

ReLU Neural Networks of Polynomial Size for Exact Maximum Flow Computation*

Christoph Hertrich [†]
London School of Economics
and Political Science
c.hertrich@lse.ac.uk

Leon Sering [‡]
ETH Zürich
sering@math.ethz.ch

Abstract

This paper studies the expressive power of artificial neural networks with rectified linear units. In order to study them as a model of *real-valued* computation, we introduce the concept of *Max-Affine Arithmetic Programs* and show equivalence between them and neural networks concerning natural complexity measures. We then use this result to show that two fundamental combinatorial optimization problems can be solved with polynomial-size neural networks. First, we show that for any undirected graph with n nodes, there is a neural network (with fixed weights and biases) of size $\mathcal{O}(n^3)$ that takes the edge weights as input and computes the value of a minimum spanning tree of the graph. Second, we show that for any directed graph with n nodes and m arcs, there is a neural network of size $\mathcal{O}(m^2n^2)$ that takes the arc capacities as input and computes a maximum flow. Our results imply that these two problems can be solved with strongly polynomial time algorithms that solely uses affine transformations and maxima computations, but no comparison-based branchings.

1 Introduction

Artificial neural networks (NNs) achieved breakthrough results in various application domains like computer vision, natural language processing, autonomous driving, and many more [42]. Also in the field of combinatorial optimization (CO), promising approaches to utilize NNs for problem solving or improving classical solution methods have been introduced [9]. However, the theoretical understanding of NNs still lags far behind these empirical successes.

All neural networks considered in this paper are *feedforward neural networks with rectified linear unit (ReLU) activations*, one of the most popular models in practice [22]. These NNs are directed, acyclic, computational graphs in which each edge is equipped with a fixed weight and each node with a fixed bias. Each node (*neuron*) computes an affine transformation of the outputs of its predecessors and applies the ReLU activation function $x \mapsto \max\{0, x\}$ on top. The full NN then computes a function mapping real-valued inputs to real-valued outputs. A simple example is given in Figure 1.

*A large portion of this work was completed while both authors were affiliated with TU Berlin. We thank Max Klimm, Jennifer Manke, Arturo Merino, Martin Skutella, and László Végh for many inspiring and fruitful discussions and valuable comments.

[†]Funded by DFG-GRK 2434 “Facets of Complexity” and by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement ScaleOpt-757481).

[‡]Funded by DFG Excellence Cluster MATH+ (EXC-2046/1, project ID: 390685689).

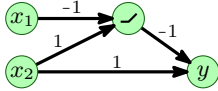


Figure 1: A small NN with two input neurons \mathbf{x}_1 and \mathbf{x}_2 , a single ReLU neuron labelled with the shape of the ReLU function, and one output neuron \mathbf{y} . It computes the function

$$\begin{aligned} \mathbf{x} &\mapsto \mathbf{y} \\ &= \mathbf{x}_2 - \max\{0, \mathbf{x}_2 - \mathbf{x}_1\} \\ &= -\max\{-\mathbf{x}_2, -\mathbf{x}_1\} \\ &= \min\{\mathbf{x}_1, \mathbf{x}_2\}. \end{aligned}$$

The neurons are commonly organized in *layers*. The *depth*, *width*, and *size* of an NN are defined as the number of layers, the maximum number of neurons per layer, and the total number of neurons, respectively. An important theoretical question about these NNs is concerned with their expressivity: which functions can be represented by an NN of a certain depth, width, or size?

Neural network expressivity has been thoroughly investigated from an approximation point of view. For example, so-called *universal approximation theorems* [4, 13, 33] show that every continuous function on a bounded domain can be arbitrarily well approximated with only a single nonlinear layer. However, for a full theoretical understanding of this fundamental machine learning model it is necessary to understand what functions can be *exactly* expressed with different NN architectures. For instance, insights about exact representability have boosted our understanding of the computational complexity of the task to train an NN with respect to both, algorithms [5, 38] and hardness results [11, 20, 23]. It is known that a function can be expressed with a ReLU NN if and only if it is *continuous and piecewise linear* (CPWL) [5]. However, many surprisingly basic questions remain open. For example, it is not known whether two layers of ReLU units (with any width) are sufficient to compute the function $f: \mathbb{R}^4 \rightarrow \mathbb{R}, x \mapsto \max\{0, x_1, x_2, x_3, x_4\}$ [3, 30].

In this paper we explore another fundamental question within the research stream of exact representability: what are families of CPWL functions that can be represented with ReLU NNs of polynomial size? In other words, using NNs as a model of computation operating on *real* numbers (in contrast to Turing machines or Boolean circuits, which operate on binary encodings), which problems do have polynomial complexity in this model?

Our motivation to study this model stems from a variety of different perspectives, including strongly polynomial time algorithms, arithmetic circuit complexity, parallel computation, and learning theory. We believe that classical combinatorial optimization problems are a natural example to study this model of computation because their algorithmic properties are well understood in each of these areas.

Clearly, 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 be false. 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, every function computed by such NNs is continuous, making it impossible to realize instructions like a simple **if**-

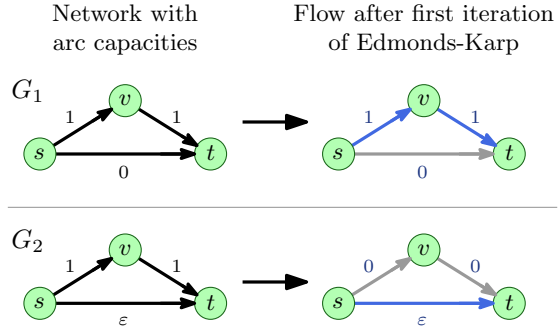


Figure 2: This example shows that the outcome of one iteration of the Edmonds-Karp algorithm for computing a maximum flow depends discontinuously on the arc capacities. Here, a small adjustment of the capacity of arc st leads to a drastic change of the flow after the first iteration.

branching based on a comparison of real numbers. In fact, there are related models of computation for which the use of branchings is exponentially powerful [34].

For some CO problems, classical algorithms do not involve comparison-based branchings and, thus, can easily be implemented as an NN. This is, for example, true for many dynamic programs. In these cases, the existence of efficient NNs follows immediately. We refer to Hertrich and Skutella [31] for some examples of this kind. In particular, polynomial-size NNs to compute the length of a shortest path in a network from given arc lengths are possible.

For other problems, like the Minimum Spanning Tree Problem or the Maximum Flow Problem, all classical algorithms use comparison-based branchings. For example, many maximum flow algorithms use them to decide whether an arc is part of the *residual network*. More specifically, in the Edmonds-Karp algorithm a slight perturbation (from 0 to ε) in the capacities can lead to different augmenting path and therefore to a completely different intermediate flow; see Figure 2. Such a discontinuous behavior can never be represented by a ReLU NN. Without the results of this paper, it would have remained unclear whether there exist polynomial-size NNs to solve Minimum Spanning Tree and Maximum Flow problems at all.

1.1 Our Main Results

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). 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 constructing NNs with bounded size and could be useful for further research about NN expressivity beyond the scope of this paper.

We use this result to prove our two main theorems. The first one shows that computing the value of a minimum spanning tree has polynomial complexity on NNs. The proof is based on a result from subtraction-free circuit complexity [19].

Theorem 1. *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.*

The second result shows that computing a maximum flow has polynomial complexity on NNs. Since all classical algorithms involve conditional branchings based on the comparison of real numbers, the proof involves the development of a new strongly polynomial maximum flow algorithm which avoids such branchings. While, in terms of standard running times, the algorithm is definitely not competitive with algorithms that exploit comparison-based branchings, it is of independent interest with respect to the structural understanding of flow problems.

Theorem 2. *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 $\mathcal{O}(m^2 n^2)$ and width $\mathcal{O}(1)$ that correctly maps a vector of arc capacities to a vector of flow values in a maximum s - t -flow.*

Let us point out that in case of minimum spanning trees, the NN computes only the objective value, while for maximum flows, the NN computes the actual solution. There is a structural reason for this difference: Due to their continuous nature, ReLU NNs cannot compute a discrete solution vector, like an indicator vector of the optimal spanning tree, because infinitesimal changes of the edge weights would lead to jumps in the output. For the Maximum Flow Problem, however, the optimal flow itself does indeed have a continuous dependence on the arc capacities.

1.2 Discussion of the Results

Before presenting our results in more detail, we discuss the significance and limitations of our results from various perspectives.

Learning Theory A standard approach to create a machine learning model usually contains the following two steps. The first step is to fix a particular *hypothesis class*. When using NNs, this means to fix an architecture, that is, the underlying graph of the NN. Then, each possible choice of weights and biases of all affine transformations in the network constitutes one hypothesis in the class. The second step is to run an optimization routine to find a hypothesis in the class that fits given training data as accurately as possible.

A core theme in learning theory is to analyse how the choice of the hypothesis class influences different kind of errors made by the machine learning model. If the chosen hypothesis class is too small, then even the best hypothesis might not be good enough and the model incurs a large *approximation error*. If the chosen hypothesis class is too large, then the model is likely to overfit on the training data resulting in a large *generalization error* when applied to unseen data. Finding a good tradeoff between these different errors is the art of every machine learning practitioner.

Classical learning theory provides a rich toolbox for understanding the effect of a specific hypothesis class on the learning error using notions like PAC learnability, VC-dimension and Rademacher complexity [58]. However, these theories struggle to explain the success of modern NN architectures, which are massively overparameterized. Yet they usually do not suffer from overfitting and the generalization error is much lower than expected [67].

While there exist many attempts to mathematically explain the mysterious success of modern NNs [10], there is still a long way ahead of us. Understanding what CPWL functions are actually contained in the hypothesis classes defined by NNs of a certain size (in particular, polynomial size) is a key insight in this direction. We see our combinatorial, exact perspective as a counterbalance and complement to the usual approximate point of view.

Strongly Polynomial Time Algorithms As pointed out above, polynomial-size NNs correspond to a subclass of strongly polynomial time algorithms with a very limited set of operations allowed. Given that this subclass stems from one of the most basic machine learning models, our grand vision, to which we contribute with our results, is to understand for different CO problems whether they admit strongly polynomial time algorithms of this type.

Algorithms of this type have not been known before for the two problems considered in this paper. It remains an open question whether such algorithms, and hence, polynomial-size NNs, exist to solve other CO problems for which strongly polynomial time algorithms are known. Can they, for instance, compute the weight of a minimum weight perfect matching in (bipartite) graphs? Can they compute the cost of a minimum cost flow from either node demands or arc costs, while the other of the two quantities is considered to be fixed?

A major open question is also to prove lower bounds on NN sizes. Can we find a family of CPWL functions (corresponding to a CO problem or not) which can be evaluated in strongly polynomial time, but *not* computed by polynomial-size NNs? While proving lower bounds in complexity theory always seems to be a challenging task, we believe that not all hope is lost. For example, in the area of *extended formulations*, it has been shown that there exist problems (in particular, minimum weight perfect matching) which can be solved in strongly polynomial time, but every linear programming formulation to this problem must have exponential size [55]. Possibly, one can show in the same spirit that also polynomial-size NN representations are not achievable.

Parametric Algorithms Our results also have an interesting interpretation from the perspective of parametric algorithms. There exists a variety of literature concerning the question how solutions to the Maximum Flow Problem can be represented if the input depends on one or several unknown parameters; see, e.g., the works by Gallo et al. [21] and McCormick [45]. Our results imply that there exists such a representation of polynomial size for the most general form of parametric maximum flow problems, namely the one where *all* arc capacities are independent free parameters. This representation is given in the form of a polynomial-size NN and can be evaluated in polynomial time.

Boolean Circuits 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. Interestingly, this makes understanding the computational power of NNs much easier. It is easy to see that ReLU NNs can directly simulate AND-, OR-, and NOT-gates, and thus every Boolean circuit [47]. Hence, in Boolean arithmetics, every problem in P can be solved with polynomial-size NNs.

However, requiring the networks to solve a problem for all possible real-valued inputs seems to be much stronger. Consequently, the class of functions representable with polynomial-size NNs is much less understood than in Boolean arithmetics. Our results suggest that rethinking and forbidding basic algorithmic paradigms (like comparison-based branchings) can help towards improving this understanding.

Arithmetic Circuits As a circuit model with real-valued computation, ReLU networks are naturally closely related to *arithmetic circuits*. Just like NNs, arithmetic circuits are computational graphs in which each node computes some arithmetic expression (traditionally addition or multiplication) from the outputs of all its predecessors. Arithmetic circuits are well-studied objects in complexity theory [60]. Closer to ReLU NNs, there is a special kind of arithmetic circuits called *tropical circuits* [35]. 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 the max-plus algebra.

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. In fact, 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 [36], no polynomial-size tropical circuit can do this. However, Theorem 1 shows that NNs of cubic size (in the number of nodes of the input graph) are 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 [36]. 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 reasons, lower bounds from arithmetic circuit complexity do not transfer to NNs. Hence, we identify it as a major challenge to prove meaningful lower bounds of any kind for the computational model of NNs.

Parallel Computation Similar to Boolean circuits, NNs are naturally a model of parallel computation by performing all operations within one layer at the same time. 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 [5, 17, 43, 56, 62, 63, 66].

Interestingly, a related observation can be made for the Maximum Flow Problem using complexity theory. By the result of Arora et al. [5] mentioned above, any CPWL function can be represented with logarithmic depth in the input dimension, while not giving any guarantee on the total size. In particular, this is also true for the function mapping arc capacities to a maximum flow. Hence, NNs with logarithmic depth can solve the Maximum Flow Problem and it arises the question whether such shallow NNs are also possible while maintaining polynomial total size.

The answer is most likely “no” since the Maximum Flow Problem is *P-complete* [25]. 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 [26]. 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, Theorem 2 can probably not be improved to neural networks with polylogarithmic depth while maintaining overall polynomial size.

1.3 Further Related Work

Using NNs to solve CO problems started with so-called *Hopfield networks* [32] and related architectures in the 1980s and has been extended to general nonlinear programming problems later on [37]. Smith [61] surveys these early approaches. Also, specific NNs to solve the Maximum Flow Problem have been developed before [2, 16, 48]. However, the NNs used in these works are conceptually very different from modern feedforward NNs that are considered in this paper.

In recent years interactions between NNs and CO have regained a lot of attention in the literature [9]. For example, NNs have successfully been used to decide on which variables to branch when solving mixed-integer linear programs [44]. Also, a large variety of NN approaches for specific CO problems has been proposed [8, 18, 39, 40, 50, 64]. However, these approaches have in common that they are of heuristic nature and no guarantees on the solution quality or running time can be given.

Concerning general expressivity of NNs, so-called *universal approximation theorems* [4, 13, 33] state that a single layer between inputs and outputs is already sufficient to approximate any continuous function on a compact domain arbitrarily well. These results do not have implications on exact representability. Various trade-offs between depth and width of NNs [5, 17, 27, 29, 43, 49, 54, 56, 62, 63, 66] and approaches to count and bound the number of linear regions of a ReLU NN [28, 46, 53, 54, 57] have been found.

NNs have been studied from a circuit complexity point of view before [7, 52, 59]. 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.

For an introduction to classical maximum flow algorithms, we refer to textbooks [1, 41, 65]. The former two of these books also cover minimum spanning tree algorithms. Standard maximum

flow algorithms include the ones by Edmonds and Karp [15], Dinic [14], as well as the Push-Relabel algorithm by Goldberg and Tarjan [24]. The asymptotically fastest known combinatorial algorithm due to Orlin [51] runs in $\mathcal{O}(nm)$ time for networks with n nodes and m arcs. Recently, almost linear, weakly polynomial algorithms based on interior point methods have been developed [12]. However, polynomial-size NNs necessarily correspond to strongly polynomial algorithms, and even among those, classical algorithms are usually not applicable for direct implementation on an NN because they require conditional branchings as outlined above.

2 Algorithms and Proof Overview

Before diving into all technical details, we provide the reader with an overview of our proof techniques.

2.1 Max-Affine Arithmetic Programs

For the purpose of algorithmic investigations of ReLU NNs, we introduce the pseudo-code language *Max-Affine Arithmetic Programs* (MAAPs). A MAAP operates on real-valued variables. The only operations allowed in a MAAP are computing maxima and affine transformations of variables as well as parallel and sequential **for** loops with a *fixed*¹ number of iterations. In particular, no **if** branchings are allowed. With a MAAP A , we associate three complexity measures $d(A)$, $w(A)$, and $s(A)$, which can easily be calculated from a MAAP’s description. The intuition behind these measures is that they correspond (up to constant factors) to the depth, width, and size of an NN computing the same function as the MAAP does. We formalize this intuition by proving the following proposition.

Proposition 3. *For a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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 $s(A) = 4s$.*

The proof of the proposition works by providing explicit constructions to convert a MAAP into an NN (part (i)), and vice versa (part (ii)) while taking care that the different complexity measures translate respectively.

The takeaway from this exercise is that for proving that NNs of a certain size can compute certain functions, it is sufficient to develop an algorithm in the form of a MAAP that computes the same function and to bound its complexity measures $d(A)$, $w(A)$, and $s(A)$.

2.2 Minimum Spanning Trees

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 Minimum Spanning Tree Problem is to find a spanning tree with the least possible total edge weight.

Classical algorithms for the Minimum Spanning Tree Problem, for example Kruskal’s or Prim’s algorithm, compare the edge weights and use comparison-based branchings to determine the order

¹In this context, *fixed* means that the number of iterations cannot depend on the specific instance. It can still depend on the size of the instance (e.g., the size of the graph in case of the two CO problems considered in this paper).

Algorithm 1: MST_n : Compute the value of a minimum spanning tree for the complete graph on $n \geq 3$ vertices.

Input: Edge weights $(x_{ij})_{1 \leq i < j \leq n}$.

- 1 $y_n \leftarrow \min_{i \in [n-1]} x_{in}$
- 2 **for each** $1 \leq i < j \leq n - 1$ **do parallel**
- 3 | $x'_{ij} \leftarrow \min \{ x_{ij}, x_{in} + x_{jn} - y_n \}$
- 4 **return** $y_n + \text{MST}_{n-1} \left((x'_{ij})_{1 \leq i < j \leq n-1} \right)$

in which edges are added to the solution. Thus, they cannot be written as a MAAP or implemented as an NN. Instead, Theorem 1 can be shown by “tropicalizing” a result by Fomin et al. [19] from arithmetic circuit complexity.

To be more precise, Fomin et al. [19] 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) . If \mathcal{T} denotes the set of all spanning trees, then this polynomial is $\sum_{T \in \mathcal{T}} \prod_{e \in T} x_e$ 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 a minimum spanning tree: $\min_{T \in \mathcal{T}} \sum_{e \in T} x_e$.

In the same way, one can tropicalize the arithmetic circuit provided by Fomin et al. [19]. In fact, every sum gate is just replaced with the small NN from Figure 1 computing the minimum of its inputs, 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 a minimum spanning tree 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 a minimum spanning tree, to unveil the algorithmic ideas behind this construction, we provide an equivalent, completely combinatorial proof of Theorem 1, making use of MAAPs and Proposition 3.

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 a minimum spanning tree. 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 1.

Let us mention that the use of recursions is just a technicality because for each fixed n , the recursion can be unrolled and the MAAP can be stated explicitly. In each step, 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 smaller graph. This idea of removing the vertices one by one can be seen as the translation of the so-called *star-mesh transformation* used by Fomin et al. [19] into the combinatorial world.

We prove Theorem 1 by, firstly, showing that Algorithm 1 indeed computes the correct objective value, and secondly, bounding its complexity measures $d(A)$, $w(A)$, and $s(A)$ and applying Proposition 3.

Algorithm 2: Compute a maximum flow for a fixed graph $G = (V, E)$.

Input: Capacities $(\nu_e)_{e \in E}$.

// Initializing:

- 1 **for each** $uv \in \vec{E}$ **do parallel**
- 2 $x_{uv} \leftarrow 0$ // flow; negative value correspond to flow on vu
- 3 $c_{uv} \leftarrow \nu_{uv}$ // residual forward capacities
- 4 $c_{vu} \leftarrow \nu_{vu}$ // residual backward capacities

// Main part:

- 5 **for** $k = 1, \dots, n - 1$ **do**
- 6 **for** $i = 1, \dots, m$ **do**
- 7 $(y_e)_{e \in \vec{E}} \leftarrow \text{FindAugmentingFlow}_k((c_e)_{e \in E})$
 /* Returns an augmenting flow (respecting the residual capacities)
 that only uses paths of length exactly k and saturates at least
 one arc. */
- 8 **for each** $uv \in \vec{E}$ **do parallel**
- 9 $x_{uv} \leftarrow x_{uv} + y_{uv}$
- 10 $c_{uv} \leftarrow c_{uv} - y_{uv}$
- 11 $c_{vu} \leftarrow c_{vu} + y_{uv}$
- 12 **return** $(x_e)_{e \in \vec{E}}$

2.3 Maximum Flows

For a given directed graph with a source node s , a sink node t , and nonnegative capacities on each arc, the Maximum Flow Problem asks to find a flow value for each arc such that no capacity is exceeded, the inflow equals the outflow at each node except for s and t , and the outflow at s (or equivalently the inflow at t) is maximised.

Since classical maximum flow algorithms rely on conditional branchings based on the comparison of real numbers (for instance, to check which arcs are contained in the residual network), we develop a new maximum flow algorithm in the form of a MAAP (see Algorithms 2 and 3), which then translates to an NN of the claimed size by Proposition 3.

To explain our algorithm, let us start by recalling the key ideas of the classical Edmonds-Karp-Dinic algorithm [14, 15]. The algorithm repeatedly finds a shortest s - t path in the residual graph $G^* = (V, E^*)$, and sends the maximum possible amount of flow on such a path, that is, saturates at least one arc. The algorithm terminates by returning a minimum cut once t cannot be reached from s in the residual graph. The key insight in the analysis is that the distance from s to t in the residual graph is non-decreasing, and strictly increases within at most m such iterations. Thus, the number of iterations can be bounded by $\mathcal{O}(nm)$.

A shortest path can be characterized by *distance labels*. The vector $d \in \mathbb{R}_+^V$ is a distance labelling if $d(s) = 0$ and $d(v) \leq d(u) + 1$ for every residual arc $uv \in E^*$. If there exists an s - t path P such that $d(v) = d(u) + 1$ for every arc in P , then P is a shortest path. Identifying a shortest path is equivalent to finding distance labels and such a path. We note that the preflow-push algorithm [24] explicitly relies on using distance labels and pushing flow on residual arcs uv with $d(v) = d(u) + 1$. However, finding such a labelling requires **if**-branchings as it needs to identify the arcs in E^* , that is, arcs with positive residual capacity.

Algorithm 3: FindAugmentingFlow_k for a fixed graph $G = (V, E)$ and a fixed length k .

Input: Residual capacities $(c_e)_{e \in E}$.

```

// Initializing:
1 for each  $vw \in \vec{E}$  do parallel
2   |  $z_{vw} \leftarrow 0$  // flow in residual network
3   |  $z_{wv} \leftarrow 0$ 
4 for each  $(i, v) \in [k] \times (V \setminus \{t\})$  do parallel
5   |  $Y_v^i \leftarrow 0$  // excessive flow at  $v$  in iteration  $i$  (from  $k$  to 1)
6   |  $a_{i,v} \leftarrow 0$  // initialize fattest path values

// Determining the fattest path values:
7 for each  $v \in N_t^-$  do parallel
8   |  $a_{1,v} \leftarrow c_{vt}$ 
9 for  $i = 2, 3, \dots, k$  do
10  | for each  $v \in V \setminus \{t\}$  do parallel
11  |   |  $a_{i,v} \leftarrow \max_{w \in N_v^+ \setminus \{t\}} \min \{ a_{i-1,w}, c_{vw} \}$ 

// Pushing flow of value  $a_{k,s}$  from  $s$  to  $t$ :
12  $Y_s^k \leftarrow a_{k,s}$  // 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
16  |     | // Push flow out of  $v$  and into  $w$ :
16  |     |  $f \leftarrow \min \{ Y_v^i, c_{vw}, a_{i-1,w} - Y_w^{i-1} \}$  // value we can push over  $vw$  such
16  |     |   that this flow can still arrive at  $t$ 
17  |     |  $z_{vw} \leftarrow z_{vw} + f$ 
18  |     |  $Y_v^i \leftarrow Y_v^i - f$ 
19  |     |  $Y_w^{i-1} \leftarrow Y_w^{i-1} + f$ 
20 for each  $v \in N_t^-$  do parallel
21  | // Push flow out of  $v$  and into  $t$ :
21  |  $z_{vt} \leftarrow Y_v^1$ 
22  |  $Y_v^1 \leftarrow 0$ 

// 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  |     |  $b \leftarrow \min \{ Y_w^i, z_{vw} \}$  // value we can push backwards along  $vw$ 
27  |     |  $z_{vw} \leftarrow z_{vw} - b$ 
28  |     |  $Y_w^i \leftarrow Y_w^i - b$ 
29  |     |  $Y_v^{i+1} \leftarrow Y_v^{i+1} + b$ 

30 for each  $uv \in \vec{E}$  do parallel
31  |  $y_{uv} \leftarrow z_{uv} - z_{vu}$ 
32 return  $(y_e)_{e \in \vec{E}}$ 

```

At a high level, our algorithm is similar, but it avoids knowing the arcs in the residual graph and the length k of the shortest residual s - t path explicitly. Instead, we guess k in each iteration of the main procedure (Algorithm 2), making sure that we never overestimate the true length. The guess is initialized as $k = 1$ and, in accordance with the Edmonds-Karp-Dinic analysis, we increment k by one in every m iterations. Based on our guess for k , we use a subroutine `FindAugmentingFlow $_k$` (Algorithm 3) with the following feature: if the actual shortest path length is exactly k , the subroutine will send flow from s to t on (possibly multiple) paths of length exactly k , saturating at least one arc. If the shortest path is longer than k , nothing happens in the current iteration.

Instead of distance labels, the subroutine computes *fattest path values* $\mathbf{a}_{i,v}$ (line 7 to 11) that represent the maximum amount of flow that can be sent from v to t on a path of length exactly i . Such values can be obtained by a simple dynamic program that is easy to implement as a MAAP. Thus, a path $(s = v_k, v_{k-1}, \dots, v_1, v_0 = t)$ of length exactly k is contained in the residual network if and only if $\mathbf{a}_{i,v_i} > 0$ for all $i = 1, \dots, k$. Our algorithm makes sure that we only send flow along arcs that are contained in such paths. In particular, the current iteration will send positive flow if and only if $\mathbf{a}_{k,s} > 0$. However, we cannot recover the shortest s - t path with capacity $\mathbf{a}_{k,s}$. Therefore, in general, flow will not be sent along a single path and the value of the flow output by `FindAugmentingFlow $_k$` might be strictly less than $\mathbf{a}_{k,s}$.

After computing the $\mathbf{a}_{i,v}$ values, `FindAugmentingFlow $_k$` pushes flow greedily from s towards the sink, using a lexicographic selection rule to pick the next arc to push flow on (line 12 to 22). On the high level, this is similar to the preflow-push algorithm, but using the $\mathbf{a}_{i,v}$ values that encode the shortest path distance information implicitly. This may leave some nodes with excess flow; a final cleanup phase (line 23 to 29) is then needed to send the remaining flow back to the source s .

An example for the `FindAugmentingFlow $_k$` -subroutine is given in Figure 3. We emphasize again that, although the description of the subroutine in the example in Figure 3 seems to rely heavily on the distance of a node to t , this information is calculated and used only in an implicit way via the precomputed $\mathbf{a}_{i,v}$ values. This way, we are able to implement the subroutine without the usage of comparison-based branchings.

The proof of correctness for our algorithm consists of two main steps. The first step is the analysis of the `FindAugmentingFlow $_k$` subroutine. This involves carefully showing that the returned flow indeed satisfies flow conservation, is feasible with respect to the residual capacities, uses only arcs that lie on a s - t -path of length exactly k in the residual network, and most importantly, if such a path exists, it saturates at least one arc. This last property can be shown using the lexicographic selection rule to pick the next arc to push flow on. Note that, in general, the subroutine neither returns a single path (as in the Edmonds-Karp algorithm [15]), nor a blocking flow (as in the Dinic algorithm [14]). The second main step is to show that, nevertheless, the properties of the subroutine are sufficient to ensure that the distance from s to t in the residual network increases at least every m iterations, such that we terminate with a maximum flow after nm iterations.

With the correctness of the whole MAAP at hand, Theorem 2 follows by simply counting the complexity measures $d(A)$, $w(A)$, and $s(A)$, and applying Proposition 3.

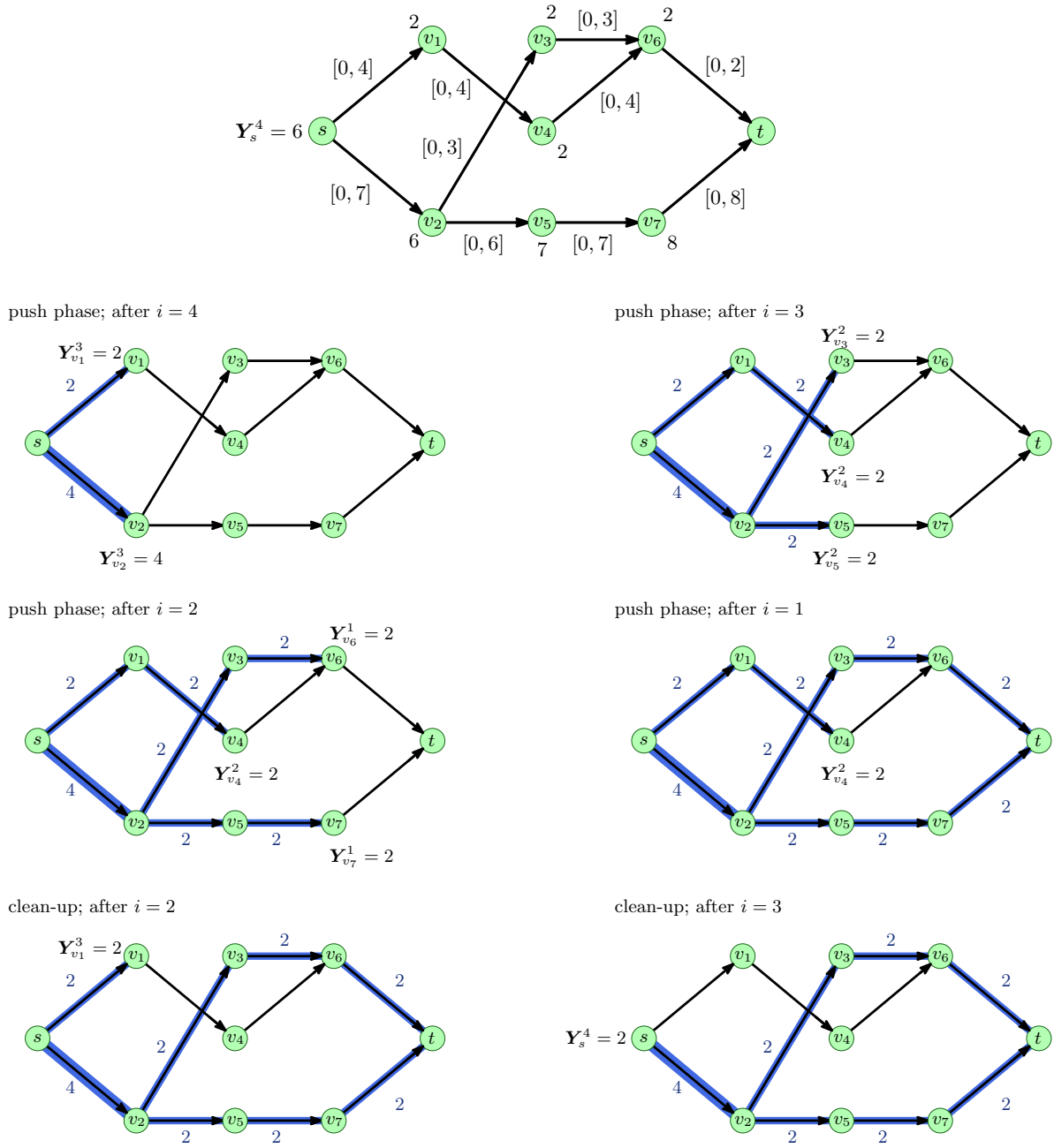


Figure 3: Example of the $\text{FindAugmentingFlow}_k$ subroutine for $k = 4$. The edge labels in the top figure are the residual capacity bounds in the current iteration. The first step is to compute the fastest path values $\mathbf{a}_{i,v}$, which are depicted as node labels in the top figure. The values \mathbf{Y}_v^i always denote the excessive flow of a vertex v with distance i from the sink. All values that are not displayed are zero. At s , we initialize $\mathbf{Y}_s^4 = \mathbf{a}_{4,s} = 6$. Then, excessive flow is pushed greedily towards the sink, as shown in the four figures in the middle. While doing so, we ensure that at each vertex the arriving flow does not exceed its value $\mathbf{a}_{i,v}$. For this reason, flow can get stuck, as it happens at v_4 in this example. Therefore, in a final cleanup phase, depicted in the two bottom figures, we push flow back to the source s . 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 .

3 Neural Networks with Rectified Linear Units

In this section, we formally define the primary object of study in this paper, using notations similar to [58, Chapter 20]. A *feedforward neural network with rectified linear units*, abbreviated by ReLU NN or simply NN, is a directed acyclic graph (V, E) , for which each arc $e \in E$ is equipped with a *weight* $w_e \in \mathbb{R}$ and each node $v \in V \setminus V_0$ is equipped with a *bias* $b_v \in \mathbb{R}$. Here, V_0 denotes the set of all nodes with no incoming arcs. The nodes V of an NN are called *neurons* and the *depth* k is given by the length of a longest path. The neurons are partitioned into *layers* $V = V_0 \cup V_1 \cup \dots \cup V_k$ in such a way that the layer index strictly increases along each arc.² In addition, we assume that V_k is exactly the set of neurons with out-degree zero. The neurons in V_0 and V_k are called *input neurons* and *output neurons*, respectively. All other neurons in $V \setminus (V_0 \cup V_k)$ are so-called *hidden neurons*. Let $n_\ell := |V_\ell|$ be the number of neurons in layer ℓ . The *width* and *size* of the NN are given by $\max\{n_1, \dots, n_{k-1}\}$ and $\sum_{\ell=1}^{k-1} n_\ell$, respectively.

In our paper it is crucial to distinguish fixed parameters of NNs, like architectural details and weights, from input, activation, and output values of neurons. We denote the latter by bold symbols in order to make the difference visible.

The *forward pass* of an NN is given by a function $\mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_k}$ that is obtained as follows. For an input vector $\mathbf{x} \in \mathbb{R}^{n_0}$ we inductively compute an *activation* $\mathbf{a}(v)$ for every $v \in V \setminus V_0$ and an *output* $\mathbf{o}(v)$ for every $v \in V \setminus V_k$. First, the output values $\mathbf{o}(v)$ of the input neurons are set to the corresponding entries of the input vector \mathbf{x} . Second, the activation of a neuron $v \in V \setminus V_0$ is given by the weighted sum of the outputs of all of its predecessors plus the bias b_v . Formally, we set $\mathbf{a}(v) := b_v + \sum_{u: uv \in E} w_{uv} \mathbf{o}(u)$. Third, we apply the so-called *activation function* σ to obtain the output of all hidden neurons, i.e., $\mathbf{o}(v) := \sigma(\mathbf{a}(v))$. In this work we only consider the *ReLU function* $\sigma(z) = \max\{0, z\}$ as activation function. Finally, the activation values $\mathbf{a}(v)$ of the output neurons $v \in V_k$ provide the *output vector* $\mathbf{y} \in \mathbb{R}^{n_k}$. Note that the activation function is not applied to these last activation values.

We point the reader once again to the example in Figure 1. Note that by our definitions the NN in the figure has depth 2, width 1, and size 1.

4 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 handling of such NNs, 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*. In addition, there are natural- or real-valued *constants* and different kinds of *instructions*. 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. A similar concept

²In other literature arcs are only allowed between successive layers. Clearly, this can always be achieved by introducing additional neurons. For our purposes, however, we want to avoid this restriction.

is known from circuit complexity, where Boolean circuit families are used to measure complexity; compare [6].

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 $\max \left\{ b^{(i)} + \sum_j c_j^{(i)} \mathbf{v}_j^{(i)} \mid i = 1, \dots, n \right\}$, 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.³
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.⁴

Algorithm 4 shows an example MAAP to illustrate the possible instructions.

Algorithm 4: Instructions.

```

Input: Input variables  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ .

// 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  $k = 1, \dots, n - 1$  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  $k = 4, \dots, n$  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  $(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n)$ 

```

³Local variables in different blocks may have the same name if their scope is limited to their block.

⁴Again, local variables within different iterations may have the same name if their scope is limited to their iteration.

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) [26]. 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 define $d(A) := \lceil \log_2 k \rceil$, $w(A) := 2k$, and $s(A) := 4k$.
- For a sequence A of instruction blocks B_1, B_2, \dots, B_k we define $d(A) := \sum_{i=1}^k d(B_i)$, $w(A) := \max_{i=1}^k w(B_i)$, and $s(A) := \sum_{i=1}^k s(B_i)$.
- For a **Do-Parallel** instruction A consisting of parallel blocks B_1, B_2, \dots, B_k we define $d(A) := \max_{i=1}^k d(B_i)$, $w(A) := \sum_{i=1}^k w(B_i)$, and $s(A) := \sum_{i=1}^k s(B_i)$.
- For a **For-Do** loop A with k iterations that executes block B_i in iteration i we define $d(A) := \sum_{i=1}^k d(B_i)$, $w(A) := \max_{i=1}^k w(B_i)$, and $s(A) := \sum_{i=1}^k s(B_i)$.
- For a **For-Do-Parallel** loop with k iterations that executes block B_i in iteration i we define $d(A) := \max_{i=1}^k d(B_i)$, $w(A) := \sum_{i=1}^k w(B_i)$, and $s(A) := \sum_{i=1}^k s(B_i)$.

With these definitions at hand, we can prove the desired correspondence between the complexities of MAAPs and NNs.

Proposition 3. *For a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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 $s(A) = 4s$.*

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 **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 the construction of [5, Lemma D.3], an NN with depth $\lceil \log_2 k \rceil + 1$, width $2k$, and size $4k$ can compute f , which settles the induction base.

Algorithm 5: 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 $\mathbf{a}(v) \leftarrow b_v + \sum_{u: uv \in E} w_{uv} \mathbf{o}(u)$

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 $\mathbf{a}(v) \leftarrow b_v + \sum_{u: uv \in E} w_{uv} \mathbf{o}(u)$

7 **return** $(\mathbf{a}(v))_{v \in V_{d+1}}$.

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, 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 $s(B_1) + s(B_2) = s(A)$; cf. [5, Lemma D.1]. 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 described in Section 3. It is easy to verify that Algorithm 5 computes f with the claimed complexity measures. \square

5 Exact Neural Networks for Minimum Spanning Trees

In this section we prove Theorem 1 by showing correctness of Algorithm 1 and applying Proposition 3.

Proposition 4. *Algorithm 1 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. For the induction step, we separately show that Algorithm 1 does neither over- nor underestimate the true objective value. For this purpose we show that minimum spanning trees in the original graph and the smaller graph with updated weights can be constructed from each other in a way that is consistent with the computation performed by Algorithm 1.

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 $\mathbf{y}_n + \text{MST}_{n-1} \left((\mathbf{x}'_{ij})_{1 \leq i < j \leq n-1} \right)$ is indeed the MST value for n vertices.

First, we show that the value computed by Algorithm 1 is not larger than the correct objective value. To this end, 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

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

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 subset of edges v_iv_j , with $1 \leq i < j \leq n - 1$, 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 $v_iv_j \in T' \setminus E^*$, 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 $v_iv_j \in E^*$ 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_1v_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 $v_1v_i \in E^*$, we add the edge v_iv_n to T . Finally, we also add v_1v_n to T . Again it is immediate to verify that this construction results in fact in a spanning tree, and we obtain

$$\begin{aligned}
\sum_{v_iv_j \in T} \mathbf{x}_{ij} &= \mathbf{x}_{1n} + \sum_{v_1v_i \in E^*} \mathbf{x}_{in} + \sum_{v_iv_j \in T' \setminus E^*} \mathbf{x}_{ij} \\
&= \mathbf{y}_n + \sum_{v_1v_i \in E^*} (\mathbf{x}_{in} + \mathbf{x}_{1n} - \mathbf{y}_n) + \sum_{v_iv_j \in T' \setminus E^*} \mathbf{x}_{ij} \\
&= \mathbf{y}_n + \sum_{v_1v_i \in E^*} \mathbf{x}'_{1i} + \sum_{v_iv_j \in T' \setminus E^*} \mathbf{x}'_{ij}
\end{aligned}$$

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

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. \square

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

Theorem 1. *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, we have seen that Algorithm 1 performs the required computation. We show that $d(\text{MST}_n) = \mathcal{O}(n \log n)$, $w(\text{MST}_n) = \mathcal{O}(n^2)$, and $s(\text{MST}_n) = \mathcal{O}(n^3)$. Then, the claim follows by Proposition 3.

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 $d(\text{MST}_n) = \mathcal{O}(n \log n)$.

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 $w(\text{MST}_n) = \mathcal{O}(n^2)$.

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 $s(\text{MST}_n) = \mathcal{O}(n^3)$. \square

6 Exact Neural Networks for Maximum Flows

In this section we show that, given a fixed directed graph with n nodes and m edges, there is a polynomial-size NN computing a function of type $\mathbb{R}^m \rightarrow \mathbb{R}^m$ that maps arc capacities to a corresponding maximum flow. We achieve this by proving correctness of Algorithm 2 and applying Proposition 3. Before we start, we formally set definitions and notations around the Maximum Flow Problem.

6.1 The Maximum Flow Problem

Let $G = (V, E)$ be a directed graph with a finite node set $V = \{v_1, \dots, v_n\}$, $n \in \mathbb{N}$, containing 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 $\text{dist}_G(v, w)$ denotes the minimum number of arcs on any path from v to w in G .

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 $E^* := \{e \in E \mid c_e > 0\}$.

6.2 Analysis of the Subroutine

We first analyse the subroutine `FindAugmentingFlowk` given in Algorithm 3. The following theorem states that the subroutine indeed computes an augmenting flow fulfilling all required properties.

Theorem 5. *Let $(c_e)_{e \in E}$ 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 3 returns an s - t -flow $\mathbf{y} = (y_{uv})_{uv \in \vec{E}}$ with $-c_{vu} \leq y_{uv} \leq c_{uv}$ such that there is 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 $y_e = c_e$.*

The proof idea is as follows. We first show that we correctly track the excessive flow at each vertex u throughout the whole subroutine with variables \mathbf{Y}_u^i . This will be used to show two essential properties. First, by the way how we bound the amount of our pushes in terms of the fattest flow values, we ensure that the augmenting flow will only be positive on arcs contained in a shortest s - t -path. Second, by a careful analysis of the cleanup procedure, we obtain that the final flow fulfills flow conservation. It is then easy to see that it also respects the residual capacities.

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, 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 .

Proof of Theorem 5. It is easy to check that lines 7 to 11 from the algorithm do indeed compute the maximal flow value $\alpha_{i,v}$ 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 $\mathbf{z} = (z_e)_{e \in E}$ forms an s - t -flow in the residual network $G^* = (V, E^*)$ that satisfies $0 \leq z_e \leq c_e$. For this, recall that $\text{dist}_{G^*}(s, u)$ 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

$$\mathbf{Y}_u^j = \begin{cases} \sum_{e \in \delta_u^-} z_e - \sum_{e \in \delta_u^+} z_e & \text{if } j = k - \text{dist}_{G^*}(s, u), \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

holds throughout the execution of the subroutine.

Claim 6. *Equation (1) holds after the pushing procedure (lines 12 to 22).*

Proof of Claim 6. For $j < k - \text{dist}_{G^*}(s, u)$, there does not exist any u - t -path of length j (since $j + \text{dist}_{G^*}(s, u) < k \leq \text{dist}_{G^*}(s, t)$). 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 $j > k - \text{dist}_{G^*}(s, u)$, there is no s - u -path of length $k - j$ in the residual graph as $k - j < \text{dist}_{G^*}(s, u)$. The statement $\mathbf{Y}_u^j = 0$ then follows by an induction on $\text{dist}_{G^*}(s, u)$, as we show in the following:

For the base case of $\text{dist}_{G^*}(s, u) = 1$ 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 $\text{dist}_{G^*}(s, v) \geq \text{dist}_{G^*}(s, u) - 1$ and $i = j + 1 > k - \text{dist}_{G^*}(s, u) + 1 \geq k - \text{dist}_{G^*}(s, v)$). 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 $j = k - \text{dist}_{G^*}(s, u)$. In each iteration with $i = k - \text{dist}_{G^*}(s, u) + 1$ and $w = u$ we add \mathbf{f} to the flow value \mathbf{z}_{vu} and the same to $\mathbf{Y}_u^{k - \text{dist}_{G^*}(s, u)}$ and in each iteration with $i = k - \text{dist}_{G^*}(s, u)$ and $v = u$ we add \mathbf{f} to the flow value \mathbf{z}_{uw} and subtract \mathbf{f} from $\mathbf{Y}_u^{k - \text{dist}_{G^*}(s, u)}$. Hence, $\mathbf{Y}_u^{k - \text{dist}_{G^*}(s, u)}$ denotes exactly the excessive flow after the pushing procedure as stated in (1). \blacksquare

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 $i \neq k - \text{dist}_{G^*}(s, u)$) or $\mathbf{a}_{i-1,w} = 0$ because for $i = k - \text{dist}_{G^*}(s, u)$ there is no w - t -path of length $i - 1 = k - \text{dist}_{G^*}(s, u) - 1$ (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 7. Equation (1) holds in each iteration of the clean-up (lines 23 to 29).

Proof of Claim 7. First, we show that \mathbf{Y}_u^j stays 0 for $j \neq k - \text{dist}_{G^*}(s, u)$ by induction over $j = 2, 3, \dots, k$. 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 (1) 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 $\mathbf{Y}_u^{k - \text{dist}_{G^*}(s, u)}$. \blacksquare

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 (1) 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 (1), 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 6. 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 these nodes. We manually set (v^*, i^*) to (s, k) in the case that the set is empty.

Claim 8. *Some arc on a shortest path from v^* to t in G^* is saturated by \mathbf{y}_e .*

Proof of Claim 8. 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 5. □

6.3 Analysis of the Main Routine

The following theorem states that Algorithm 2 correctly computes a maximum flow. It turns out that the properties proven in Theorem 5 about the subroutine `FindAugmentingFlowk` are in fact sufficient to obtain correctness of the main routine as for the algorithms by Edmonds-Karp and Dinic; see, e.g., [41].

Theorem 9. *Let $G = (V, E)$ be a fixed directed graph with $s, t \in V$. For capacities $(\mathbf{v}_e)_{e \in E}$ as input, the MAAP given by Algorithm 2 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. [41, Theorem 8.5]. Since any simple path has length at most $n - 1$, it suffices to show the following claim.

Claim 10. *After iteration k of the **for** loop in line 5 of Algorithm 2, \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 [14, 41].

We will show Claim 10 about the outer **for** loop by induction on k using a similar claim about the inner **for** loop.

Claim 11. *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 11. Since (i) and (ii) hold at the beginning of the iteration, Theorem 5 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 \mathbf{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 5, 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 5, 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 11, we are now able to show Claim 10.

Proof of Claim 10. 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, \mathbf{x} is the zero flow, which is obviously feasible, and \mathbf{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, \mathbf{x} is a feasible s - t -flow with corresponding residual capacities \mathbf{c} and the distance from s to t in the residual network is at least k . Observe that by Claim 11, 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 11, $|E_k^*|$ 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 10 implies that, at the end of iteration $k = n-1$, the nodes s and t must be disconnected in the residual network. Hence, Algorithm 2 returns a maximum flow, which concludes the proof of Theorem 9. □

By applying the definition of our complexity measures to Algorithm 2 and the subroutine, we prove the following bounds.

Theorem 12. *For the complexity measures of the MAAP A defined by Algorithm 2 it holds that $d(A), s(A) \in \mathcal{O}(n^2m^2)$ and $w(A) \in \mathcal{O}(n^2)$.*

Proof. We first analyze the MAAP A' given in Algorithm 3. Concerning the complexity measures d and s , the bottleneck of Algorithm 3 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 2. 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 $w(A) \in \mathcal{O}(n^2)$. □

We remark that the reported complexity $w(A) \in \mathcal{O}(n^2)$ in Theorem 12 is actually suboptimal and can be replaced with $\mathcal{O}(1)$ instead, if the parallel **for** loops in the MAAP and the subroutine 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 2 and the subroutine 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 3 we obtain Theorem 2.

Theorem 2. *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 a vector of arc capacities to a vector of flow values in a maximum s - t -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.

Finally, observe that the total computational work of Algorithm 2, represented by $s(A)$ and, equivalently, the size of the resulting NN, differs only by a factor of n from the standard running time bound $\mathcal{O}(nm^2)$ of the Edmonds-Karp algorithm; see [41, 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, our subroutine requires $\mathcal{O}(mn)$ computational work.

7 Future Research

Our grand vision, to which we contribute in this paper, is to determine how the computational model defined by ReLU NNs compares to strongly polynomial time algorithms. In other words, is there a CPWL function (related to a CO problem or not) which can be evaluated in strongly polynomial time, but for which no polynomial-size NNs exist? Resolving this question involves, of course, the probably challenging task to prove nontrivial lower bounds on the size of NNs to compute certain functions. Particular candidate problems, for which the existence of polynomial-size NNs is open, are, for example, the Assignment Problem, different versions of Weighted Matching Problems, or Minimum Cost Flow Problems.

Another direct question for further research is to what extent the size of the NN constructions in this paper can be improved. For example, is the size of $\mathcal{O}(n^2m^2)$ to compute maximum flows best possible or are smaller constructions conceivable? Even though highly parallel architectures (polylogarithmic depth) with polynomial width are unlikely, is it still possible to make use of NNs' ability to parallelize and find a construction with a depth that is a polynomial of lower degree than n^2m^2 ?

We hope that this paper promotes further research about this intriguing model of computation defined through neural networks and its connections to classical (combinatorial optimization) algorithms.

References

- [1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, Upper Saddle River, New Jersey, USA, 1993.
- [2] 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*, pages 130–134, 1991.
- [3] Anonymous. Lower bounds on the depth of integral ReLU neural networks via lattice polytopes. In *Submitted to The Eleventh International Conference on Learning Representations*, 2023. under review.
- [4] M. Anthony and P. L. Bartlett. *Neural network learning: Theoretical foundations*. Cambridge University Press, 1999.
- [5] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee. Understanding deep neural networks with rectified linear units. In *International Conference on Learning Representations*, 2018.
- [6] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.
- [7] V. Beiu and J. G. Taylor. On the circuit complexity of sigmoid feedforward neural networks. *Neural Networks*, 9(7):1155–1171, 1996.
- [8] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv:1611.09940*, 2016.
- [9] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv:1811.06128*, 2018.

- [10] J. Berner, P. Grohs, G. Kutyniok, and P. Petersen. The modern mathematics of deep learning. *arXiv:2105.04026*, 2021.
- [11] D. Bertschinger, C. Hertrich, P. Jungeblut, T. Miltzow, and S. Weber. Training fully connected neural networks is $\exists\mathbb{R}$ -complete. *arXiv:2204.01368*, 2022.
- [12] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *arXiv:2203.00671*, 2022.
- [13] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [14] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.
- [15] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [16] S. Effati and M. Ranjbar. Neural network models for solving the maximum flow problem. *Applications and Applied Mathematics*, 3(3):149–162, 2008.
- [17] R. Eldan and O. Shamir. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pages 907–940, 2016.
- [18] P. Emami and S. Ranka. Learning permutations with sinkhorn policy gradient. *arXiv:1805.07010*, 2018.
- [19] S. Fomin, D. Grigoriev, and G. Koshevoy. Subtraction-free complexity, cluster transformations, and spanning trees. *Foundations of Computational Mathematics*, 16(1):1–31, 2016.
- [20] V. Froese, C. Hertrich, and R. Niedermeier. The computational complexity of relu network training parameterized by data dimensionality. *arXiv:2105.08675*, 2021.
- [21] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.
- [22] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.
- [23] 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 '21)*, volume 185 of *LIPICs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [24] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.
- [25] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science*, 21(1):105–111, 1982.
- [26] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.

- [27] B. Hanin. Universal function approximation by deep neural nets with bounded width and ReLU activations. *Mathematics*, 7(10):992, 2019.
- [28] B. Hanin and D. Rolnick. Complexity of linear regions in deep networks. In *International Conference on Machine Learning*, pages 2596–2604, 2019.
- [29] B. Hanin and M. Sellke. Approximating continuous functions by ReLU nets of minimal width. *arXiv:1710.11278*, 2017.
- [30] C. Hertrich, A. Basu, M. Di Summa, and M. Skutella. Towards lower bounds on the depth of ReLU neural networks. In *Proceedings of the 35th Conference on Neural Information Processing Systems (to appear)*, 2021. Full version: *arXiv:2105.14835*.
- [31] C. Hertrich and M. Skutella. Provably good solutions to the knapsack problem via neural networks of bounded size. In *Thirty-Fifth AAAI Conference on Artificial Intelligence (to appear)*, 2021. Full version: *arXiv:2005.14105*.
- [32] J. J. Hopfield and D. W. Tank. “Neural” computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, 1985.
- [33] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.
- [34] M. Jerrum and M. Snir. Some exact complexity results for straight-line computations over semirings. *Journal of the ACM (JACM)*, 29(3):874–897, 1982.
- [35] S. Jukna. Lower bounds for tropical circuits and dynamic programs. *Theory of Computing Systems*, 57(1):160–194, 2015.
- [36] S. Jukna and H. Seiwert. Greedy can beat pure dynamic programming. *Information Processing Letters*, 142:90–95, 2019.
- [37] M. P. Kennedy and L. O. Chua. Neural networks for nonlinear programming. *IEEE Transactions on Circuits and Systems*, 35(5):554–562, 1988.
- [38] S. Khalife and A. Basu. Neural networks with linear threshold activations: structure and algorithms. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 347–360. Springer, 2022.
- [39] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6348–6358. 2017.
- [40] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.
- [41] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 4th edition, 2008.
- [42] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.
- [43] S. Liang and R. Srikant. Why deep neural networks for function approximation? In *International Conference on Learning Representations*, 2017.

- [44] A. Lodi and G. Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017.
- [45] S. T. McCormick. Fast algorithms for parametric scheduling come from extensions to parametric maximum flow. *Operations Research*, 47(5):744–756, 1999.
- [46] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio. On the number of linear regions of deep neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2924–2932. 2014.
- [47] A. Mukherjee and A. Basu. Lower bounds over boolean inputs for deep neural networks with ReLU gates. *arXiv:1711.03073*, 2017.
- [48] A. Nazemi and F. Omid. A capable neural network model for solving the maximum flow problem. *Journal of Computational and Applied Mathematics*, 236(14):3498–3513, 2012.
- [49] 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*, pages 3737–3746, 2018.
- [50] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450*, 2017.
- [51] 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 '13)*, pages 765–774. Association for Computing Machinery, 2013.
- [52] I. Parberry, M. R. Garey, and A. Meyer. *Circuit complexity and neural networks*. MIT Press, 1994.
- [53] 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*, 2014.
- [54] 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*, pages 2847–2854, 2017.
- [55] T. Rothvoß. The matching polytope has exponential extension complexity. *Journal of the ACM (JACM)*, 64(6):1–19, 2017.
- [56] I. Safran and O. Shamir. Depth-width tradeoffs in approximating natural functions with neural networks. In *International Conference on Machine Learning*, pages 2979–2987, 2017.
- [57] T. Serra, C. Tjandraatmadja, and S. Ramalingam. Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pages 4565–4573, 2018.
- [58] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014.
- [59] J. S. Shawe-Taylor, M. H. Anthony, and W. Kern. Classes of feedforward neural networks and their circuit complexity. *Neural networks*, 5(6):971–977, 1992.

- [60] A. Shpilka and A. Yehudayoff. *Arithmetic circuits: A survey of recent results and open questions*. Now Publishers Inc, 2010.
- [61] K. A. Smith. Neural networks for combinatorial optimization: A review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.
- [62] M. Telgarsky. Representation benefits of deep feedforward networks. *arXiv:1509.08101*, 2015.
- [63] M. Telgarsky. Benefits of depth in neural networks. In *Conference on Learning Theory*, pages 1517–1539, 2016.
- [64] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. 2015.
- [65] D. P. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019.
- [66] D. Yarotsky. Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103–114, 2017.
- [67] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.