.6 (1992), pp. 971–977.

[STR18] T. Serra, C. Tjandraatmadja, and S. Ramalingam. "Bounding and Counting Linear Regions of Deep Neural Networks". In: *International Conference on Machine Learning (ICML)*. 2018.

[SY10] A. Shpilka and A. Yehudayoff. *Arithmetic Circuits: A Survey of Recent Results and Open Questions*. Now Publishers Inc, 2010.

[Tel15] M. Telgarsky. "Representation Benefits of Deep Feedforward Networks". In: *arXiv:1509.08101* (2015).

[Tel16] M. Telgarsky. "Benefits of depth in neural networks". In: *Conference on Learning Theory (COLT)*. 2016.

[Tur50] A. M. Turing. "Computing machinery and intelligence". In: *Mind* LIX.236 (1950), pp. 433–460.

[Var+21] G. Vardi, D. Reichman, T. Pitassi, and O. Shamir. "Size and depth separation in approximating benign functions with neural networks". In: *arXiv:2102.00314* (2021).

[Vaz01] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[Vel+20] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. "Neural execution of graph algorithms". In: *International Conference on Learning Representations (ICLR)*. 2020.

[VFJ15] O. Vinyals, M. Fortunato, and N. Jaitly. "Pointer Networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2015.

[Vid+17] R. Vidal, J. Bruna, R. Giryes, and S. Soatto. "Mathematics of deep learning". In: *arXiv:1712.04741* (2017).

[Wan04] S. Wang. "General constructive representations for continuous piecewise-linear functions". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 51.9 (2004), pp. 1889–1896.

[Wil19] D. P. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019.

[Woe00] G. J. Woeginger. "When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)?" In: *INFORMS Journal on Computing* 12.1 (2000), pp. 57–74.

[WS05] S. Wang and X. Sun. "Generalization of hinging hyperplanes". In: *IEEE Transactions on Information Theory* 51.12 (2005), pp. 4425–4431.

[WS11] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[Xu+20a] K. Xu, J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka. "What can neural networks reason about?" In: *International Conference on Learning Representations (ICLR)*. 2020.

[Xu+20b] S. Xu, S. S. Panwar, M. S. Kodialam, and T. V. Lakshman. "Deep Neural Network Approximated Dynamic Programming for Combinatorial Optimization". In: *AAAI Conference on Artificial Intelligence*. 2020.

[Yan+18] F. Yang, T. Jin, T.-Y. Liu, X. Sun, and J. Zhang. "Boosting Dynamic Programming with Neural Networks for Solving NP-hard Problems". In: *Asian Conference on Machine Learning (ACML)*. 2018, pp. 726–739.

[Yar17] D. Yarotsky. "Error bounds for approximations with deep ReLU networks". In: *Neural Networks* 94 (2017), pp. 103–114.

[Zie01] M. Ziegelmann. "Constrained Shortest Paths and Related Problems". PhD thesis. Universität des Saarlandes Saarbrücken, 2001.

[ZNL18] L. Zhang, G. Naitzat, and L.-H. Lim. "Tropical Geometry of Deep Neural Networks". In: *International Conference on Machine Learning (ICML)*. 2018.