The Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI-21)

Provably Good Solutions to the Knapsack Problem via Neural Networks of Bounded Size *

Christoph Hertrich† **and Martin Skutella**
Institute of Mathematics, Technische Universität Berlin
hertrich@math.tu-berlin.de, martin.skutella@tu-berlin.de

Abstract

The development of a satisfying and rigorous mathematical understanding of the performance of neural networks is a major challenge in artificial intelligence. Against this background, we study the expressive power of neural networks through the example of the classical NP-hard Knapsack Problem. Our main contribution is a class of recurrent neural networks (RNNs) with rectified linear units that are iteratively applied to each item of a Knapsack instance and thereby compute optimal or provably good solution values. We show that an RNN of depth four and width depending quadratically on the profit of an optimum Knapsack solution is sufficient to find optimum Knapsack solutions. We also prove the following tradeoff 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. Our results build upon a classical dynamic programming formulation of the Knapsack Problem as well as a careful rounding of profit values that are also at the core of the well-known fully polynomial-time approximation scheme for the Knapsack Problem. Finally, we point out that our results can be generalized to many other combinatorial optimization problems that admit dynamic programming solution methods, such as various Shortest Path Problems, the Longest Common Subsequence Problem, and the Traveling Salesperson Problem.

1 Introduction

Deep learning and neural networks (NNs) are at the heart of some of the greatest advances in modern computer science. They enable huge breakthroughs in applications like computer vision, translation, speech recognition, and autonomous driving, to name just a few; see, e.g., LeCun, Bengio, and Hinton (2015). While numerous computational studies present impressive empirical proof of neural networks' computational power, we are still far away from a more rigorous theoretical explanation of these observations.

Apart from the popular applications named above, it has been shown that NNs have high potential for practically solving combinatorial optimization (CO) problems or empirically improving classical solution methods (Bengio, Lodi, and Prouvost 2018). For example, Yang et al. (2018) and Xu et al. (2020) utilize NNs in order to empirically enhance *dynamic programming*, a very classical CO method. While the methods used in these papers indeed provide fast and empirically near-optimal solutions, their use of NNs makes it virtually impossible to give theoretical optimality or worst-case approximation guarantees. Motivated by this imbalance, and focusing on the Knapsack Problem, which is a prime example of CO problems that can be solved via dynamic programming, we investigate the following question:

Which neural network size is theoretically sufficient to find solutions of provable quality for the Knapsack Problem?

We give an answer to this question by presenting a class of carefully constructed NNs with provable quality guarantees. In addition, we argue that our approach is not at all specific for the Knapsack Problem, but can be generalized to many other CO problems, e.g., various Shortest Path Problems, the Longest Common Subsequence Problem, and the Traveling Salesperson Problem.

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 (Karp 1972) 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., Martello and Toth (1990); Kellerer, Pferschy, and Pisinger (2004). 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

*The full version of this article is available on arXiv (Hertrich and Skutella 2020).

†Supported by DFG-GRK 2434 Facets of Complexity.

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 (2005); Fischetti and Lodi (2010). The Knapsack Problem is therefore a natural and important object of study in order to advance our theoretical understanding of neural networks and get closer to a rigorous explanation of their stunning success in so many applications, including miscellaneous optimization problems.

Related work. The idea of using neural networks (NNs) to practically solve CO problems became popular with the work of Hopfield and Tank (1985). 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. Smith (1999) reviews this early stream of research. While most authors mainly focus on the Traveling Salesperson Problem (TSP), Ohlsson, Peterson, and Söderberg (1993) study a so-called mean field NN for (generalizations of) the Knapsack Problem and empirically assess the quality of its solutions.

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. Bengio, Lodi, and Prouvost (2018) review these developments from a practical perspective. 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 (2017) for a survey. Machine learning has also been applied to modeling aspects of CO, as reviewed by Lombardi and Milano (2018), and to several specific CO problems, where the TSP is often one of them (Vinyals, Fortunato, and Jaitly 2015; Bello et al. 2016; Khalil et al. 2017; Nowak et al. 2017; Emami and Ranka 2018; Kool, van Hoof, and Welling 2019). 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. For example, Bello et al. (2016) utilize an RNN trained by reinforcement learning and present a computational study demonstrating their approach's empirical effectiveness for the TSP and the Knapsack Problem. Particularly related to our work, Yang et al. (2018) and Xu et al. (2020) 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.

The recent success of deep neural networks has also triggered a lot of research on their expressivity. As we do in this paper, many authors focus on the simple but practically powerful model of feedforward NNs with activations in the form of rectified linear units (ReLU). Since Glorot, Bordes, and Bengio (2011) corroborated their empirical success, such ReLU NNs have been established as a standard model in Machine Learning within the past decade. ReLU NNs can compute any continuous piecewise linear function (Goodfellow et al. 2013; Arora et al. 2018). This fact implies universal approximation properties. A variety of results has been achieved on depth vs. width tradeoffs (Telgarsky 2015; Eldan and Shamir 2016; Telgarsky 2016; Hanin and Sellke 2017; Liang and Srikant 2017; Safran and Shamir 2017; Yarotsky 2017; Arora et al. 2018; Nguyen, Mukkamala, and Hein 2018; Hanin 2019). Closely related are investigations concerning the number and structure of linear regions that NNs with certain size and depth may have (Montufar et al. 2014; Pascanu, Montufar, and Bengio 2014; Raghu et al. 2017; Hanin and Rolnick 2019). Serra, Tjandraatmadja, and Ramalingam (2018) use mixed-integer programming for precisely counting the number of such regions. Mukherjee and Basu (2017) prove size lower bounds to represent Boolean functions with NNs of limited depth.

Our contribution. We present a rigorous mathematical study on the expressivity of NNs through the example of the NP-hard Knapsack Problem. To this end, we show that there is a class of feedforward ReLU NNs of bounded size that compute *provably* good solutions to the NP-hard Knapsack Problem. In Section 3, we first present such an NN of depth $\mathcal{O}(n)$ and width $\mathcal{O}((p^*)^2)$ that always finds 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 optimum solution value is found by iteratively applying an RNN of depth four and width $\mathcal{O}((p^*)^2)$ to the n items of a Knapsack instance. As p^* can, e.g., be chosen as the total size of all items, the RNN'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.

In Section 4, we prove that the width of the NNs can be drastically decreased while still obtaining solution values of provable quality in the worst case. We construct 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 P = NP. To the best of our knowledge, our results establish the first rigorous tradeoff between the size of neural networks for CO problems and their worst-case solution quality.

The idea behind our construction of the NNs is to mimic the classical dynamic program for the Knapsack Problem. More precisely, 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. 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 addition, our NNs of fixed width rely 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.

In this paper, the Knapsack Problem mainly serves as a prominent showcase for a novel approach to the rigorous analysis of neural networks' expressivity. This approach is by no means specific for the Knapsack Problem. In Section 5, we discuss how it can be applied to NNs for other combinatorial optimization problems that can be solved via dynamic programming. In Particular, we establish similar results for the Longest Common Subsequence Problem, the Single-Source and All-Pairs Shortest Path Problems, as well as the NP-hard Traveling Salesperson Problem and the Constrained Shortest Path Problem. For the latter problem one can show similar results on the tradeoff between the size of NNs and their solution quality.

2 Preliminaries

Neural networks with rectified linear units. We use definitions and notations similar to Shalev-Shwartz and Ben-David (2014, Chapter 20). A *feedforward neural network with rectified linear units*, abbreviated by ReLU NN, or simply NN, is a finite directed acyclic graph (V, E), 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 $v \in V \setminus V_0$. Here, V_0 is the set of nodes with in-degree zero. The nodes in V are called *neurons*. The *depth* k is the length of a longest path in the graph. In the following we suppose that neurons are grouped into *layers* $V = V_0 \dot\cup V_1 \dot\cup \cdots \dot\cup V_k$ such that the layer index strictly increases along each arc.[1] Further, we assume that V_0 and V_k are precisely the sets of neurons with in-degree and out-degree zero, respectively. Consequently, they are called *input neurons* and *output neurons*, respectively. Neurons in $V \setminus (V_0 \cup V_k)$ are called *hidden neurons*. Let $n_\ell = |V_l|$ be the number of neurons in the ℓ-th layer. The *width* and *size* of the NN are defined to be $\max\{n_1, \ldots, n_{k-1}\}$ and $\sum_{\ell=1}^{k-1} n_\ell$, respectively.

Every NN computes a function $\mathbb{R}^{n_0} \to \mathbb{R}^{n_k}$ as follows. Given an input vector $x \in \mathbb{R}^{n_0}$, we associate an activation $a(v)$ with every neuron $v \in V \setminus V_0$ and an output $o(v)$ with every neuron $v \in V \setminus V_k$. First, the output values $o(v)$, $v \in V_0$, of the n_0 input neurons equal the n_0 components of input vector x. Second, the activation of a neuron $v \in V \setminus V_0$ is the weighted sum of outputs of all predecessors plus its bias, that is, $a(v) = b_v + \sum_{u:\,(u,v)\in E} w_{uv} o(u)$. Third, for each hidden neuron $v \in V \setminus (V_0 \cup V_k)$, the output is determined by $o(v) = \sigma(a(v))$, where σ is the so-called *activation function*. In this paper, σ is always the *rectifier function* $\sigma(z) = \max\{0, z\}$. Neurons having this activation function are called *rectified linear units* (ReLUs). Finally, the output vector $y \in \mathbb{R}^{n_k}$ consists of the n_k activation values $a(v)$ of the n_k output neurons $v \in V_k$. Figure 1 gives an example, which will also be used as a subnetwork in later sections.

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

[1] Some authors only allow connections between successive layers. One can create such a structure by adding additional neurons propagating the values of neurons from former layers through the network. For our purposes, however, it is convenient to omit this restriction.

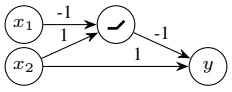

Figure 1: An NN with two input neurons, labeled x_1 and x_2, one hidden neuron, labeled with the shape of the rectifier function, and one output neuron, labeled y. The arcs are labeled with their weights and all biases are zero. The network has depth 2, width 1, and size 1. It computes the function $x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

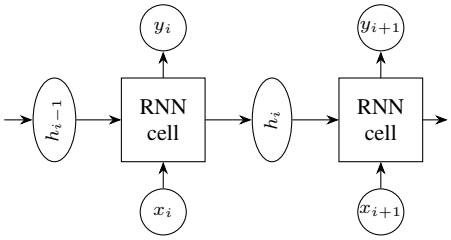

Figure 2: Basic structure of an (unfolded) RNN.

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, 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 the same manner as a feedforward NN described above, it then computes the i-th output vector y_i, as well as, the new hidden state vector h_i. The basic structure of an RNN is shown in Figure 2. Sometimes it holds that $y_i = h_i$, that is, the i-th output is actually equal to the i-th hidden state.

Notations and Algorithms for the Knapsack Problem. An instance of the Knapsack Problem consists of n items $1, 2, \ldots, n$, 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 $\sum_{i\in M} p_i$. Here and in the following, we use $\mathbb{N} \coloneqq \{1, 2, 3, \ldots\}$ to denote the natural numbers (without zero), and for every $k \in \mathbb{N}$, we let $[k] \coloneqq \{1, 2, \ldots, k\}$.

We outline a classical dynamic programming formulation for the Knapsack Problem. Let p^* 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

$$f(p, i) \coloneqq \min\Big\{\sum\nolimits_{j\in M} s_j \;\Big|\; M \subseteq [i],\ \sum\nolimits_{j\in M} p_j \geq p\Big\}$$

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

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

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 $\max\{p \in [p^*] \mid f(p,n) \leq 1\}$, 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, for each $\varepsilon > 0$, one can achieve a feasible solution with guaranteed profit of at least $1 - \varepsilon$ times the optimal profit, while the running time can be bound polynomially in the input size and $1/\varepsilon$. Such a class of algorithms with arbitrary good approximation guarantees is called a *fully polynomial-time approximation scheme* (FPTAS). For more details, we refer to the books by Hochbaum (1997), Vazirani (2001), or Williamson and Shmoys (2011).

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 (2006, Section 6.4). Our construction in Section 3 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.

3 An Exact RNN for the Knapsack Problem

In this section we introduce the *DP-NN*, an NN that exactly executes the dynamic program described in Section 2. In fact, the DP-NN is an RNN that receives the items one by one and computes the state space of the dynamic program for the items seen so far.

Like the dynamic program in Section 2, the DP-NN requires a fixed upper bound p^* on the optimal objective value of the Knapsack Problem. We relax this condition in Section 4, when we investigate how the FPTAS for the Knapsack Problem can be implemented as an NN.

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, overall it has $p^* + 2$ input neurons and p^* output neurons. Figure 3 illustrates the recurrent structure of the NN, which computes the state space of the dynamic program.

In the following it is very important to distinguish fixed parameters of the NN from activation and output values of neurons that depend on the particular Knapsack instance. We denote the latter by bold symbols in order to make the difference visible. Moreover, 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 $n_0 = p^* + 2$ input values 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

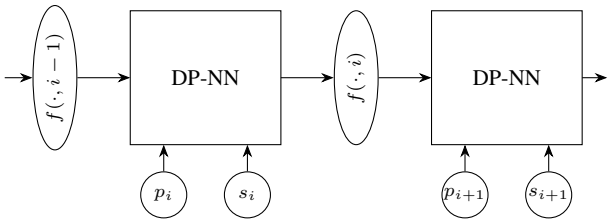

Figure 3: Recurrent structure of the DP-NN to solve the Knapsack Problem.

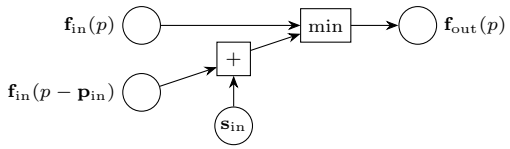

Figure 4: Desirable architecture for computing $\mathbf{f}_{\text{out}}(p)$, $p \in [p^*]$, from the inputs. However, the existence of an edge (nonzero weight) depends critically on the input value $\mathbf{p}_{\text{in}}$, which is not allowed.

$p \in [p^*]$. 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. (1). 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 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 5. 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 if $p' \neq \mathbf{p}_{\text{in}}$.

Due to the integrality of the profit values, it is possible to show that this additional unit can be realized with two hidden layers and a constant number of neurons for every value of $p \in [p^*]$ and $p' \in [p-1]$. 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

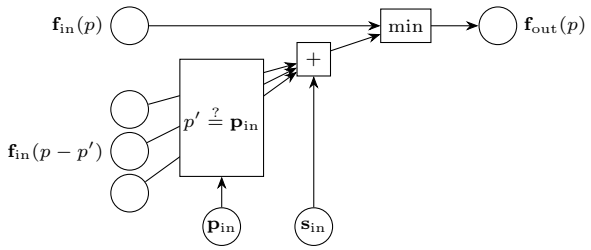

Figure 5: High-level idea how the DP-NN computes $\mathbf{f}_{\text{out}}(p)$ for $p \in [p^*]$ from the inputs.

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 full version of this paper (Hertrich and Skutella 2020) we provide a detailed construction of the DP-NN and prove the following theorem.

Theorem 1. *For a Knapsack instance with capacity $S = 1$, $s_i \in]0, 1]$, and $p_i \in \mathbb{N}$, for $i \in [n]$, with an upper bound p^* on the optimal solution value, the corresponding dynamic programming values $f(p, i)$, $i \in [n]$, $p \in p^*$, can be exactly computed by iteratively applying the DP-NN n times.*

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

4 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. Instead, we prove an approximation ratio (i.e., a worst-case bound) for the solution value computed by the FPTAS-NN. As in the standard Knapsack FPTAS (Hochbaum 1997; Vazirani 2001; Williamson and Shmoys 2011), the idea of this construction is to round the profit values if p^* becomes too large for an exact computation. Our approximation result can be interpreted as a tradeoff between the width of the NN and the quality of the Knapsack solution obtained.

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 a required 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 6 illustrates the recurrent structure of this NN.

As in Section 3, we use bold symbols in order to distinguish input, activation, and output values that depend on the concrete Knapsack instance from fixed parameters of the network. We again drop the index i in order to make the recurrent structure obvious. We denote the $n_0 = P + 3$ input parameters by $\mathbf{g}_{\text{in}}(p)$, for $p \in [P]$, as well as $\mathbf{p}_{\text{in}}^*$, $\mathbf{p}_{\text{in}}$, and $\mathbf{s}_{\text{in}}$. The $P + 1$ output values are denoted by $\mathbf{g}_{\text{out}}(p)$, for

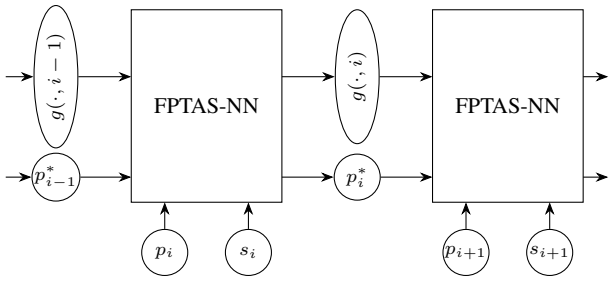

Figure 6: Recurrent structure of the FPTAS-NN for the Knapsack Problem.

$p \in [P]$, and $\mathbf{p}_{\text{out}}^*$. Similar to the DP-NN in Section 3, the basic idea is to implement a recursion of the type

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

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 $p - \mathbf{p}_{\text{in}}$, 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, $\mathbf{p}_{\text{in}}^*$ and $\mathbf{p}_{\text{in}}$ are used in order to calculate the old and new rounding granularities $\mathbf{d}_{\text{old}} = \max\{1, \mathbf{p}_{\text{in}}^*/P\}$ and $\mathbf{d}_{\text{new}} = \max\{1, (\mathbf{p}_{\text{in}}^* + \mathbf{p}_{\text{in}})/P\}$. Since this 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 subsequent steps. An overview of the entire network structure is given in Figure 7.

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

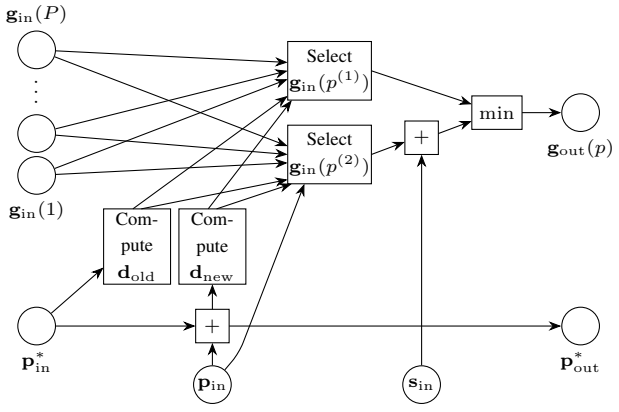

Figure 7: High-level idea how the FPTAS-NN computes $\mathbf{g}_{\text{out}}(p)$, $p \in [P]$, and $\mathbf{p}_{\text{out}}^*$ from the inputs.

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 7 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 full version of this paper (Hertrich and Skutella 2020) we provide a formal description of the FPTAS-NN as well as proofs of the following two theorems. The first one ensures that the FPTAS-NN produces only feasible Knapsack solutions, while the second one shows that the FPTAS-NN indeed provides a fully polynomial-time approximation scheme to solve the Knapsack Problem.

Theorem 2. *Suppose the FPTAS-NN is applied to a Knapsack instance with capacity $S = 1$, $s_i \in \,]0,1]$, and $p_i \in \mathbb{N}$, for $i \in [n]$. For every $i \in [n]$ and every $p \in [P]$, if $g(p,i) \leq 1$, then there exists a subset of $[i]$ with profit at least pd_i and size at most $g(p,i)$.*

Theorem 3. *For a Knapsack instance with capacity $S = 1$, $s_i \in \,]0,1]$, $p_i \in \mathbb{N}$, for $i \in [n]$, and for $\varepsilon \in \,]0,1]$, set $P := \lceil n^2/\varepsilon \rceil$. Let p^{OPT} be the profit of the optimal solution and $p^{\text{NN}} = \max\{pd_n \mid g(p,n) \leq 1\}$ be the best possible profit found by the FPTAS-NN. Then $p^{\text{NN}} \geq (1-\varepsilon)p^{\text{OPT}}$.*

Theorem 3 implies a tradeoff 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 $\mathcal{O}(P^2) = \mathcal{O}(n^4/\varepsilon^2)$ 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.

5 Neural Networks for Other CO Problems

In this section we demonstrate that our approach is by no means bound to the Knapsack Problem. In fact, for many other CO problems it is possible to convert a dynamic programming solution method into a provably correct NN. For certain NP-hard CO problems, a dynamic programming solution even implies the existence of a fully polynomial-time approximation scheme (Woeginger 2000). This, in turn, might shed light on the tradeoff between size of corresponding NNs and their solution quality, as for the Knapsack Problem in Section 4. In the following we provide several examples in order to support these claims.

Longest Common Subsequence. First, consider the problem of finding the length of the longest common subsequence of two finite integer sequences $x_1, x_2, \ldots, x_m$ and $y_1, y_2, \ldots, y_n$. A standard dynamic programming procedure, see, e.g., Cormen et al. (2001, Section 15.4), computes values $f(i,j)$ equal to the length of the longest common subsequence of the partial sequences $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j$ by applying the recursion

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

Since the sequence consists of integers, the check whether x_i equals y_j can be performed similarly to checking whether $p' = \mathbf{p}_{\text{in}}$ in Section 3. The remainder of the recursion only consists of maxima and sums. Hence, computing $f(i,j)$ from $f(i-1,j-1)$, $f(i-1,j)$, $f(i,j-1)$, x_i, and y_j can be realized via an NN of constant size. These basic units can be plugged together in a two-dimensional way for computing all values $f(i,j)$, $i \in [m]$, $j \in [n]$. The resulting NN can be seen as a two-dimensional RNN of constant size that is applied in an m by n grid structure, an architecture introduced by Graves, Fernández, and Schmidhuber (2007). Unfolding the RNN results in a feedforward NN of depth $\mathcal{O}(m+n)$ and size $\mathcal{O}(mn)$ for computing the length of the longest common subsequence.

Single-Source Shortest Path Problem. As a second example, we consider the Bellman-Ford algorithm for the Single-Source Shortest Path Problem, see, e.g., Kleinberg and Tardos (2006, Section 6.8). If $(c_{uv})_{u,v\in V}$ is the length matrix of a graph with vertex set V and $s \in V$ is the source vertex, this 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

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

Since this recursion consists only of sums and minima, it can be easily implemented in an NN. The sequential time complexity of the Bellman-Ford algorithm on complete digraphs with $n = |V|$ is $\mathcal{O}(n^3)$, which can naturally be parallelized into $\mathcal{O}(n)$ rounds. Since the best known NNs for computing the minimum of n numbers require $\mathcal{O}(\log n)$ depth (Arora et al. 2018), there exists an NN executing the Bellman-Ford algorithm with depth $\mathcal{O}(n \log n)$ and size $\mathcal{O}(n^3 \log n)$. Observe that in each round $i \in [n]$, the computation mapping the values $f(i-1,v)$, $v \in V$, to $f(i,v)$, $v \in V$, is the same. Therefore, this NN can also be seen as an RNN of depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n^2 \log n)$ that is applied n times.

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 (1991, Section 2.5.4). By repeated squaring, this can be achieved with only $\mathcal{O}(\log n)$ min-plus matrix multiplications. For a single multiplication it is required to compute $\mathcal{O}(n^2)$ times in parallel the minimum of n numbers. One of these minimum computations requires depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n \log n)$. Putting them in parallel to execute one min-plus matrix product results in depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n^3 \log n)$. Note that the whole execution consists of $\mathcal{O}(\log n)$ repetitions of the same procedure, namely squaring a matrix in the min-plus sense. Hence, this can again be viewed as an RNN with depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n^3 \log n)$ that is applied $\mathcal{O}(\log n)$ times. Unfolding results in a single feedforward NN with depth $\mathcal{O}(\log^2 n)$ and size $\mathcal{O}(n^3 \log^2 n)$ for solving the All-Pairs Shortest Path Problem.

Constrained Shortest Path Problem. Next, consider a common generalization of the Shortest Path Problem and the Knapsack Problem, namely the NP-hard Constrained Shortest Path Problem. Here, in addition to a (nonnegative) length matrix $(c_{uv})_{u,v\in V}$, 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 $\sum_{(u,v)\in P} r_{uv} \leq R$ 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 (2001). 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 3 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 3. 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 $\mathcal{O}(nc^* \log n)$. 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 tradeoff for the Constrained Shortest Path Problem similar to what we have shown in Section 4.

Traveling Salesperson Problem. Finally, consider the Bellman-Held-Karp algorithm for solving the (asymmetric) Traveling Salesperson Problem (TSP), see Bellman (1962); Held and Karp (1962). 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 $\min_{u\in V\setminus\{s\}}\{f(V\setminus\{s\},u)+c_{us}\}$. While the sequential time complexity of this algorithm on digraphs with $n = |V|$ is $\mathcal{O}(n^2 2^n)$, in an NN we can compute the values of f for all T with equal cardinality in parallel. As before, computing the minimum introduces an additional factor of $\log n$ in the depth and size of the network. Hence, in total, the TSP can be solved with an NN of depth $\mathcal{O}(n \log n)$ and size $\mathcal{O}(n^2 2^n \log n)$. In particular, a polynomially deep NN suffices to solve the NP-hard (asymmetric) TSP.

6 Conclusions and Future Work

An obvious open problem is to improve the obtained bounds on the required width of our neural network constructions. In particular, an interesting question is whether meaningful lower bounds beyond those immediately implied by the NP-hardness of the Knapsack Problem can be obtained, as suggested by our experimental results.

Notice that our networks only output the solution value but not the corresponding solution, i.e., subset of items. It is easy to see that, as for the dynamic program solving the Knapsack Problem, the subset of items can be obtained in a straightforward way via backtracking. On the other hand, notice that it is impossible for a ReLU NN (without threshold gates) to output (the characteristic vector of) the optimum subset of items: while the function computed by a ReLU NN is piecewise linear and continuous (Goodfellow et al. 2013; Arora et al. 2018), already infinitesimal changes of the input (i.e., the profit values of items) might change the optimum subset of items.

Another interesting direction for future research is to generalize our results of Section 5 by describing a general procedure to convert dynamic programs into ReLU NNs. Ideally, one could exactly classify the type of dynamic programs that guarantee the existence of a corresponding ReLU NN. Similar in spirit, Woeginger (2000) classifies dynamic programs that guarantee the existence of a fully polynomial-time approximation scheme.

References

Arora, R.; Basu, A.; Mianjy, P.; and Mukherjee, A. 2018. Understanding Deep Neural Networks with Rectified Linear Units. In *International Conference on Learning Representations*.

Bellman, R. 1962. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM* 9(1): 61–63.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural Combinatorial Optimization with Reinforcement Learning. *arXiv:1611.09940* .

Bengio, Y.; Lodi, A.; and Prouvost, A. 2018. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv:1811.06128* .

Bertsimas, D.; and Weismantel, R. 2005. *Optimization over Integers*. Dynamic Ideas.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to algorithms*. MIT press, second edition.

Eldan, R.; and Shamir, O. 2016. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, 907–940.

Emami, P.; and Ranka, S. 2018. Learning Permutations with Sinkhorn Policy Gradient. *arXiv:1805.07010* .

Fischetti, M.; and Lodi, A. 2010. On the knapsack closure of 0-1 Integer Linear Programs. *Electronic Notes in Discrete Mathematics* 36: 799–804.

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

Goodfellow, I.; Warde-Farley, D.; Mirza, M.; Courville, A.; and Bengio, Y. 2013. Maxout Networks. In *International Conference on Machine Learning*, 1319–1327.

Graves, A.; Fernández, S.; and Schmidhuber, J. 2007. Multi-dimensional recurrent neural networks. In *International Conference on Artificial Neural Networks*, 549–558.

Hanin, B. 2019. Universal function approximation by deep neural nets with bounded width and ReLU activations. *Mathematics* 7(10): 992.

Hanin, B.; and Rolnick, D. 2019. Complexity of Linear Regions in Deep Networks. In *International Conference on Machine Learning*, 2596–2604.

Hanin, B.; and Sellke, M. 2017. Approximating continuous functions by ReLU nets of minimal width. *arXiv:1710.11278* .

Held, M.; and Karp, R. M. 1962. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics* 10(1): 196–210.

Hertrich, C.; and Skutella, M. 2020. Provably Good Solutions to the Knapsack Problem via Neural Networks of Bounded Size. *arXiv:2005.14105* .

Hochbaum, D. S. 1997. Various Notions of Approximations: Good, Better, Best, and More. In Hochbaum, D. S., ed., *Approximation Algorithms for NP-hard Problems*, 346–446. PWS Publishing Co.

Hopfield, J. J.; and Tank, D. W. 1985. "Neural" computation of decisions in optimization problems. *Biological Cybernetics* 52(3): 141–152.

Karp, R. M. 1972. Reducibility among combinatorial problems. In Miller, R. E.; and Thatcher, J. W., eds., *Complexity of Computer Computations*, 85–103. Springer.

Kellerer, H.; Pferschy, U.; and Pisinger, D. 2004. *Knapsack Problems*. Springer.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning Combinatorial Optimization Algorithms over Graphs. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 30*, 6348–6358.

Kleinberg, J.; and Tardos, E. 2006. *Algorithm Design*. Pearson Education.

Kool, W.; van Hoof, H.; and Welling, M. 2019. Attention, Learn to Solve Routing Problems! In *International Conference on Learning Representations*.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *Nature* 521: 436–444.

Leighton, F. T. 1991. *Introduction to parallel algorithms and architectures: Arrays · trees · hypercubes*. Morgan-Kaufmann.

Liang, S.; and Srikant, R. 2017. Why deep neural networks for function approximation? In *International Conference on Learning Representations*.

Lodi, A.; and Zarpellon, G. 2017. On learning and branching: a survey. *TOP* 25(2): 207–236.

Lombardi, M.; and Milano, M. 2018. Boosting combinatorial problem modeling with machine learning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 5472–5478.

Martello, S.; and Toth, P. 1990. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons.

Montufar, G. F.; Pascanu, R.; Cho, K.; and Bengio, Y. 2014. On the Number of Linear Regions of Deep Neural Networks. In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 27*, 2924–2932.

Mukherjee, A.; and Basu, A. 2017. Lower bounds over Boolean inputs for deep neural networks with ReLU gates. *arXiv:1711.03073* .

Nguyen, Q.; Mukkamala, M. C.; and Hein, M. 2018. Neural Networks Should Be Wide Enough to Learn Disconnected Decision Regions. In *International Conference on Machine Learning*, 3737–3746.

Nowak, A.; Villar, S.; Bandeira, A. S.; and Bruna, J. 2017. Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450* .

Ohlsson, M.; Peterson, C.; and Söderberg, B. 1993. Neural Networks for Optimization Problems with Inequality Constraints: The Knapsack Problem. *Neural Computation* 5(2): 331–339.

Pascanu, R.; Montufar, G.; and Bengio, Y. 2014. On the number of inference regions of deep feed forward networks with piece-wise linear activations. In *International Conference on Learning Representations*.

Raghu, M.; Poole, B.; Kleinberg, J.; Ganguli, S.; and Dickstein, J. S. 2017. On the Expressive Power of Deep Neural Networks. In *International Conference on Machine Learning*, 2847–2854.

Safran, I.; and Shamir, O. 2017. Depth-width tradeoffs in approximating natural functions with neural networks. In *International Conference on Machine Learning*, 2979–2987.

Serra, T.; Tjandraatmadja, C.; and Ramalingam, S. 2018. Bounding and Counting Linear Regions of Deep Neural Networks. In *International Conference on Machine Learning*, 4565–4573.

Shalev-Shwartz, S.; and Ben-David, S. 2014. *Understanding machine learning: From theory to algorithms*. Cambridge University Press.

Smith, K. A. 1999. Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research. *INFORMS Journal on Computing* 11(1): 15–34.

Telgarsky, M. 2015. Representation Benefits of Deep Feedforward Networks. *arXiv:1509.08101* .

Telgarsky, M. 2016. Benefits of depth in neural networks. In *Conference on Learning Theory*, 1517–1539.

Vazirani, V. V. 2001. *Approximation Algorithms*. Springer.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer Networks. In Cortes, C.; Lawrence, N. D.; Lee, D. D.; Sugiyama, M.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 28*, 2692–2700.

Williamson, D. P.; and Shmoys, D. B. 2011. *The design of approximation algorithms*. Cambridge University Press.

Woeginger, G. J. 2000. When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)? *INFORMS Journal on Computing* 12(1): 57–74.

Xu, S.; Panwar, S. S.; Kodialam, M. S.; and Lakshman, T. V. 2020. Deep Neural Network Approximated Dynamic Programming for Combinatorial Optimization. In *AAAI Conference on Artificial Intelligence*, 1684–1691.

Yang, F.; Jin, T.; Liu, T.-Y.; Sun, X.; and Zhang, J. 2018. Boosting Dynamic Programming with Neural Networks for Solving NP-hard Problems. In *Asian Conference on Machine Learning*, 726–739.

Yarotsky, D. 2017. Error bounds for approximations with deep ReLU networks. *Neural Networks* 94: 103–114.

Ziegelmann, M. 2001. *Constrained Shortest Paths and Related Problems*. Ph.D. thesis, Universität des Saarlandes Saarbrücken.