UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2024:35

Degree project 30 credits

Master`s Programme in Mathematics

August 2024

Learning Distributed Optimization with Graph Neural Networks

Henri Alexander Doerks

Department of Mathematics, Uppsala University

Supervisor: Jens Sjölund

Subject reviewer: Benny Avelin

Examiner: Julian Külshammer

Abstract

The emergence of large networked systems, where the agents typically have access only to local private information, together with the explosion in data dimensionality, has necessitated the development of efficient algorithms for distributed optimization. The Alternating Direction Method of Multipliers (ADMM) is well suited for this task, as it can be implemented in a distributed form, relying on decentralized communication between the agents. Still, the need and possibility for acceleration remain, especially when adapting the algorithm to certain problem classes. Learning-to-Optimize (L2O) has the potential to automate this process by leveraging machine learning methods. In this context, Graph Neural Networks (GNNs) seem to be a natural choice as they are able to learn and reason about graph-structured data, thereby outperforming traditional neural network architectures.

Thus, this thesis explores the abilities of GNNs to enhance the performance of distributed ADMM, with a specific focus on how GNNs operate. It adopts a model-based L2O approach that allows the GNN to influence certain components of the algorithm, while crucially preserving its convergence property. In particular, we propose two architectures for pivotal parts of ADMM, namely the step size and the local optimization step, that can be transferred to other graph algorithms. With experiments, we finally provide first insights into how much these strategies can improve the convergence speed of ADMM on unseen problems of the same class.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Jens Sjölund, for his exceptional mentorship during the thesis. From your intriguing project proposal to your continuous input of new ideas and approaches, you have significantly contributed to this work and enhanced its quality. Our enriching meetings and your expertise have guided me through the exciting world of machine learning. Thank you for all the time and effort you invested!

In addition, I would like to thank Daniel Escobar for our insightful discussions on ADMM and Paul Häusner for his excellent advice on Graph Neural Networks. A special thanks goes to all the members of Jens' group, who welcomed me with open arms and made my time here truly enjoyable. Finally, I am very grateful to my subject reviewer, Benny Avelin, for making this thesis possible and providing valuable feedback along the way.

Henri Doerks, August 2024

Contents

1 Introduction

Data is power. A statement whose relevance is inescapable in today's world, even if we wish it to be. But the truth goes deeper: only those who master the processing and utilization of their data capitalize on its power. The more data to be used, the greater the power. However, data is often collected and stored at different locations. Modern network technology enabled the sharing of knowledge from the data and the coordination of tasks between these locations. But in many applications, gathering the data at a central unit is not economical or not even possible (Shi et al., 2014).

The infeasibility of collecting data in one place has created a need for algorithms that allow the users to process the data in a decentralized manner at its storage locations while still extracting all relevant knowledge from the combined dataset. Typical use cases include smart energy management, aircraft- and transportation networks, the healthcare sector, telecommunication, wireless sensor networks, or the internet itself. Also, as a side effect of decentralization, the computations are often more robust against attacks and not as vulnerable to system failure (Nedić and Liu, 2018).

Such methods are commonly known as distributed optimization algorithms and rely on decentralized communication between collaborating locations, referred to as agents. With the recent explosion in the amount and dimensionality of the data another necessity for distributed optimization emerged. A single computing unit cannot efficiently handle such large-scale data anymore making it inevitable to divide the workload on multiple machines. Currently-hyped large language models are one prominent example that rely on parallelization but also advances in the aforementioned fields depend on processing large-scale datasets. A need for powerful and reliable algorithms that are both scalable and fast in their computations is evident.

Ongoing research has led to the development of various algorithms, addressing this need (see Nedić and Liu (2018) for an overview). Improvements, especially in convergence speed, are still of great value, and also tuning and customizing the algorithms to specific problem formulations is often not sufficiently understood (Ichnowski et al., 2021).

Although the topic can be advanced theoretically, this thesis follows an alternative approach with the potential to exceed the limits of analytic methods.

Rather than relying solely on theory, we use machine learning to accelerate existing optimization algorithms, an emerging research field known as Learning-to-Optimize (L2O). While machine learning has gained tremendous popularity due to recent breakthroughs, its ability to enhance optimization remains less commonly recognized (Chen et al., 2022). For a given entity that employs distributed optimization, the problems that arise over time often share similarities in terms of (a) the data to be processed and (b) the communication possibilities between the agents. This allows a machine learning model to learn from historical problems of the entity and automatically tune the algorithm to its specific needs, speeding up future computations for this class of problems.

Graph Neural Networks (GNNs) appear to be the natural choice for this task as they are capable of learning complex relationships in graph-structured data and align with the computational structure of distributed optimization algorithms. Once trained, they perform a deterministic sequence of operations on a given graph - exactly what an iterative algorithm for distributed optimization does. Such an alignment is proven to be beneficial for the learning capabilities of the neural network (Xu et al., 2020) and thus motivates our choice of GNNs.

The thesis explores the abilities of GNNs to improve distributed optimization in terms of convergence speed. We follow a model-based L2O approach that implements an already existing algorithm and allows the GNN to influence certain parts without destroying its convergence property. The algorithm we test in this thesis is a distributed version of the Alternating Direction Method of Multipliers (ADMM), a powerful algorithm of simple form that is well-suited for a wide range of convex, large-scale optimization problems (Boyd et al., 2011).

The literature has seen promising L2O-attempts for the classical ADMM algorithm (e.g. Yang et al. (2020)), and also a few GNNs have been used to accelerate other optimization methods (e.g. Cappart et al. (2023)). However, to the best of our knowledge, we have not seen any approach of GNN-based Learning-to-Optimize for distributed ADMM, motivating us to fill that gap. In particular, we focus on understanding of how GNNs work and can be utilized to learn distributed ADMM, leading to two different approaches. Ultimately, we answer the research question of whether GNNs are capable of improving distributed ADMM on unseen problems and compare the two

approaches against each other. Moreover, we provide inspiration for further research in this area.

The outline of the remainder of this thesis is as follows. In Chapter 2, we first make ourselves familiar with the distributed optimization problem, derive the later applied distributed version of ADMM, and prove its convergence. Chapter 3 provides a foundational introduction to GNNs with a specific focus on their computational structure. Subsequently, Chapter 4 presents the main contribution of this thesis, addressing the central research question of how GNNs can be applied to improve the introduced ADMM algorithm. Two approaches are proposed, along with their respective GNN architectures and loss functions. Furthermore, it suggests how an improvement can be measured for an algorithm like ADMM that calculates local solutions for each agent. Chapter 5 presents the obtained results of the two approaches. In particular, we investigate their generalization ability to unseen problems and compare the results to the baseline algorithm in terms of our improvement measure. Finally, in Chapter 6, we discuss the results and point out further research directions.

2 Algorithms for Distributed Optimization

In this chapter, we first specify our distributed optimization problem in more detail. After that, we introduce the Alternating Direction Method of Multipliers (ADMM), a generic algorithm for solving a wide range of convex optimization problems. Further, we continue to derive a specific version of ADMM that is particularly well-suited for solving our problem. This version of ADMM serves as the starting point for the primary goal of the thesis in the later chapters, namely to develop and learn an improved distributed optimization algorithm in terms of convergence speed.

2.1 Foundations of Distributed Optimization

A distributed, or multi-agent, optimization problem is based on a network of m different agents who jointly aim to solve the (constrained) problem

$$\min_{\tilde{x}\in\tilde{X}} \quad \tilde{f}(\tilde{x}) \qquad \text{where} \quad \tilde{f}(\tilde{x}) = \sum_{i=1}^{m} f_i(\tilde{x})\,. \tag{1}$$

The function $f_i : \tilde{X} \to \mathbb{R} \cup \{\infty\}$ for $\tilde{X} = \mathbb{R}^n$ is known only to agent i making it necessary to communicate with each other. For communication, the agents can use the underlying network structure, which is given by an undirected graph $G = (V, E)$. Here every node $i \in V$ symbolizes agent i and an edge $\{i, j\} \in E$ the opportunity for agents i and j to communicate.

Since communicating the f_i is not possible, every agent is left with its own private objective. By assigning a variable $x_i \in \tilde{X}$ to each agent i and forcing them to be equal, we can equivalently rewrite (1) as follows:

Distributed Optimization Problem

Given a network of agents, represented by $G = (V, E)$, and a set of local functions $f_i : \tilde{X} \to \mathbb{R} \cup \{\infty\}$, a distributed optimization problem is defined by:

$$\begin{aligned} \min_{x\in X} \quad & f(x) && \text{where} \quad f(x) = \sum_{i=1}^{m} f_i(x_i) \\ \text{s.t.} \quad & x_i = x_j && \forall i, j \in \{1, \dots, m\}\,. \end{aligned} \tag{2}$$

Now $x = (x_1^T, \dots, x_m^T)^T \in X := \tilde{X}^m$ and every agent can solve its own local optimization problem. For finding the global solution, the agents are then tasked to reach a consensus, i.e. $x_i = x_j$, that has the lowest global function value $f(x)$. There are many algorithms for solving (2), see Nedić and Liu (2018) for an initial overview. One of these algorithms is ADMM, which we introduce in detail in Section 2.2.

In practice, distributed optimization problems are often challenging because they arise for large n and complicated f_i that handle large amounts of private data. Additionally, the number of agents m can be very large. To accelerate convergence on such problems, it is particularly important that the algorithm allows the agents to solve their respective optimization problems in parallel. This is why we focus on deriving a parallel version of ADMM. For the remainder of the thesis, we make the following assumptions on problem (2):

Assumption 1 (Distributed Optimization problem)**.**

(i) The functions f_i are convex for every $i \in \{1, \dots, m\}$.

(ii) The function f is proper, i.e. $f(x) < +\infty$ for some $x \in X$ and $f(x) > -\infty$ for all $x \in X$, and closed, i.e. the set $\{x \in X : f(x) \leq \gamma\}$ is closed for any $\gamma \in \mathbb{R}$.

(iii) The underlying graph G is connected.

For example, note that any continuous function f over a closed domain is also closed (Bertsekas, 2009, Prop. 1.1.2.). Assumptions (i) and (ii) guarantee that f and every f_i are closed, proper convex functions. Together they imply that both the global optimization problem (2) and every local optimization problem $\min_{x_i \in \tilde{X}} f_i(x_i)$ have a global solution (Boyd et al., 2011, Ch. 3.2). Moreover, it follows from the convexity and $\tilde{X} = \mathbb{R}^n$ that every local solution $x^* \in \tilde{X}$ is also a global solution of the respective problem and that the set of global solutions is itself a convex set (Wright and Recht, 2022, Thm. 2.6). Both of these properties are important conditions for many optimization algorithms to guarantee convergence which is why they are typically found in the literature. In a more general context, we could also impose constraints on x such that $x \in X$ for some closed, convex set $X \subset \mathbb{R}^{mn}$. This would not affect the aforementioned statements and was omitted for simplicity.

Assumption (iii) is more specific to our distributed optimization problem. Since we only allow communication over the edges E an unconnected graph would imply that information cannot be exchanged between at least two sets of agents. Consequently, we can only expect to reach a consensus between the connected components of G but not between all agents in this case. Thus, we would not be able to construct an algorithm that solves problem (2).

2.2 Alternating Direction Method of Multipliers

In general, the Alternating Direction Method of Multipliers, in the following ADMM, is an algorithm for solving convex optimization problems of the specific form

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{s.t.} \quad & Ax + Bz = c\,, \end{aligned} \tag{3}$$

where the two variables are of arbitrary length $x \in \mathbb{R}^p$, $z \in \mathbb{R}^q$ and the r linear constraints on them are given by $A \in \mathbb{R}^{r\times p}$, $B \in \mathbb{R}^{r\times q}$ and $c \in \mathbb{R}^r$. As we see later in this section it is especially well-suited for solving distributed optimization problems like (2). In the following, we briefly discuss the classic ADMM algorithm and continue by explaining how it can be utilized in distributed optimization.

2.2.1 Classic Alternating Methods of Multipliers (ADMM)

The ADMM algorithm was originally proposed by Glowinski and Marroco (1975) and Gabay and Mercier (1976). It combines features of the dual ascent algorithm on the one hand, which allows for parallel optimization due to variable separation, and of the method of multipliers on the other hand, which converges under milder assumptions on the problem. Note that the global objective has to be separable into the two functions f, g operating on distinct sets of variables x, z. However, x and z can be connected to each other by adding respective constraints.

The algorithm operates on the augmented Lagrangian (Wright and Recht, 2022), given in the setting of (3) by

$$L_\alpha(x, z, \lambda) = f(x) + g(z) + \lambda^T(Ax + Bz - c) + \frac{\alpha}{2}\|Ax + Bz - c\|_2^2 \tag{4}$$

for a chosen step size $\alpha > 0$ in front of the quadratic penalty term. One iteration of ADMM is then given by the three steps (Boyd et al., 2011):

$$x^{k+1} = \arg\min_x L_\alpha(x, z^k, \lambda^k)\,, \tag{5a}$$

$$z^{k+1} = \arg\min_z L_\alpha(x^{k+1}, z, \lambda^k)\,, \tag{5b}$$

$$\lambda^{k+1} = \lambda^k + \alpha(Ax^{k+1} + Bz^{k+1} - c)\,. \tag{5c}$$

This is essentially one step of block coordinate descent on the variables x and z followed by an update of the Lagrange multiplier λ (Wright and Recht, 2022). Note that splitting the optimization is not equivalent to jointly optimizing over x, z, as one would do with the method of multipliers, but this still allows the algorithm to converge. Also note that a possible restriction of x on a closed, convex set X can be either encoded by additional constraints or by restricting the space in the x-update step.

2.2.2 ADMM for the consensus problem

In this section, we show how ADMM can be applied to solve our distributed optimization problem (2). Remember that we want each agent to independently solve their own local problem and reach consensus through communication. We show that ADMM can be brought into a form that makes this possible. At first, we familiarize ourselves with the algorithm on the distributed optimization setting and derive an initial distributed ADMM version, which relies on centralized communication over a newly introduced master node. At the end of this section, we extend the algorithm to a version that does not need such a central node and communicates exclusively over the given graph structure. However, this makes the algorithm slightly more involved.

Centralized distributed ADMM

To derive a parallel and distributed ADMM version, we rewrite problem (2) by introducing a help variable $z \in \mathbb{R}^n$. Obviously (2) is equivalent to

$$\begin{aligned} \min_{x,z} \quad & f(x) && \text{where} \quad f(x) = \sum_{i=1}^{m} f_i(x_i) \\ \text{s.t.} \quad & x_i - z = 0 && \forall i \in \{1, \dots, m\} \end{aligned} \tag{6}$$

with $g \equiv 0$, causing z to not directly affect the objective. The constraints can be brought into the form of (3) by defining $A := I_{nm}$, $B := -(I_n, \dots, I_n)^T \in \mathbb{R}^{nm \times n}$ and $c = 0$. The augmented Lagrangian of (6) becomes

$$\begin{aligned} L_\alpha(x, z, \lambda) &= \sum_{i=1}^{m} \left(f_i(x_i) + \lambda_i^T (x_i - z) \right) + \frac{\alpha}{2} \|x + Bz\|_2^2 \\ &= \sum_{i=1}^{m} f_i(x_i) + \lambda_i^T (x_i - z) + \frac{\alpha}{2} \|x_i - z\|_2^2 \end{aligned} \tag{7}$$

where the Lagrange multiplier $\lambda = (\lambda_1^T, \dots, \lambda_m^T)^T \in \mathbb{R}^{nm}$ is partitioned in the same way as the variable x. This gives the following ADMM iteration

$$x_i^{k+1} = \arg\min_{x_i} \left(f_i(x_i) + (\lambda_i^k)^T (x_i - z^k) + \frac{\alpha}{2} \|x_i - z^k\|_2^2 \right) \tag{8a}$$

$$z^{k+1} = \frac{1}{m} \sum_{i=1}^{m} x_i^{k+1} + \frac{1}{\alpha} \lambda_i^k \tag{8b}$$

$$\lambda_i^{k+1} = \lambda_i^k + \alpha (x_i^{k+1} - z^{k+1}) \,. \tag{8c}$$

By Assumption 1 (i) and (ii) the minimization problem in (8a) is solvable. Note in particular that the update step for x and λ can be performed separately for every i allowing for the desired, local computations of these iterates. This is because the penalty term splits into local penalties due to the introduction of the variable z as we have seen in (7). As a consequence, each minimization of x_i is independent of the other iterates x_j, $j \neq i$ for a fixed z. Therefore, we understand that the introduction of z is indeed crucial here. The more natural approach of just keeping the constraints $x_i = x_j$ for $i \neq j$ and trying to apply block separation along the given partition of $f(x) = \sum_{i=1}^m f_i(x_i)$ fails since the penalty term couples the different x_i.

This observation was generalized by Boyd et al. (2011, Section 4.4.1) in his criterion for block-separability of the x update. The latter is possible when $A^T A$ is block diagonal with respect to the partition causing the quadratic term $\|Ax\|_2^2$ to be separable. In fact, any $\tilde{A}$ encoding the constraint $x_i = x_j$ fails this criterion whereas $A^T A = I_{nm}$.

However, the reader might ask correctly how the introduction of z is in line with the communication structure of the problem since the z-update requires knowledge about all local iterates x_i^{k+1}, λ_i^k. Consequently, we have to include

a master node z in our graph which has an edge to all nodes, i.e. $\tilde{G} := (\tilde{V}, \tilde{E})$ where $\tilde{V} := V \cup \{z\}$ and $\tilde{E} := E \cup \{(z, x_i) | x_i \in V\}$. The algorithm would then only use these newly created communication ways between local nodes and master node, making the original communication structure obsolete. Note that therefore the connectivity assumption on the original graph G (Assumption 1 (iii)) is not necessary for the convergence of this algorithm since $\tilde{G}$ is connected over z anyway. Each iteration would result in exactly $2m$ sent messages (see Algorithm 4 in the Appendix):

(i) After the x-update each node sends $x_i^{k+1} + \lambda_i^k/\alpha \in \mathbb{R}^n$ to the master node,

(ii) After the z-update the master node sends $z^{k+1} \in \mathbb{R}^n$ to all m local nodes.

However, we can think of several reasons why we do not want or cannot allow this setup, for example limited communication bandwidth or privacy matters. In the following section, we therefore derive a distributed version of ADMM that uses decentralized communication.

Decentralized distributed ADMM

In the literature, there are many approaches on how to use ADMM for distributed optimization without a central node. Wei and Ozdaglar (2012) introduced a first convergent, decentralized ADMM algorithm coming at the cost of losing the ability to perform the x-update in parallel since it has to be done sequentially in a specific order. Later, Wei and Ozdaglar (2013) extended it to a random order, still not fully parallel. Shi et al. (2014) came up with a fully decentralized and parallel ADMM version. They derived an edge-based algorithm, proved its linear convergence and reduced it to a much simpler node-based version. Even though this algorithm is very similar, the following derivation is based on Makhdoumi and Ozdaglar (2017). It stands out for its less technical nature. Also, the convergence proof is based on the final node-based algorithm.

We start by stating the final algorithm of Makhdoumi and Ozdaglar (2017), adapted to our setting, as Algorithm 1. For its derivation, we again reformulate problem (2). This time we construct a matrix system $Ax = 0$ that is

equivalently encoding the original constraints. In contrast to the paper, we do this for a specific matrix A to simplify the arguments and portray more intuition whereas Makhdoumi and Ozdaglar (2017) derived it for a general communication matrix that has to fulfill several assumptions. Note, however, that the final algorithm remains the same for the more general matrix A.

Algorithm 1 Decentralized, distributed ADMM (DD-ADMM, parallel)

Initialize $x_i^0 \in \mathbb{R}^n, \lambda_i^0 = 0 \in \mathbb{R}^n$ for $i = 1, \dots, m$, and $k = 0$
Communicate x_i^0 to all neighbors $j \in N(i) \setminus \{i\}$
Initialize y_i^0 for $i = 1, \dots, m$ by

$$y_i^0 = \frac{1}{d_i + 1} \sum_{j \in N(i)} A_{ij} x_j^0$$

while stopping criterion is not reached **do**
    for $i = 1, \dots, m$ **in parallel do**
        Communicate y_i^k and λ_i^k to all neighbors $j \in N(i) \setminus \{i\}$
        Update local solution candidate x:

$$x_i^{k+1} = \arg\min_{x_i} \; f_i(x_i) + \sum_{j \in N(i)} (\lambda_j^k)^T (A_{ji} x_i) + \frac{\alpha}{2} \| A_{ji}(x_i - x_i^k) + y_j^k \|_2^2 \,.$$

        Communicate x_i^{k+1} to all neighbors $j \in N(i) \setminus \{i\}$
        Update local deviation-variable y:

$$y_i^{k+1} = \frac{1}{d_i + 1} \sum_{j \in N(i)} A_{ij} x_j^{k+1} \,.$$

        Update local dual-variable λ:

$$\lambda_i^{k+1} = \lambda_i^k + \alpha y_i^{k+1} \,.$$

    end for
    $k \leftarrow k + 1$
end while
return $x^k = [\, x_i^k \,]_{i=1}^m$

Before constructing A, we must define some basic notation for a given undirected graph G. With $N(i) := \{j \in V : \{i, j\} \in E\} \cup \{i\}$ we define the neighborhood of a node $i \in V$ that includes i itself. The degree of a node $i \in V$ is given by $d_i := |N(i)| - 1$ and is equal to the number of edges connected to i (we do not allow parallel edges). Moreover, we define $d_{\min} := \min_{i \in V} d_i$ as the minimal degree of all nodes.

Furthermore, the Laplacian matrix $\mathcal{L} \in \mathbb{R}^{m \times m}$ of a graph G, which is essential for the definition of A, is given by:

$$\mathcal{L}_{ij} := \begin{cases} d_i & \text{if } i = j \\ -1 & \text{if } j \in N(i) \setminus \{i\} \\ 0 & \text{otherwise}\,. \end{cases} \tag{9}$$

For a connected graph it holds for the kernel of $\mathcal{L}$ that $\ker(\mathcal{L}) = \operatorname{span}(\mathbf{1})$ where $\mathbf{1} \in \mathbb{R}^n$ is the vector containing n ones (Marsden, 2013, Thm. 3.10). This means that the only solution of the system $\mathcal{L}x = 0$ is a vector x whose entries are all equal. As a consequence, we get the following equivalent formulations, which introduce the matrix A:

Lemma 2.1. *For a connected graph G, we can equivalently formulate the constraints $x_i = x_j$ from Problem (2) as*

(a) $Ax = 0$ where $A := \mathcal{L} \otimes I_n \in \mathbb{R}^{mn \times mn}$ for the matrix tensor product $\otimes$,

(b) or $\alpha Q x = 0$ where $Q \in \mathbb{R}^{mn \times mn}$ is the square root of the singular value decomposition of $A^T D^{-1} A = U \Sigma U^T$ given by $Q := U \Sigma^{1/2} U^T$, and where $D := \operatorname{diag}(d_1 + 1, \ldots, d_m + 1) \otimes I_n$.

We derive the algorithm with the system $Ax = 0$ from (a) of Lemma 2.1. Part (b) is later applied in the convergence proof of the algorithm in Section 2.2.3. The proof of Lemma 2.1 is given in Appendix A.2. It is important to point out that in addition to Makhdoumi and Ozdaglar (2017) we also have to ensure that $Ax = 0$ follows from the constraint in (2). Otherwise, there could be possible solution candidates that fulfill the consensus constraint without being in the space defined by $Ax = 0$. Also note that the connectivity assumption of G is necessary for the equivalence since otherwise $\operatorname{span}(\mathbf{1}) \subsetneq \ker(\mathcal{L})$.

Derivation of decentralized distributed ADMM

We recognize that since $\mathcal{L}_{ij} = 0$ for $j \notin N(i)$, it holds for the i-th component of Ax that

$$0 = [Ax]_i = \sum_{j=1}^{m} A_{ij} x_j = \sum_{j \in N(i)} A_{ij} x_j \tag{10}$$

implying that $Ax = 0$ is equivalent to $\sum_{j \in N(i)} A_{ij} x_j = 0$ for all i. Here $A_{ij} = \mathcal{L}_{ij} I_n \in \mathbb{R}^{n \times n}$ is an $n \times n$ matrix by definition. Previously, we have already seen the trick that allows us to compute the iterates x_i^k in parallel. We again introduce a help variable z, but this time we simply have more of them, leading to the following system:

$$\begin{aligned} \min_{x,z} \quad & f(x) && \text{where} \quad f(x) = \sum_{i=1}^{m} f_i(x_i) \\ \text{s.t.} \quad & A_{ij} x_j - z_{ij} = 0 && \forall i \in \{1, \dots, m\}, j \in N(i) \\ & \sum_{j \in N(i)} z_{ij} = 0 && \forall i \in \{1, \dots, m\}\,. \end{aligned} \tag{11}$$

In fact, we introduce $d_i + 1$ variables for every node i, resulting in a total of $2|E| + m$ new variables $z_{ij} \in \mathbb{R}^n$. By assigning a Lagrange multiplier $\tilde{\lambda}_{ij} \in \mathbb{R}^n$ to the first set of constraints and another $\lambda_i \in \mathbb{R}^n$ to the second set of constraints, we obtain the following augmented Lagrangian:

$$\begin{aligned} L_\alpha(x, z, \tilde{\lambda}, \lambda) = \sum_{i=1}^{m} f_i(x_i) + \sum_{i=1}^{m} \sum_{j \in N(i)} \tilde{\lambda}_{ij}^T (A_{ij} x_j - z_{ij}) + \frac{\alpha}{2} \left\| A_{ij} x_j - z_{ij} \right\|_2^2 \\ + \sum_{i=1}^{m} \lambda_i^T \left(\sum_{j \in N(i)} z_{ij} \right) + \frac{\alpha}{2} \left\| \sum_{j \in N(i)} z_{ij} \right\|_2^2 . \end{aligned} \tag{12}$$

Here we have already used the separability of the penalty term, which is made possible by the auxiliary variables. By just assigning a Lagrange multiplier $\tilde{\lambda}_{ij}$ to the first constraints and enforcing the second ones directly, we would be able to derive an edge-based algorithm that is very similar to the one presented in Shi et al. (2014). This algorithm iterates over $(x_i, z_{ij}, \tilde{\lambda}_{ij})$ giving $m + 2(2|E| + m) = 3m + 4|E|$ variables of dimension n in total. However, we directly proceed to derive Algorithm 1, which allows us to eliminate the

edge-based iterates z_{ij} and $\tilde{\lambda}_{ij}$. We achieve this by reformulating the update rules of all iterates in terms of the node-based multiplier λ_i and a new node-based variable y_i, resulting in Algorithm 1 that only iterates over the three node-based variables (x_i, y_i, λ_i).

To reformulate the algorithm, we first look at the z-update. As mentioned before, we enforce the constraint $\sum_{j\in N(i)} z_{ij} = 0$ directly by restricting its search space respectively. By taking the derivative of L_α with respect to z_{ij} and solving for the minimum, this gives

$$\begin{aligned} z_{ij}^{k+1} &= \arg\min_{z_{ij}} L_\alpha(x^{k+1}, z, \tilde{\lambda}^k, \lambda^{k+1}) \\ &= A_{ij}x_j^{k+1} + \frac{\tilde{\lambda}_{ij}^k - \lambda_i^{k+1}}{\alpha} + \underbrace{\sum_{j\in N(i)} z_{ij}}_{=0} = A_{ij}x_j^{k+1} + \frac{\tilde{\lambda}_{ij}^k - \lambda_i^{k+1}}{\alpha}\,. \end{aligned} \tag{13}$$

Note that we inserted the $(k+1)$-th iterate of λ_i in the Lagrangian so we are in fact slightly deviating from the classic order of ADMM. This is necessary for the desired reformulation. The first Lagrange multiplier is updated in the usual manner by

$$\tilde{\lambda}_{ij}^{k+1} = \tilde{\lambda}_{ij}^k + \alpha(A_{ij}x_j^{k+1} - z_{ij}^{k+1})\,. \tag{14}$$

By inserting the definition of the z-update from (13) into this equation we obtain

$$\tilde{\lambda}_{ij}^{k+1} = \tilde{\lambda}_{ij}^k + \alpha\left(A_{ij}x_j^{k+1} - \left(A_{ij}x_j^{k+1} + \frac{\tilde{\lambda}_{ij}^k - \lambda_i^{k+1}}{\alpha}\right)\right) = \lambda_i^{k+1}\,. \tag{15}$$

If we initialize $\tilde{\lambda}_{ij}^0 = \lambda_i^0 = 0$ this gives $\tilde{\lambda}_{ij}^k = \lambda_i^k$ for any $k \in \mathbb{N}$, allowing us to replace $\tilde{\lambda}_{ij}^k$ in the entire algorithm. By doing this, summing over the z-update and using the enforced constraint $\sum_{j\in N(i)} z_{ij} = 0$ we further obtain

$$\begin{aligned} 0 &= \sum_{j\in N(i)} z_{ij}^{k+1} \\ &= \sum_{j\in N(i)} \left(A_{ij}x_j^{k+1} + \frac{\lambda_i^k - \lambda_i^{k+1}}{\alpha}\right) \\ &= \sum_{j\in N(i)} A_{ij}x_j^{k+1} + \frac{d_i+1}{\alpha}(\lambda_i^k - \lambda_i^{k+1})\,. \end{aligned} \tag{16}$$

This is equivalent to the update rule for the second Lagrange multiplier

$$\lambda_i^{k+1} = \lambda_i^k + \frac{\alpha}{d_i + 1} \sum_{j \in N(i)} A_{ij} x_j^{k+1} = \lambda_i^k + \alpha y_i^{k+1} \tag{17}$$

where we defined the mentioned new iterate as

$$y_i^{k+1} := \frac{1}{d_i + 1} \sum_{j \in N(i)} A_{ij} x_j^{k+1} \,. \tag{18}$$

Revisiting the z-update from (13) we can rewrite it with this newly discovered relation:

$$\begin{aligned} z_{ij}^{k+1} = A_{ij} x_j^{k+1} + \frac{\tilde{\lambda}_{ij}^k - \lambda_i^{k+1}}{\alpha} &\overset{(15)}{=} A_{ij} x_j^{k+1} + \frac{\lambda_i^k - \lambda_i^{k+1}}{\alpha} \\ &\overset{(17)}{=} A_{ij} x_j^{k+1} - y_i^{k+1} \,. \end{aligned} \tag{19}$$

Now we finally take a look at the x-update which is the first to be performed. It is given by

$$\begin{aligned} x_i^{k+1} &= \arg\min_{x_i} L_\alpha(x, z^k, \tilde{\lambda}^k, \lambda^k) \\ &= \arg\min_{x_i} \sum_{i=1}^m f_i(x_i) + \sum_{i=1}^m \sum_{j \in N(i)} (\tilde{\lambda}_{ij}^k)^T (A_{ij} x_j - z_{ij}^k) + \frac{\alpha}{2} \| A_{ij} x_j - z_{ij}^k \|_2^2, \\ &= \arg\min_{x_i} \; f_i(x_i) + \sum_{j \in N(i)} (\tilde{\lambda}_{ji}^k)^T (A_{ji} x_i) + \frac{\alpha}{2} \| A_{ji} x_i - z_{ji}^k \|_2^2 \,. \end{aligned} \tag{20}$$

Here we first used that the second type of constraints are independent of x. The second step follows from the fact that only some of the remaining summands are relevant for the separate optimization in x_i. Since we have rewritten $\tilde{\lambda}_{ij}^k$ (15) and z_{ij}^k (19) in terms of λ_i^k and y_i^k we can also reformulate the x-update to

$$x_i^{k+1} = \arg\min_{x_i} \; f_i(x_i) + \sum_{j \in N(i)} (\lambda_j^k)^T (A_{ji} x_i) + \frac{\alpha}{2} \| A_{ji}(x_i - x_i^k) + y_j^k \|_2^2 \,. \tag{21}$$

As a result we have update rules in the variables $(x_i^k, y_i^k, \lambda_i^k)$ that only depend on previous versions of each other making the computation of z_{ij}^k and $\tilde{\lambda}_{ij}^k$ irrelevant. In chronological order, this gives the previously introduced decentralized, distributed ADMM algorithm of Makhdoumi and Ozdaglar (2017). For convenience, we refer to Algorithm 1 as DD-ADMM or, informally, distributed ADMM in the remainder of this thesis.

Reflection on DD-ADMM

This algorithm indeed makes it possible to update all iterates in a decentralized and completely parallel manner. Each node just has to wait for all incoming messages of its neighbors to perform the next synchronous update. In contrast to the starting point of the derivation, the number of n-dimensional iterates decreased by $4|E|$ to now just $3m$ variables, since we replaced $(z_{ij}, \tilde{\lambda}_{ij})$ and incorporated (y_i, λ_i). All these $3m$ variables are sent to their neighboring nodes once per iteration, resulting in a total of $3(2|E|) = 6|E|$ sent messages.

Also, the algorithm maintained an intuitive character. The newly introduced variable y_i is nothing but a scaled version of the difference between the local iterates value x_i and their neighbors x_j, $j \neq i$. This becomes clear if one inserts the definition of the Laplacian which was used to construct A:

$$y_i^{k+1} = \frac{1}{d_i + 1} \left(d_i x_i^k - \sum_{j \in N(i) \setminus \{i\}} x_j^k \right) = \frac{1}{d_i + 1} \sum_{j \in N(i) \setminus \{i\}} \left(x_i^k - x_j^k \right) \tag{22}$$

Obviously, it does not follow from $y_i^k = 0$ that all nodes in the neighborhood $N(i)$ have to have the same value. However, if this holds for all nodes i the only possible solution is that $x_i^k = x_j^k$ for any $i, j \in \{1, \ldots, m\}$ resulting in a feasible solution of our system (11). This can be seen since we can write the update of the whole y vector compactly as $y^k = D^{-1} A x^k$. Consequently, $y^k = 0$ is equivalent to $Ax^k = 0$ which is the constraint in (11). We can therefore view y^k as a certificate for the primal feasibility of our solution.

Looking at the x-update we also see that the penalty term forces x_i^k in a direction that brings y_j^k closer to 0 for all $j \in N(i)$. This is very similar to the penalty term in the centralized algorithm of Boyd et al. (2011) which is forcing x_i^k closer to the average of all nodes.

Comparing this new algorithm as a whole with the parallel but centralized version of Boyd et al. (2011) we have an increase of $m - 1$ variables due to the introduction of y_i for each node compared to a central variable z. Also, the amount of messages is now dependent on the number of edges $m-1 \leq |E| \leq m(m-1)/2$ compared to a linear dependence on the number of nodes m. In contrast to the centralized algorithm, we expect the convergence speed to be influenced by the underlying communication graph as it is now

used for message passing. This is because we can only use the x-values of the neighbors' neighbors of node i that are processed in the y_j^k to find a consensus. Consequently, it takes longer until information from a distant node arrives and can be processed.

In this context, it is also important that all local update steps can be performed by aggregating the incoming messages from neighbors without saving all neighboring iterates. This aligns with the computation structure of graph neural networks, which is discussed in more detail in Chapter 4.

2.2.3 Convergence of distributed ADMM

As Boyd et al. (2011, Ch. 3.2.2) observe and as our experiments in Chapter 5 demonstrate, ADMM can be quite slow to converge to high accuracy. However, it can achieve modest accuracy in a relatively small number of steps, which is sufficient for many applications. In particular, we observe a significant impact of the step size $\alpha > 0$ on the convergence speed, with notable differences for optimal choices of this hyperparameter. These differences occur even for similar problem instances, providing an initial indication of potentially beneficial parts for tuning ADMM with Graph Neural Networks in Chapter 4.

We now focus on the convergence proof of DD-ADMM (Algorithm 1) from Makhdoumi and Ozdaglar (2017). For a proof of overall ADMM convergence and therefore also for the centralized version, we refer to Boyd et al. (2011, Appendix A).

Since we never assumed differentiability of f_i (or f), we potentially have to rely on subgradients of f_i in the optimization problem (21) for the x-update step. This can lead to fluctuations in the iterates, which is why it is common practice to prove the convergence of a time-average, or so-called ergodic average, of the x-iterates (Wright and Recht, 2022, sec. 9.2). We define it analogous to Makhdoumi and Ozdaglar (2017) as

$$\hat{x}^K := \left(\hat{x}_1^K, \ldots, \hat{x}_m^K\right), \qquad \text{where } \hat{x}_i^K := \frac{1}{K}\sum_{k=1}^{K} x_i^k. \tag{23}$$

The ergodic average is typically better behaved and smoothens the effect of fluctuations, which can arise due to the use of subgradients. Note that

each agent could easily construct its ergodic average vector $\hat{x}_i^K$ by recursively summing up its iterates x_i^k and therefore strictly guarantee the convergence of its solution by the following convergence result. For the theorem, we now use the result from Lemma 2.1 (b), which gives for a connected G that the system

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \alpha\, Qx = 0\,, \end{aligned} \tag{24}$$

is equivalent to system (2) which we aim to solve with the algorithm. We define $(x^*, r^*) \in \mathbb{R}^{2mn}$ as an optimal primal-dual solution of (24) that is used in the convergence theorem. Note that, due to the equivalence, the set of primal solutions is the same as in (2) and thus $x^* \in \mathbb{R}^{mn}$ is also an optimal solution of (2).

Additionally, we let the matrix $M \in \mathbb{R}^{mn \times mn}$ be the diagonal matrix that is defined block-wise by $M_{ik} = \delta_{ik} \sum_{j \in N(i)} A_{ji}^T A_{ji}$ for the Kronecker delta δ_{ik}, e.g. for $A := \mathcal{L} \otimes I_n$, M is given by

$$M = \operatorname{diag}(d_1^2 + d_1, \ldots, d_m^2 + d_m) \otimes I_n = (D^2 - D) \succ 0 \tag{25}$$

Note that $M - A^T D^{-1} A$ is positive semidefinite (Makhdoumi and Ozdaglar, 2017, Lem. 9) and therefore induces the seminorm $\| \cdot \|_{M - A^T D^{-1} A}$, where $\|x\|_H^2 := x^T H x$ for a positive semidefinite matrix H.

Theorem 2.2. *(Convergence of Algorithm 1) (Makhdoumi and Ozdaglar, 2017, Thm. 1)*
Let (x^*, r^*) be an optimal primal-dual solution of (24). Under Assumption 1, the function value of the ergodic average $\hat{x}^T$ of the sequence created by Algorithm 1 converges against $f(x^*)$ with rate $O(1/T)$, i.e. for any $T > 0$ we have

$$|f(\hat{x}^T) - f(x^*)| \leq \frac{\alpha}{2T} \left(\|x^0 - x^*\|_{M - A^T D^{-1} A}^2 + \|2r^*\|_2^2 \right). \tag{26}$$

Moreover, the ergodic average converges to a feasible solution, i.e. for any $T > 0$ we have

$$\|Q\hat{x}^T\|_2 \leq \frac{1}{2T} \left(\|x^0 - x^*\|_{M - A^T D^{-1} A}^2 + 2\|r^*\|_2^2 + 2 \right). \tag{27}$$

Theorem 2.2 shows that the ergodic average of the x-iterates created by Algorithm 1 converges toward an optimal solution of problem (2) with a

sublinear convergence rate of $O(1/T)$. This result can be further bounded by the magnitudes of some eigenvalue of the involved matrices and the minimal degree $d_{\min}$.

Corollary 2.3. *(Makhdoumi and Ozdaglar, 2017, Prop. 2)*
Let x^* be an optimal primal solution of (24) and Algorithm 1 initialized with $x^0 = 0$. Moreover, let $\tilde{\lambda}_{min} \in \mathbb{R}$ be the smallest non-zero eigenvalue of $A^T D^{-1} A$. Under Assumption 1, the ergodic average $\hat{x}^T$ of the sequence created by Algorithm 1 satisfies for any $T > 0$

$$|f(\hat{x}^T) - f(x^*)| \leq \frac{1}{T}\left(\frac{\alpha}{2}\left(\frac{d_{min}+2}{d_{min}+1}\right)\|A\|_2^2\,\|x^*\|_2^2 + \frac{2}{\alpha}\,\frac{u^2}{\tilde{\lambda}_{min}}\right) \tag{28}$$

and

$$\|Q\hat{x}^T\|_2 \leq \frac{1}{2T}\left(\left(\frac{d_{min}+2}{d_{min}+1}\right)\|A\|_2^2\,\|x^*\|_2^2 + 2\frac{u^2}{\alpha^2\tilde{\lambda}_{min}} + 2\right), \tag{29}$$

where u is an upper bound on the norm of the subgradients of f at x^*, i.e. $\|h\|_2 \leq u$ for any $h \in \partial f(x^*)$.

The proof of Corollary 2.3 is sketched in Appendix A.2. The remainder of this section deals with the proof of the important convergence result from Theorem 2.2. Without loss of generality, we assume $n = 1$ here. Moreover, we use the shorthand notation $\partial f(x)$ for the vector of subgradients of the local objectives f_i, i.e. $\partial f(x) := [h_i(x_i)]_{i=1}^m$ where $h_i(x_i) \in \partial f_i(x_i)$.

As a first crucial step toward the convergence result, we eliminate the dependence of x_i^{k+1} on y_j^k and λ_j^k for $j \in N(i)$ to write it as a linear combination of the past x-iterates $\{x^s\}_{s=0}^k$ and a perturbation caused by some subgradient $h_i(x_i^{k+1}) \in \partial f_i(x_i^{k+1})$. This result is summarized in the following Lemma

Lemma 2.4. *(Pertubated Linear Update) (Makhdoumi and Ozdaglar, 2017, Lem. 2)*
Under Assumption 1, the sequence of x-iterates generated by Algorithm 1 follows the update rule

$$x^{k+1} = \left(I_{mn} - M^{-1}(A^T D^{-1} A)\right) x^k - M^{-1}(A^T D^{-1} A)\sum_{s=1}^{k} x^s - \frac{1}{\alpha} M^{-1} h(x^{k+1}), \tag{30}$$

for some $h(x^{k+1}) \in \partial f(x^{k+1})$.

See Appendix A.2 for the proof of Lemma 2.4. Intuitively, this Lemma implies that if the sequence converges to a point $x^k = x^{k+1} = x^*$, then this point is a consensus point. We obtain this result by subtracting (30) for x^{k+1} and x^k from each other, which yields

$$\begin{aligned}(x^{k+1} - x^k) = \left(I_{mn} - M^{-1}(A^T D^{-1} A)\right)(x^k - x^{k-1}) \\ - M^{-1}(A^T D^{-1} A)x^k - \frac{1}{\alpha} M^{-1}\left(h(x^{k+1}) - h(x^k)\right) .\end{aligned} \tag{31}$$

This simplifies for a convergent sequence to

$$0 = M^{-1}(A^T D^{-1} A)x^k \,, \tag{32}$$

which is equivalent to the consensus condition $\alpha\, Qx = 0$. Lemma 2.4 is used in the following technical Proposition, which involves the ergodic average $\hat{x}^K$.

Proposition 2.5. *(Makhdoumi and Ozdaglar, 2017, Prop. 1 + Proof Thm.1) Let $r \in \mathbb{R}^{mn}$ be arbitrary and x^* an optimal primal solution of (24). Under Assumption 1, the ergodic average $\hat{x}^K$ of the x-iterates created by Algorithm 1 satisfies for any $K \geq 0$ that*

$$f(\hat{x}^K) - f(x^*) + \alpha r^T Q \hat{x}^K \leq \frac{\alpha}{2K}\left(\|x^0 - x^*\|^2_{M - A^T D^{-1} A} + \|r\|_2^2\right) . \tag{33}$$

Proposition 2.5 follows essentially from Lemma 2.4 and by the definition of a subgradient of a convex function f. Again, see Appendix A.2 for the proof. We conclude this section now with the proof of the convergence Theorem 2.2. In the proof, we apply Proposition 2.5 multiple times and make use of strong duality over the saddle-point property for (x^*, r^*).

Proof of Theorem 2.2. In order to prevent any potential confusion with the transpose, we replace the time variable $T \in \mathbb{N}$ with a capital K in this proof. Inserting $r = 0$ into Proposition 2.5 yields

$$f(\hat{x}^K) - f(x^*) \leq \frac{\alpha}{2K}\|x^0 - x^*\|^2_{M - A^T D^{-1} A}. \tag{34}$$

Since $\hat{x}^K$ is not necessarily a feasible solution of (24), $f(\hat{x}^K) - f(x^*) \geq 0$ does not need to hold and we must upper bound $f(x^*) - f(\hat{x}^K)$ as well to achieve convergence of the absolute value. Since f is convex and the constraints $\alpha\, Qx = 0$ are feasible for some relative interior point $x \in \mathrm{ri}(\mathbb{R}^{mn}) = \mathbb{R}^{mn}$,

problem (24), and equivalently problem (2), satisfies the Slater condition and therefore strong duality holds (Boyd and Vandenberghe, 2004, Ch. 5.2.3). Thus, since (x^*, r^*) is an optimal primal-dual solution of (24), it satisfies the saddle-point property (Boyd and Vandenberghe, 2004, Ch. 5.4):

$$L(x^*, r) \leq \underbrace{L(x^*, r^*)}_{= f(x^*)} \leq L(x, r^*) \qquad \forall x, r \in \mathbb{R}^{mn}\,, \tag{35}$$

where $L(x, r) = f(x) + r^T(\alpha Q x)$ is the unaugmented Lagrangian function of problem (24). Note that $Qx^* = 0$ since x^* is a feasible solution in particular. For $x = \hat{x}^K$ the second saddle point inequality in (35) implies

$$0 \leq f(\hat{x}^K) - f(x^*) + \alpha(r^*)^T Q \hat{x}^K \tag{36}$$

and equivalently

$$f(x^*) - f(\hat{x}^K) \leq \alpha(r^*)^T Q \hat{x}^K\,. \tag{37}$$

The latter shows that it is sufficient to bound the term on the right-hand side to obtain a bound on the absolute value. Adding $\alpha(r^*)^T Q \hat{x}^K$ to (36) on both sides gives

$$\begin{aligned} \alpha(r^*)^T Q \hat{x}^K &\leq f(\hat{x}^K) - f(x^*) + 2\alpha(r^*)^T Q \hat{x}^K \\ &\leq \frac{\alpha}{2K}\left(\|x^0 - x^*\|^2_{M - A^T D^{-1} A} + \|2r^*\|_2^2\right) \end{aligned} \tag{38}$$

where we used Proposition 2.5 again for the second inequality with $r = 2r^*$. Together with (34), this leads to

$$|f(\hat{x}^K) - f(x^*)| \leq \frac{\alpha}{2K}\left(\|x^0 - x^*\|^2_{M - A^T D^{-1} A} + \|2r^*\|_2^2\right).$$

concluding the first statement of Theorem 2.2. The bound on the feasibility violation of the ergodic average $\hat{x}^K$ also follows essentially from the saddle point inequality and from Proposition 2.5. For $Q\hat{x}^K \neq 0$, Proposition 2.5 with $r = r^* + \frac{Q\hat{x}^K}{\|Q\hat{x}^K\|_2}$ yields

$$\begin{aligned} f(\hat{x}^K) - f(x^*) + \alpha(r^*)^T Q \hat{x}^K + \alpha\|Q\hat{x}^K\|_2 & \\ \leq \frac{\alpha}{2K}\left(\|x^0 - x^*\|^2_{M - A^T D^{-1} A} + \left\| r^* + \frac{Q\hat{x}^K}{\|Q\hat{x}^K\|_2}\right\|_2^2\right) & \\ \leq \frac{\alpha}{2K}\left(\|x^0 - x^*\|^2_{M - A^T D^{-1} A} + 2\|r^*\|_2^2 + 2\right), & \end{aligned} \tag{39}$$

where we used in the last inequality that $\|a+b\|_2^2 \leq 2\|a\|_2^2 + 2\|b\|_2^2$, which follows from $0 \leq \|a-b\|_2^2$. Combining (39) with (36), the implication from the saddle point inequality, and dividing by $\alpha > 0$ concludes the second statement

$$\|Q\hat{x}^K\|_2 \leq \frac{1}{2K}\left(\|x^0 - x^*\|^2_{M-A^TD^{-1}A} + 2\|r^*\|_2^2 + 2\right).$$

□

2.2.4 Stopping criterion for distributed ADMM

In order to detect that the algorithm has converged to a (near-)optimal solution, it is necessary to use a reliable stopping criterion, given that the exact optimal solution of problem (2) is not known in practice. In fact, an exact solution might not even be computable without a distributed algorithm like ADMM when the f_i are only privately known to each agent as assumed. Therefore, this section derives such a stopping criterion for the DD-ADMM algorithm that can be employed in our experiments in Chapter 5.

For a general ADMM problem of the form (3) and a convex f and g (remember that we have $g = 0$) the primal feasibility

$$Ax^* + Bz^* = c \tag{40}$$

and the dual feasibility

$$\begin{aligned} 0 &\in \partial f(x^*) + A^T\lambda^* \\ 0 &\in \partial g(z^*) + B^T\lambda^* \end{aligned} \tag{41}$$

are both necessary and sufficient conditions for the optimality of a solution (x^*, z^*, λ^*) (Boyd et al., 2011, Ch. 3.3) (compare to the Karush-Kuhn-Tucker conditions). Given that ADMM does not ensure any of these conditions, this motivates us to define a primal and a dual residual that measure whether the solution is feasible. For distributed ADMM, Boyd et al. (2011, Ch. 7.1) showed that such residuals are given by

$$r^k = (x_1^k - \bar{x}^k, \ldots, x_m^k - \bar{x}^k), \qquad s^k = -\alpha(\bar{x}^k - \bar{x}^{k-1}, \ldots, \bar{x}^k - \bar{x}^{k-1}), \tag{42}$$

where $\bar{x}^k := \frac{1}{m}\sum_{i=1}^m x_i^k$ is the average over the x-iterates of all agents. Intuitively, we observe that the primal residual $r^k = 0$ if and only if the agents

have reached consensus, which is exactly what is encoded in the primal constraints. The dual residual s^k checks whether this consensus point is still changing. The two residuals could be used for our DD-ADMM algorithm. However, computing these would again require an undesired central communication step for the x-iterates to obtain the average $\bar{x}$. To avoid this, we propose the following local residuals

$$\tilde{r}_i^k = y_i^k, \qquad \tilde{s}_i^k = -\alpha \left(x_i^k - x_i^{k-1} \right) . \tag{43}$$

This definition ensures that the original residuals from (42) are equal to 0 if and only if $\tilde{r}_i^k$ and $\tilde{s}_i^k$ are equal to 0 for all i. At the end of Section 2.2.2, we have already acknowledged the equivalence of $y_i^k = 0$ for all i to $Ax^k = 0$ and thus to $r^k = 0$. Obviously, $x_i^k = x_i^{k-1}$ for all i is stronger than $\bar{x}^k = \bar{x}^{k-1}$. But under consensus, they are also equivalent.

These local residuals can be now computed in parallel by each agent, without the need for further communication. Each agent checks after each iteration if

$$\|\tilde{r}_i^k\|_2^2 \leq \frac{\epsilon^{\text{pri}}}{m} \qquad \text{and} \qquad \|\tilde{s}_i^k\|_2^2 \leq \frac{\epsilon^{\text{dual}}}{m} \tag{44}$$

for some bounds $\epsilon^{\text{pri}}, \epsilon^{\text{dual}} \in \mathbb{R}^+$ to ensure that their sum over all agents is smaller than the desired bounds. For simplicity, the agents report to some central operator of the algorithm in each iteration if both conditions are fulfilled. The algorithm is stopped once all agents have reported to stop in the same iteration. If necessary this could also be done over the given graph structure by sending certificates, but it is not considered here since no potentially private information is exchanged. Moreover, it should be considered to incorporate a maximum number of iterations in the stopping criterion as ADMM tends to converge slowly on certain instances.

The choice of the two upper bounds ϵ^{pri} and ϵ^{dual} depends on the problem instances and the desired accuracy of the solution. In particular, they should be of size $O(m)$.

3 Fundamentals of Graph Neural Networks

This chapter provides a concise introduction to Graph Neural Networks (GNNs), which serves as a foundation for the upcoming chapters of this thesis. In particular, it specifies the overall structure and framework of the later applied GNNs.

First introduced by Gori et al. (2005) and Scarselli et al. (2009), GNNs have gained increasing popularity in recent years, as they have been proven to be applicable and successful for a variety of different tasks. To name a few examples, we have seen effective use of GNNs for problems like machine translation, continuous control, learning the dynamics of physical systems, but also for combinatorial optimization and boolean satisfiability (Battaglia et al., 2018). Although initially designed to address the limitations of convolutional neural networks and adapt them to graph problems, GNNs offer a way of reasoning about and updating data represented by a graph G. A key advantage of GNNs compared to basic neural networks is that they are not limited to data of a fixed input size (Moore et al., 2023). As we see later in more detail, this is due to the fact that operations are defined on each component of the graph, allowing the same network to process data structured in very different graphs. This provides considerable flexibility for applications beyond structured data problems like image classification. Here, similar to the approach taken by convolutional neural networks, a pixel can be represented by a vertex of G and a neighboring relation between the pixels by an edge. In addition, it enables the handling of unstructured data, such as that found in computational tasks involving non-uniform meshes or social networks (Moore et al., 2023).

Unsurprisingly, this flexibility has motivated the introduction of many different GNN models in the literature. A unifying approach, which we focus on in this chapter, was given by Battaglia et al. (2018) through their Graph Network (GN) framework. In the remainder of this chapter, we first introduce these Graph Networks. We continue with a brief definition of Multi-Layer Perceptrons in order to finally explain how Graph Neural Networks are constructed from and generalize these building blocks.

3.1 Graph Networks (GNs)

Before we can specify how a GN operates, we must define what instances it works on and how those instances are represented. In the Graph Network framework by Battaglia et al. (2018), a graph $G = (V, E)$ comes with a feature vector $\mathbf{V} = [\mathbf{v}_i]_{i=1}^m$ where $\mathbf{v}_i \in \mathbb{R}^{n_v}$ is assigned to node $i \in V$ and with a feature vector $\mathbf{E} = [\mathbf{e}_{ij}]_{e_{ij} \in E}$ where $\mathbf{e}_{ij} \in \mathbb{R}^{n_e}$ is assigned to edge $e_{ij} \in E$. Additionally, there exists the possibility to assign a global feature $\mathbf{u} \in \mathbb{R}^{n_u}$ to G. The dimensions of the feature vectors for nodes, edges or the global feature may vary, i.e. it is allowed to have $n_v \neq n_e \neq n_u \in \mathbb{N}$. In such a case, the literature refers to this as a heterogenous Graph Network.

Consequently, an instance of a graph for a GN block is given by the tuple

$$(G, (\mathbf{V}, \mathbf{E}, \mathbf{u}))\,. \tag{45}$$

It is not necessary to include the nodes V in this tuple, as one could use the indices in the feature vector $\mathbf{V}$ to refer to a specific node. However, to emphasize that each instance depends on a graph G, we keep this notation in contrast to Battaglia et al. (2018). Also note, that this definition is assuming G to be a directed graph. For our undirected graph from Chapter 2, we simply insert directed edges in both ways.

3.1.1 Update structure of a full GN block

Given this representation of an instance, we can define how a GN block works on it. Essentially, a GN is nothing but an update function for the feature vectors mapping

$$(G, (\mathbf{V}, \mathbf{E}, \mathbf{u})) \mapsto (G, (\mathbf{V}', \mathbf{E}', \mathbf{u}'))\,, \tag{46}$$

while, crucially, not changing the graph itself. A full GN block contains three update functions, one for each feature vector, and three aggregation functions. We introduce them according to the order in which they are applied. Algorithm 2 provides the compact pseudo-code and Figure 1 a visualization of the following procedure.

Algorithm 2 Full GN block

function GRAPHNETWORK($G, \mathbf{E}, \mathbf{V}, \mathbf{u}$)
  for $e_{ji} \in E$ **do**
    $\mathbf{e}'_{ji} \leftarrow \phi^e\left(\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i, \mathbf{u}\right)$ ▷ 1. Edge update
  end for
  for $i \in V$ **do**
    $\bar{\mathbf{e}}_{\rightarrow i} \leftarrow \rho^{e\rightarrow v}\left(\mathbf{e}'_{*i}, \ldots, \mathbf{e}'_{*i}\right)$ ▷ 2. Edge-to-Node aggregation
    $\mathbf{v}'_i \leftarrow \phi^v\left(\bar{\mathbf{e}}_{\rightarrow i}, \mathbf{v}_i, \mathbf{u}\right)$ ▷ 3. Node update
  end for
  let $\mathbf{V}' = \{\mathbf{v}'_i\}_{i=1}^m$, $\mathbf{E}' = \{\mathbf{e}'_{ij}\}_{e_{ij}\in E}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\rightarrow u}\left(\mathbf{E}'\right)$ ▷ 4. (a) Edge-to-Global aggregation
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\rightarrow u}\left(\mathbf{V}'\right)$ ▷ 4. (b) Node-to-Global aggregation
  $\mathbf{u}' \leftarrow \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right)$ ▷ 5. Global update
  return $(G, \mathbf{E}', \mathbf{V}', \mathbf{u}')$
end function

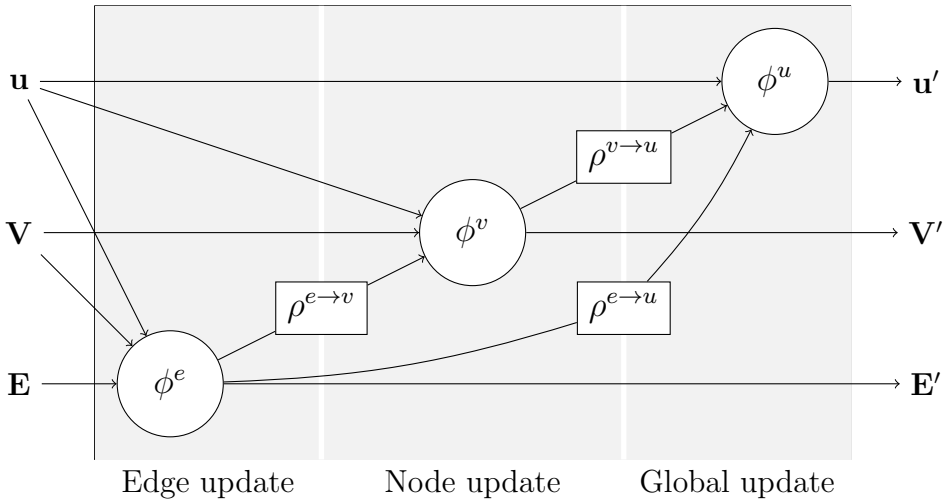

Figure 1: Computation structure of a full GN block as in Battaglia et al. (2018, Figure 4 (a)).

1. **Edge update function**: First, each edge feature vector is updated by the update function

$$\begin{aligned} \phi^e : \mathbb{R}^{n_e} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_u} &\to \mathbb{R}^{n'_e} \\ (\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i, \mathbf{u}) &\mapsto \mathbf{e}'_{ji} := \phi^e(\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i, \mathbf{u}) \,. \end{aligned} \tag{47}$$

   It takes the feature vector of the edge $e_{ji} \in E$, the feature vector of both adjacent nodes $i, j \in V$ and the global feature as input to produce a new edge feature vector $\mathbf{e}'_{ji}$ of possibly different dimension $n'_e \in \mathbb{N}$.

2. **Edge-to-Node aggregation function:** Next, for each node $i \in V$, a single edge feature vector $\bar{\mathbf{e}}_{\to i}$ of dimension $n_{e\to v} \in \mathbb{N}$ is created. This is done by aggregating the updated edge features over all incoming edges $e_{ji} \in E$ of i, given by $\mathbf{E}'_i := \{\mathbf{e}'_{ji} | e_{ji} \in E\}$, with the function

$$\begin{aligned} \rho^{e\to v} : \overbrace{\mathbb{R}^{n'_e} \times \ldots \times \mathbb{R}^{n'_e}}^{d_i \text{ times}} &\to \mathbb{R}^{n_{e\to v}} \\ \mathbf{E}'_i &\mapsto \bar{\mathbf{e}}_{\to i} := \rho^{e\to v}(\mathbf{E}'_i) \,, \end{aligned} \tag{48}$$

   Note that this aggregation function (like the following two) has to have the property that it can take a variable number of input arguments since the size of $\mathbf{E}'_i$ varies with the degree d_i of each node. The subscript notation with the arrow should indicate that the aggregated feature contains information from the incoming edges to node i.

3. **Node update function:** In the next step, each node feature vector is updated by the function

$$\begin{aligned} \phi^v : \mathbb{R}^{n_{e\to v}} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_u} &\to \mathbb{R}^{n'_v} \\ (\bar{\mathbf{e}}_{\to i}, \mathbf{v}_i, \mathbf{u}) &\mapsto \mathbf{v}'_i := \phi^v(\bar{\mathbf{e}}_{\to i}, \mathbf{v}_i, \mathbf{u}) \,. \end{aligned} \tag{49}$$

   It takes the respective aggregated edge feature, the old node feature and the global feature vector. Again, the updated vector $\mathbf{v}'_i$ can have a different dimension $n'_v \in \mathbb{N}$ than before.

4. (a) **Edge-to-Global aggregation function**: In preparation for the global update, all updated edge feature vectors $\mathbf{e}'_{ij}$ are aggregated to a single feature vector of dimension $n_{e\to u} \in \mathbb{N}$ by

$$\begin{aligned} \rho^{e\to u} : \overbrace{\mathbb{R}^{n'_e} \times \ldots \times \mathbb{R}^{n'_e}}^{|E| \text{ times}} &\to \mathbb{R}^{n_{e\to u}} \\ \mathbf{E}' &\mapsto \bar{\mathbf{e}}' := \rho^{e\to u}(\mathbf{E}') \,. \end{aligned} \tag{50}$$

(b) **Node-to-Global aggregation function**: Analogously, all updated node feature vectors $\mathbf{v}_i'$ are aggregated to a single feature vector of dimension $n_{v\to u} \in \mathbb{N}$ by

$$\begin{aligned} \rho^{v\to u}(\mathbf{V}') : \overbrace{\mathbb{R}^{n_v'} \times \ldots \times \mathbb{R}^{n_v'}}^{m \text{ times}} &\to \mathbb{R}^{n_{v\to u}} \\ \mathbf{V}' &\mapsto \bar{\mathbf{v}}' := \rho^{v\to u}(\mathbf{V}') \,. \end{aligned} \tag{51}$$

5. **Global update function**: Finally, the global feature is updated by

$$\begin{aligned} \phi^u : \mathbb{R}^{n_{e\to u}} \times \mathbb{R}^{n_{v\to u}} \times \mathbb{R}^{n_u} &\to \mathbb{R}^{n_u'} \\ (\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}) &\mapsto \mathbf{u}' := \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) \,, \end{aligned} \tag{52}$$

which takes the result of both aggregation functions from 4 and the old global feature vector. The resulting $\mathbf{u}'$ is of dimension $n_u' \in \mathbb{N}$

Note especially, that a GN block is designed to run on any instance of the form (45) given at the beginning of this section, so the same GN can be applied to several different instances. In particular, it is not tied to a specific graph G or a fixed number of nodes and edges but is still able to compute the updates according to the certain graph structure of the input instance. This is partly because the update functions are defined on a single node or edge and can therefore be applied to any arbitrary number of them (in parallel). But the main reason is the use of the aggregation functions, which can take a variable number of input arguments and reduce them into a single feature of fixed size that represents the aggregated information. This single argument is then passed to the subsequent update function, which allows the latter to be uniquely defined on a fixed amount of arguments for all features of the same type. For example, all nodes can be updated with the same update function incorporating the aggregated information of all its neighboring edges, despite the fact that different nodes may have different degrees across the graph. Naturally, the aggregation functions should be invariant to permutations of their inputs. Typical choices include the sum or maximum functions.

3.1.2 Other possible GNs

The previously introduced full GN block, which operates on a full instance and uses all three update and aggregation functions, has the most extensive

functionalities of a single block. However, the framework allows for a lot of variation, which means that not all functionalities need to be included. At first, we could work on instances that do not include all three kinds of feature vectors. For example, we could run a GN on an instance which only contains a feature vector for the edges and the nodes, but no global feature.

Likewise, one can decide not to use certain feature vectors in a particular update function, even though they are present in the chosen instance form. For example, we can include the edge feature in the node update function, but not in the global update function. If a feature vector that has been previously updated according to the chosen update order (here edges-nodes-global) is not used as an input for a subsequent update function then there is trivially no need to include the respective aggregation function, otherwise it is necessary. In the latest example, this means we have to define the Edge-to-Node but not the Edge-to-Global aggregation function.

Consequently, the framework offers a lot of flexibility and this is the reason why it unifies many different approaches of established GNN designs. For the sake of completeness, note that also the presented update order could be varied. For example, it could be reversed to (global-node-edges) (Battaglia et al., 2018, Ch. 3), which would then also create a need for a Node-to-Edge aggregation function. However, we stick to the presented order in this thesis.

A few examples of possible designs are shown in (Battaglia et al., 2018, Figure 4). In the following chapters, we apply the message-passing architecture shown in Figure 2, which does not involve a global feature $\mathbf{u}$. It follows the same computation graph as the so-called Non-local Neural Networks (Battaglia et al., 2018, Ch. 4.2.), but unlike them, is not restricted to a specific aggregation function.

Typically, multiple GN blocks are applied after each other. The output of the k-th applied GN block is then used as the input for the $(k+1)$-th GN block where $k \in \mathbb{N}$. For example, we can define a GN as the composition of $K \in \mathbb{N}$ possibly differently designed GN blocks

$$\mathrm{GN} := \mathrm{GN}_K \circ \mathrm{GN}_{K-1} \circ \ldots \circ \mathrm{GN}_1 \,. \tag{53}$$

The result is still a Graph Network as it transforms the feature vectors of the instance with respect to the input graph G. We refer to the k-th GN block as the k-th layer of the constructed GN. Incorporating multiple layers increases the complexity of the designed GN since previously updated features can

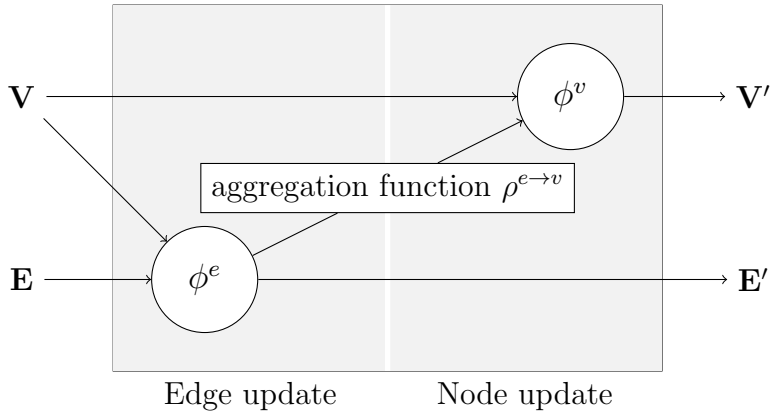

Figure 2: Computation structure of a message-passing GN block.

be passed around the graph structure of the input instance affecting later updates. This can lead to a highly intricate and interdependent updating procedure that follows the specific structure of the input graph G. However, it is not bound to a certain G, as again the same GN can be applied to different input instances.

For example, if we construct a GN consisting of K message-passing GN blocks shown in Figure 2 and apply it on a graph G, then K consecutive node updates are performed on all nodes of G. In particular, information of a certain node $i \in V$ is processed in all node features of nodes that are reachable over at most K edges from i, as in a breadth-first graph algorithm.

3.2 Multi-Layer Perceptrons (MLPs)

The other crucial building block needed to define a Graph Neural Network is a Multi-Layer Perceptron, or MLP for short. An MLP is a basic neural network of a certain structure. To give context, we now provide a brief introduction to how neural networks work and describe the architecture of an MLP. The majority of this chapter is based on Lindholm et al. (2022).

Like many other machine learning approaches, a neural network model is a function that maps the input arguments to a possibly multi-dimensional output, again following a specific computational structure. This mapping is essentially built by a linear regression together with some non-linear function,

the so-called activation function, that transforms the output of the regression. To formalize the latter, we briefly recall the concept of basic linear regression. In this context, an estimate $\hat{y} \in \mathbb{R}$ is computed by multiplying a p_0 dimensional input vector $x = (x_1, \dots, x_{p_0})^T \in \mathbb{R}^{p_0}$ with a weight vector $W = (W^1, \dots, W^{p_0})^T \in \mathbb{R}^{p_0}$, and by adding an offset term $b \in \mathbb{R}$:

$$\hat{y} = W^T x + b\,. \tag{54}$$

As a modification, the outcome $\hat{y}$ of the linear regression is now transformed by the non-linear activation function $h : \mathbb{R} \to \mathbb{R}$ leading to the new estimate

$$\hat{\hat{y}} = h(\hat{y}) = h(W^T x + b)\,. \tag{55}$$

The computational structure of this modified linear regression is visualized in Figure 3. In a neural network, not only one but multiple of these regressions are performed on the same input arguments, resulting in multiple regressed estimates. For p_1 simultaneous regressions, each $\hat{\hat{y}}_i$ is computed by its unique weight vector $W_i \in \mathbb{R}^{p_0}$ and offset term b_i for $i \in \{1, \dots, p_1\}$ but with the same activation function h:

$$\begin{aligned} \hat{\hat{y}}_1 &= h(W_1^T x + b_1) \\ \hat{\hat{y}}_2 &= h(W_2^T x + b_2) \\ &\vdots \\ \hat{\hat{y}}_{p_1} &= h(W_{p_1}^T x + b_{p_1})\,. \end{aligned} \tag{56}$$

Figure 4 visualizes this concept. To increase the model complexity, the obtained estimates can be used as input arguments for another set of regressions that come with new unique weight and offset vectors. In neural network terminology, this creates another layer in the network. The initial set of input arguments $x \in \mathbb{R}^{p_0}$ is called the input layer and is not counted towards the number of layers in the network. Thus, both Figure 3 and 4 are in fact single-layer neural networks with a different dimensional output layer.

As with GNs, we can repeat this approach and create multiple layers after each other. For $L \in \mathbb{N}$ total layers in a network, the parameters of this procedure can be compactly defined by $W^{(l)} = (W_1^{(l)}, \dots, W_{p_l}^{(l)}) \in \mathbb{R}^{p_{l-1} \times p_l}$ and $b^{(l)} = (b_1^{(l)}, \dots, b_{p_l}^{(l)})^T \in \mathbb{R}^{p_l}$, where $l \in \{1, \dots, L\}$. This results in the recursive computation rule

$$\hat{\hat{y}}_i^{(l)} = h\big((W_i^{(l)})^T\, \hat{\hat{y}}^{(l-1)} + b_i^{(l)}\big) \qquad i \in \{1, \dots, p_l\}\,, \tag{57}$$

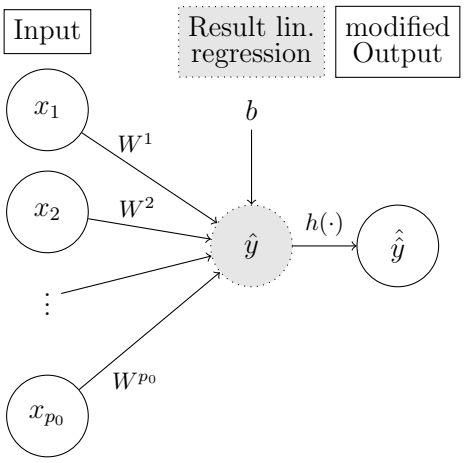

Figure 3: Computation graph of a single modified linear regression with activation function h for single estimate $\hat{\hat{y}}$, see (55).

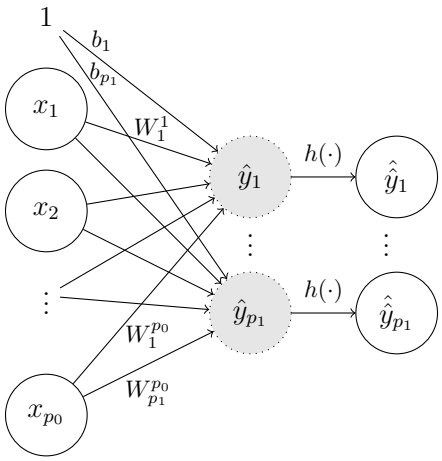

Figure 4: Computation graph of multiple simultaneous regressions of the form (55) for p_1 estimates $\hat{\hat{y}}_1, \dots, \hat{\hat{y}}_{p_1}$, see (56).

where we defined $\hat{\hat{y}}^{(0)} := x$ as the input. The constructed computational structure is shown in Figure 5. All real-valued variables $\hat{\hat{y}}_i^{(l)}$, for $l \in \{0, \dots, L\}$ and $i \in \{1, \dots, p_l\}$, are called neurons and are often visualized as a circular node. Typical examples for activation functions are the so-called sigmoid and ReLU function defined by

$$\sigma(z) = \frac{1}{1+e^{-z}}\,, \qquad \mathrm{ReLU}(z) = \max(0, z)\,. \tag{58}$$

It is common to use a different activation function $\tilde{h}$ for the output layer L depending on the particular task the network is designed for. In all other layers, the so-called hidden layers, usually the same h is applied.

The special characteristic of an MLP is, as in all neural networks we have seen so far, that all neurons of layer $l-1$ are connected to all neurons of the subsequent layer l. One speaks of fully-connected or dense layers. Other approaches, for example, convolutional layers, do not have this property since only a certain subset of neurons can influence a neuron of the following layer.

Thus, we can define a basic MLP by the activation functions $h, \tilde{h}$, the total number of layers $L \in \mathbb{N}$ and the number of neurons for each layer $p_0, p_1, \dots, p_L \in \mathbb{N}$ since this automatically determines the size of the required

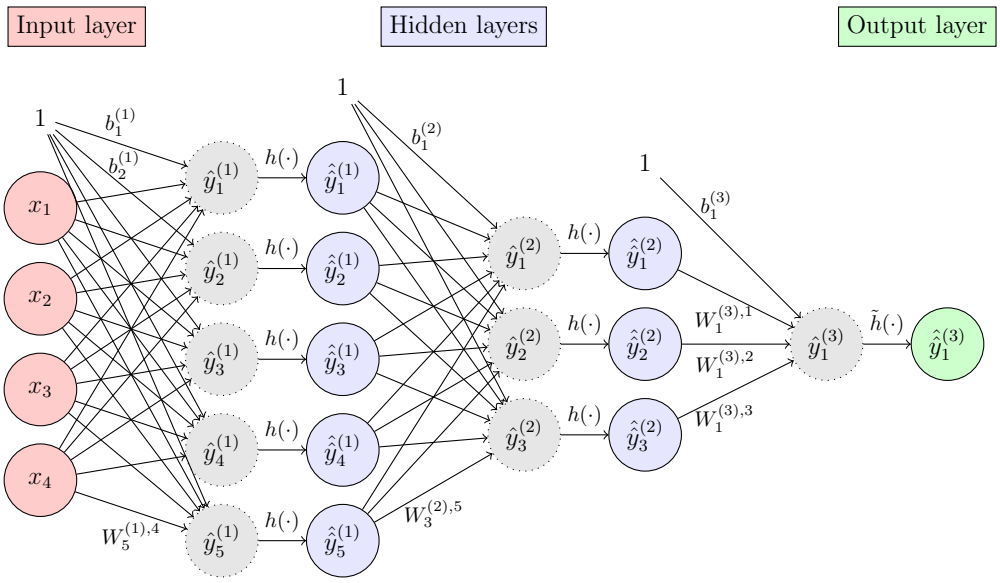

Figure 5: Computation graph of a 3-layer MLP with the architecture $[p_0, p_1, p_2, p_3] = [4, 5, 3, 1]$. Note that the added 1 is used to decode the addition of the offset vector and is not an additional neuron.

weight and offset parameters in each layer. A parameterization $\theta \in \Theta_{\text{MLP}}$ of a well-defined MLP is of the form

$$\theta = \big((W^{(1)}, b^{(1)}), \ldots, (W^{(L)}, b^{(L)})\big) \tag{59}$$

and therefore consists of a total of

$$\sum_{i=1}^{L} (p_{i-1} p_i + p_i) \tag{60}$$

real-valued parameters. Given a fixed parameterization $\theta^* \in \Theta_{\text{MLP}}$ of the network, the MLP becomes a deterministic mapping of an input vector x to the output vector $\hat{\hat{y}}^{(L)}$:

$$\begin{aligned} \text{MLP}(\,\cdot\,; \theta^*) : \mathbb{R}^{p_0} &\to \mathbb{R}^{p_L} \\ x &\mapsto \hat{\hat{y}}^{(L)} \qquad \text{(recursively by (57))}\,. \end{aligned} \tag{61}$$

For a fixed input vector, the output of such an MLP therefore uniquely depends on its parameterization θ^*, which is emphasized by the notation

$\text{MLP}(\,\cdot\,;\theta^*)$. In general, the parameterization $\theta \in \Theta_{\text{MLP}}$ is adjustable and the goal of training a neural network is to adapt θ in such a way that the output aligns with the task the MLP is designed for. This could be simple regression or classification tasks or a more abstract one as we have in Chapter 4. We do not go into the details of how an MLP is trained here, as we specify it directly for a GNN in the following Section 3.3.

Note that the non-linearity of the activation function h is crucial for the increased learning capabilities of an MLP. Without this non-linearity, an MLP would effectively reduce to a simple linear regression model, despite being written in a more complex form. Thus, the use of h is the reason that neural networks can approximate a wide variety of (non-linear) functions.

3.3 Graph Neural Networks (GNNs)

So far, any Graph Network (GN) from Section 3.1 is a deterministic update procedure of certain structure that does not include any possible learning capabilities. To change this and complete the definition of a Graph Neural Network (GNN), we incorporate at least one Multi-Layer Perceptron in one of the update functions of the included GN blocks. This makes the outcome of the GNN adjustable and dependent on the parameterization $\tilde{\theta} \in \Theta_{\text{MLP}}$ of the built-in MLP, where Θ_{MLP} depends on the chosen architecture of this MLP. Once a certain parameterization $\theta^* \in \Theta$ is inserted in the GNN it again operates deterministically on an instance as a specific GN. Thus, we can view a GNN as a collection of GNs. To emphasize this dependency of the GNN on $\theta \in \Theta$, we denote it, analogous to MLPs, as $\text{GNN}(\,\cdot\,;\theta)$.

The inserted MLP can determine the entire output of the respective update function or just set certain arguments, which are then processed in a subsequent deterministic procedure that determines the output. The parameter set Θ of the whole GNN is then the union of the parameter sets of the $q \in \mathbb{N}$ included MLPs

$$\Theta = \Theta^1_{\text{MLP}} \times \ldots \times \Theta^q_{\text{MLP}}\,. \tag{62}$$

Therefore, Θ consists of all weights and offset vectors of the included MLPs. Note that the number q and the specific design of the MLPs, especially the role of their outputs, are GNN-specific and important choices that we have to make when we define our GNN-models in Chapter 4. In general, also other

neural networks can be employed instead of an MLP. However, since it is often considered the standard choice, we use MLPs in this thesis.

As in any machine learning model, one has to train the parameterization $\theta \in \Theta$ to perform well for a certain task. This task is specified by a so-called loss function ℓ, which depends on θ and on the GNN output on an instance $\mathbf{X}$ from (45):

$$\ell(\theta, \mathbf{X}) = \varphi\big(\mathrm{GNN}(\mathbf{X}; \theta)\big)\,, \tag{63}$$

where φ is a real-valued function that is transforming the GNN output. Before training, a (random) parameterization $\theta^0 \in \Theta$ is chosen to initialize the network. In the training process, we aim to minimize the loss with respect to the current $\theta^k \in \Theta$ on a fixed set of instances, known as the training data, with the ultimate goal of also improving the performance on unseen instances, the test data. This is done iteratively by selecting instances (or batches of instances) from the training set and evaluating the loss of the current parameterization θ^k on them. Note that the latter implies running the parameterized $\mathrm{GNN}(\,\cdot\,; \theta^k)$ on these instances.

3.4 Basic machine learning terminology and tools

From this point forward, the process is analogous to training any other neural network model. Readers familiar with these conventional concepts may choose to skip the following section.

To update the parameterization to $\theta^{k+1} \in \Theta$, the gradient of ℓ with respect to θ^k is calculated. It is then used in a first-order optimization algorithm, typically some variation of stochastic gradient descent, which performs the update and aims to reduce the loss on the training instances. This procedure, commonly known as backpropagation, is usually already implemented in the respective machine learning libraries, e.g. PyTorch or JAX, and utilizes an automatic differentiation scheme to compute the gradient (Moore et al., 2023).

Backpropagation is typically applied not on a single instance, but on a batch of instances at once, to avoid large fluctuations caused by single outlier instances and due to computational advances. A complete backpropagation run on all training data instances is called epoch. After each epoch, the instances are reshuffled and put together in new batches. Then the process

is repeated until the chosen number of epochs is reached.

Another important hyperparameter that needs to be defined for training a GNN is the learning rate, which is the step size that the chosen optimizer algorithm takes in each training step.

As already mentioned, the goal of the training procedure is to also perform well on the test data, an ability of the network that we call generalization. Accordingly, the difference between the expected performance on test and training data is referred to as the generalization gap. In expectation, this difference is positive since the model is trained on the training data and therefore adapts to this specific data set. When a model adapts too strongly to the training data, it is called overfitting, which means that the model captures noise in the training data, resulting in poor performance on unseen data. Typically, the generalization gap increases with a more flexible model, e.g. with a more complex computation structure of the GNN and a larger parameter set Θ, ultimately leading to overfitting. On the other hand, a too-simple model might not be able to capture all meaningful patterns in the data, although it has a lower generalization gap. This behavior naturally leads to a trade-off in optimal model complexity, often called the bias-variance trade-off, as the expected test performance is influenced by both bias and variance[1] (Lindholm et al., 2022).

One tool to decrease the generalization gap is regularization, which refers to a variety of different techniques. A common form of regularization adds the norm of all parameters in the parameterization $\theta \in \Theta$ to the loss function. Instead of ℓ, we then use the regularized loss function

$$\ell^{\text{reg}}(\theta, \mathbf{X}) = \ell(\theta, \mathbf{X}) + \lambda^{\text{reg}} \|\theta\|_\star^2 . \tag{64}$$

Therefore, regularization penalizes large parameters within the MLP and ultimately leads to their reduction to zero if they do not show a meaningful influence on the outcome. Consequently, the variance of a model is reduced since only more meaningful parameters remain non-zero but simultaneously the bias of the model is increased.

If included, there are two hyperparameter choices to be made regarding this

[1]The variance term refers to how sensitive a model is to small fluctuations in the training data and the bias term is used to describe the capacity of a model to accurately represent the true underlying relationship between the data and the outcome.

form of regularization. We need to define the size of the regularization parameter $\lambda^{\text{reg}} \geq 0$, which determines the influence of the regularization, and the type of the applied norm. Typically, one uses either the L^1- or the L^2-norm.

All these hyperparameters, and the architecture of the GNN, are set and tuned in an iterative process with the help of a third data set, the validation data. Since this data is also not employed in the training process, it can be used to assess how well a certain model behaves on unseen data. The hyperparameters are then adjusted depending on the quality of the results. However, since it is repeatedly utilized to evaluate the generalization behavior of the different models, it is possible that the hyperparameters are defined in such a way that the model performs well on this specific data but not on other unseen data sets. To avoid this issue, the test data is rigorously separated from the training and validation data and only used to estimate the performance of the model once the training and the hyperparameter tuning processes have been completed. For more details on basic machine learning and neural network theory, we refer to Goodfellow et al. (2016).

4 Graph Neural Networks for ADMM

This chapter addresses the central research question of how the introduced GNNs can enhance the performance of distributed ADMM. Moreover, we investigate ways to measure such an enhancement. As a consequence, this chapter marks the main contribution of the thesis.

Having chosen a model-based learning approach means that we aim to keep the core structure of Algorithm 1 rather than develop a completely new method. We apply a technique called algorithm unrolling (Chen et al., 2022). In this method, one chooses a certain number of iterations for the underlying algorithm, in our case $K = 10$, that are performed. The GNN is trained only on these K iterations in the hope that a promising performance after the K iterations also translates to a faster convergence of the entire algorithm. For example, a similar unrolling technique was successfully applied by Sjölund and Bånkestad (2022) for matrix factorization. After these unrolled iterations, the quality of the computed iterates is evaluated by a loss function to improve the parameterization $\theta \in \Theta$ of the GNN in the training process. Once training is finished, one might use a different metric to measure if and how much the trained GNN improved compared to the baseline algorithm.

The outline of this chapter is as follows. First, we specify how we constructed our data set of optimization problems, which is later used to test the proposed method in Chapter 5. In Section 4.2, we then derive how a single iteration of Algorithm 1 can be written as a GN. In the subsequent Section 4.3, GNN designs for two different learnable parts of DD-ADMM are proposed, where each of these GNNs initially consists of $K = 10$ GNs from Section 4.2. After that, we define two loss functions to evaluate the GNN during training. We conclude this chapter in Section 4.5 by discussing different metrics to evaluate the performance of the GNN after training is finished.

4.1 Data generation of optimization problems

In this section, we briefly explain the form of our problem instances and how they were generated. When it comes to neural network learning, it is a typical assumption that the data follows a certain model or distribution. In particular, it is assumed that the test data is drawn from the same distribution as

the training data since otherwise we might not be able to find meaningful patterns that also hold for the unseen test data. How a machine learning model behaves when this is not the case, usually called out-of-distribution data, is a very relevant analysis but beyond the scope of this thesis. Instead, we investigate whether GNNs are capable of accelerating the performance of distributed ADMM at all. Consequently, we focus on the simpler case and artificially create data from the same distribution that we split into training, test and validation data.

In our specific task, one data point is an instance of our distributed optimization problem (2). Therefore, we actually create a distribution of optimization problems in this section. Across all instances, we must set the size of the solution space $\tilde{X} = \mathbb{R}^n$ to a fixed $n \in \mathbb{N}$. Each instance $\mathbf{X}$ consists of $m_{\mathbf{X}} \in \mathbb{N}$ agents that jointly aim to solve a problem of the type (2). For all instances, we assume that the local functions f_i for $i \in \{1, \dots, m_{\mathbf{X}}\}$ are of the same simple form

$$f_i(x_i) := \|Bx_i - b_i\|_2^2 \,, \tag{65}$$

where the shared parameter matrix $B \in \mathbb{R}^{n \times n}$ is the same for all agents on a certain instance and the private data vector $b_i \in \mathbb{R}^n$ is unique for each agent. To avoid any notational confusion, note that this data vector is unrelated to the offset vector in an MLP from Section 3.2. Additionally, note that f_i fulfills Assumption 1 since it is real-valued, continuous on the closed set $\tilde{X}$ and convex. If $B^T B \succ 0$ it is even strictly convex. In such a case, $B^T B$ is also invertible and thus the global distributed optimization problem (2) has a unique closed-form solution:

$$x^* = \frac{1}{m_{\mathbf{X}}} (B^T B)^{-1} B^T \sum_{i=1}^{m_{\mathbf{X}}} b_i \,. \tag{66}$$

In our data, B and $\mathbf{b} = [b_i]_{i=1}^{m_{\mathbf{X}}}$ are drawn from certain distributions, which lead to such a positive definite $B^T B$ with probability 1. The communication graph G of each instance is a random graph drawn from the Erdős–Rényi model $G(m_{\mathbf{X}}, p)$. Each graph from $G(m_{\mathbf{X}}, p)$ consists of $m_{\mathbf{X}}$ nodes and each possible undirected edge between the nodes exists with the probability $p \in (0, 1)$, which is fixed for all graph instances (Erdős and Rényi, 1960). In case the drawn G is not connected, it is discarded and redrawn from $G(m_{\mathbf{X}}, p)$ to fulfill the connectivity assumption. We can therefore define a single problem instance of our problem distribution by the tuple

$$\mathbf{X} = (G, B, \mathbf{b}) \in \mathcal{G} \times \mathbb{R}^{n \times n} \times \mathbb{R}^{m_{\mathbf{X}} \times n} \,, \tag{67}$$

where $\mathcal{G}$ is the set of graphs. Note that we have not included $m_{\mathbf{X}}$ directly in this tuple since it is implicitly encoded in G and $\mathbf{b}$. For the experiments in Chapter 5 we have created 11,000 such problem instances by the following explicit distributions:

- $n = 2$,
- $m_{\mathbf{X}} \sim U[6, 8]$, where U is a discrete uniform distribution,
- $G \sim G(m_{\mathbf{X}}, p)$ with $p = 0.4$ and redrawn if G was not connected,
- $B = [B_{ij}]_{i,j=1}^{n}$, where $B_{ij} \sim \mathcal{U}[0, 1]$ and $\mathcal{U}$ is a continuous uniform distribution and
- $b_i \sim \mathcal{N}(0, \Sigma)$ for $i \in \{1, \dots m_{\mathbf{X}}\}$, where $\mathcal{N}$ is a n-dimensional normal distribution with covariance matrix $\Sigma = 100 I_n$.

These problem instances are split into 9000 training instances, 1000 validation instances and 1000 test instances for evaluating our learning procedure in Chapter 5. The remainder of the chapter describes this generic approach for GNN-based algorithm unrolling. The approach and the proposed tools are not limited to our explicit problem instances. Occasionally, we use the instances to provide some intuition behind the method.

4.2 Distributed ADMM as Graph Network

In this section, we start addressing the question of how GNNs can improve the performance of distributed ADMM. Although there are undoubtedly other approaches, the crucial idea of the thesis is to exploit the specific structure of GNNs. Once equipped with a parameterization $\theta \in \Theta$ and applied to an instance $(G, (\mathbf{V}, \mathbf{E}, \mathbf{u}))$ they perform a sequence of operations to transform the feature vectors while respecting the underlying structure of each instance's graph G. This is very similar to how DD-ADMM operates on an input graph G, a property of the neural network referred to as algorithmic alignment (Xu et al., 2020). In particular, recall that a generic GN from Section 3.1, which is the essential building block of a GNN but without learnable parts, can be seen as a deterministic graph algorithm.

Therefore, the first step in constructing a GNN for model-based Learning-to-Optimize is to rewrite our DD-ADMM algorithm so that it follows the structure of a GN and actually becomes one itself. The following section shows how a single iteration of DD-ADMM (Algorithm 1) can be transformed into a GN. The resulting GN is already presented in Algorithm 3 and visualized in Figure 6.

Algorithm 3 Iteration of DD-ADMM as 2-block GN

function ADMM-GN($G, \mathbf{E}, \mathbf{V} = [(x_i^k, y_i^k, \lambda_i^k)]_{i=1}^m$)
▷ Start GN block 1
  for $e_{ji} \in E$ **do**
    $\mathbf{e}_{ji} \leftarrow \phi_1^e(\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i) := \left(A_{ji}^T \lambda_j^k, A_{ji}^T y_j^k\right)$ ▷ 1. Edge update 1
  end for
  for $i \in V$ **do**
    $\bar{\mathbf{e}}_{\to i} \leftarrow \rho^{e\to v}(\mathbf{E}_i) := \sum_{e_{ji}\in E} \mathbf{e}_{ji}$ ▷ 2. Aggregation
    $\mathbf{v}_i \leftarrow \phi_1^v(\bar{\mathbf{e}}_{\to i}, \mathbf{v}_i) := \left(x_i^{k+1}, y_i^k, \lambda_i^k\right)$, ▷ 3. Node update 1
      where x_i^{k+1} is given by (72)
  end for
▷ End GN block 1/Start GN block 2
  for $e_{ji} \in E$ **do**
    $\mathbf{e}_{ji} \leftarrow \phi_2^e(\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i) := \left(A_{ij} x_j^{k+1}, \star\right)$ ▷ 1. Edge update 2
  end for
  for $i \in V$ **do**
    $\bar{\mathbf{e}}_{\to i} \leftarrow \rho^{e\to v}(\mathbf{E}_i) := \sum_{e_{ji}\in E} \mathbf{e}_{ji}$ ▷ 2. Aggregation
    $\mathbf{v}_i \leftarrow \phi_2^v(\bar{\mathbf{e}}_{\to i}, \mathbf{v}_i) := \left(x_i^{k+1}, y_i^{k+1}, \lambda_i^{k+1}\right)$, ▷ 3. Node update 2
      where $\left(y_i^{k+1}, \lambda_i^{k+1}\right)$ is given by (77)
  end for
▷ End GN block 2
  return $(G, \mathbf{E}, \mathbf{V})$
end function

First, we need to define which feature vectors to include in an instance for this problem. Since Algorithm 1 updates three n-dimensional iterates per node, it is natural to equip each node with a $3n$-dimensional feature vector, i.e. $\mathbf{V} \in \mathbb{R}^{m\times 3n}$. Also, since we have no global communication in the algorithm, there is no need for a global vector.

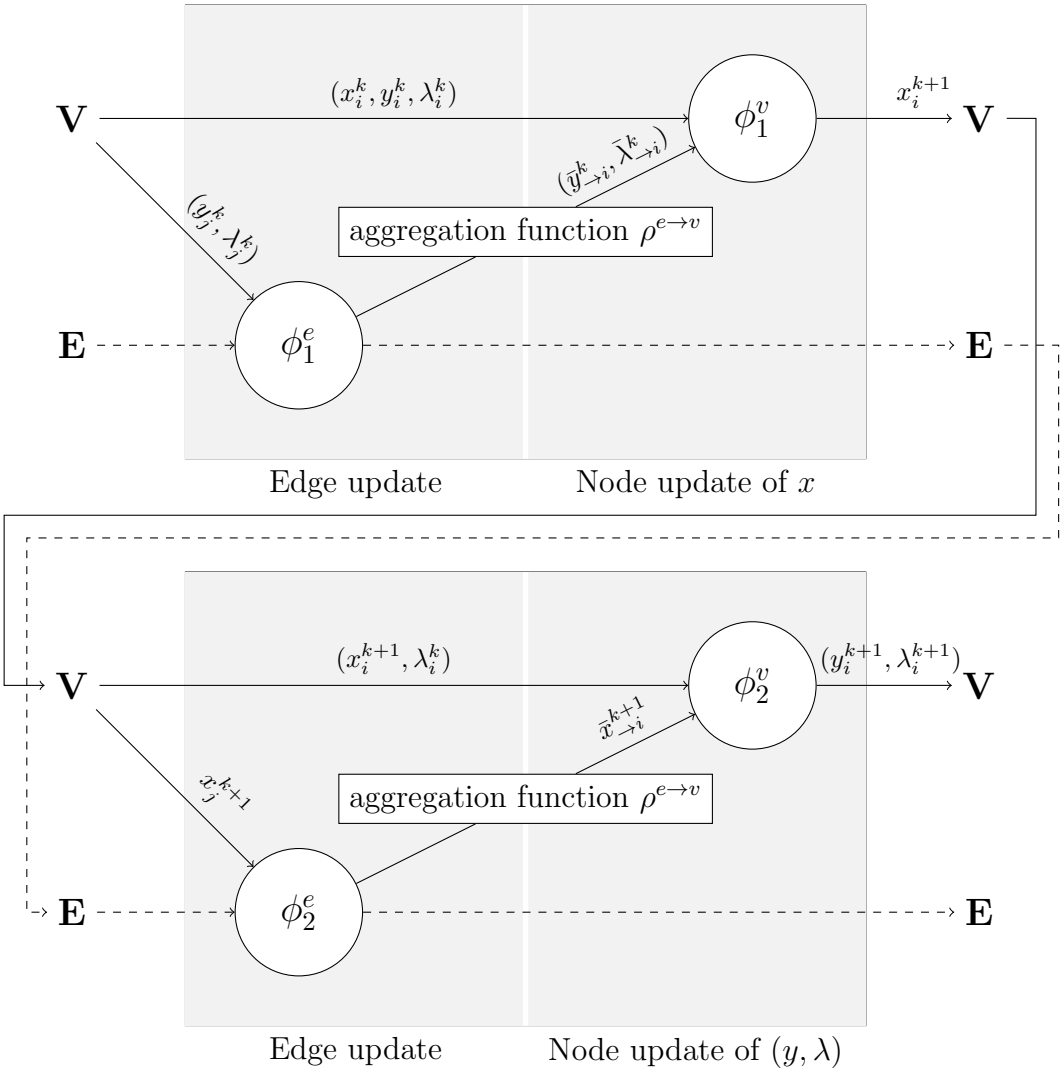

Figure 6: Computation structure of ADMM-GN (Algorithm 3). The dashed arrows for the edge features indicate that they are technically stored, but never affect subsequent edge updates.

One might expect that the same is true for the edges since there are no edge iterates, but in fact we need edge features to implement the communication steps of the algorithm in a GN. Since at most two n-dimensional iterates are

communicated to all neighbors at once (y^k and λ^k in line 6 of Algorithm 1), it is sufficient to set $\mathbf{E} \in \mathbb{R}^{|E| \times 2n}$, or $\mathbb{R}^{2|E| \times 2n}$ if G is undirected. As we see later, the edge feature only serves as a temporary variable for the communication steps and its actual entries do not matter once the subsequent aggregation function $\rho^{e \to v}$ is executed since its previous entries never influence the edge update function. Technically, this reintroduces $4|E|$ n-dimensional iterates, the same amount we initially got rid of in Section 2.2.2, to realize the message-passing within the GN. Note, however, that the need to store messages on the edges only arises since we want to follow the unifying GN block framework and its already existing implementation.

Now, we split the steps of a single DD-ADMM iteration into two parts

1. Communication of y^k, λ^k to neighbors and update of x (line 6-7),
2. Communication of x^{k+1} to neighbours and update of y, λ (line 8-10).

Each of these two blocks begins with an exchanged message and ends with a transformation of the node feature. This can be carried out by the previously shown message-passing GN block from Figure 2, which is why we reconstruct one iteration of Algorithm 1 by two consecutive GN blocks of this type. Note that there is no need to distinguish between $\mathbf{V}$ and $\mathbf{V}'$ or $\mathbf{E}$ and $\mathbf{E}'$ here since we do not change the dimension of the feature vectors and only update their entries in an iteration. In each block, messages are exchanged by overwriting the feature vectors of each directed edge $e_{ij} \in E$ by the respective feature vector of the sender node $i \in V$. Since this message should be used for the node update function, the aggregation function $\rho^{e \to v}$ has to be included to compute a single incoming message for each node. We naturally chose $\rho^{e \to v}$ as an aggregating sum in both blocks, i.e. for a fixed $i \in V$ we set

$$\bar{\mathbf{e}}_{\to i} = \rho^{e \to v}(\mathbf{E}_i) := \sum_{e_{ji} \in E} \mathbf{e}_{ji} \,. \tag{68}$$

This necessitates a slight rewrite of the algorithm since the update functions in Algorithm 1 assume knowledge over all required iterates from the node's neighbors, rather than a single aggregated feature per node.

We start by rewriting the x-update from (21). The included sum over features of neighboring nodes includes two types of summands that we have to revise.

The first one can be easily rewritten as follows

$$\sum_{j\in N(i)} (\lambda_j^k)^T (A_{ji}x_i) = (A_{ii}\lambda_i^k)^T x_i + \underbrace{\left(\sum_{j\in N(i)\setminus\{i\}} A_{ji}^T \lambda_j^k\right)^T}_{=:\bar{\lambda}_{\to i}^k} x_i \,, \tag{69}$$

where $\bar{\lambda}_{\to i}^k \in \mathbb{R}^n$ is defined as the incoming message containing the λ-features of all neighbors from node i. Note that A_{ji} has to be included in the message if $A \neq \mathcal{L} \otimes I_n$ and is also accessible. This is because the edge update of a GN block, defined in (47), can incorporate features from both the sender and the receiver node of an edge, particularly their indices i and j, in order to include the required global feature A_{ji}. The second summand, the penalty term for consensus, splits into the following

$$\begin{aligned}\sum_{j\in N(i)} \|A_{ji}(x_i - x_i^k) + y_j^k\|_2^2 = \sum_{j\in N(i)} &\|A_{ji}(x_i - x_i^k)\|_2^2 \\ &\underbrace{+\|y_j^k\|_2^2 + 2(A_{ji}(x_i - x_i^k))^T y_j^k}_{\text{includes features from neighbors}} \,.\end{aligned} \tag{70}$$

Since the x-update optimizes the equation with respect to x_i, we understand that the quadratic term $\|y_j^k\|_2^2$ is irrelevant for this update. The mixed term, which involves features from neighboring nodes, splits as follows

$$2\sum_{j\in N(i)} (A_{ji}(x_i - x_i^k))^T y_j^k = 2(x_i - x_i^k)^T \left(A_{ii}^T y_i^k + \underbrace{\sum_{j\in N(i)\setminus\{i\}} A_{ji}^T y_j^k}_{=:\bar{y}_{\to i}^k}\right), \tag{71}$$

where $\bar{y}_{\to i}^k \in \mathbb{R}^n$ is the desired message containing the y-features of all neighbors from node i. By the same argument as above, we can leave the constant term out in the optimization. To summarize the results, we have rewritten the x-update from (21) equivalently to

$$\begin{aligned}x_i^{k+1} = \arg\min_{x_i}\ f_i(x_i) + \left(A_{ii}\lambda_i^k + \bar{\lambda}_{\to i}^k + \alpha(A_{ii}y_i^k + \bar{y}_{\to i}^k)\right)^T x_i \\ + \frac{\alpha}{2}\sum_{j\in N(i)} \|A_{ji}(x_i - x_i^k)\|_2^2 \,.\end{aligned} \tag{72}$$

Note that for our f_i function form from (65) the unique solution of the x-update in (72) is given by:

$$x_i^{k+1} = (2B^TB+\alpha M_{ii})^{-1}\big(2B^Tb_i - A_{ii}\lambda_i^k - \bar{\lambda}_{\to i}^k + \alpha(M_{ii}x_i^k - A_{ii}y_i^k - \bar{y}_{\to i}^k)\big), \quad (73)$$

where the matrix is invertible since $B^TB \succeq 0$ and $M_{ii} = (d_i^2 + d_i) \otimes I_n \succ 0$. If the incoming message to the node update is $\bar{\mathbf{e}}_{\to i} = (\bar{\lambda}_{\to i}^k, \bar{y}_{\to i}^k)$, then all features required to perform the update are now available since A and α are globally known hyperparameters of the entire algorithm. Consequently, we define the edge update of the first GN block by

$$\phi_1^e\left(\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i\right) := \left(A_{ji}^T\lambda_j^k, A_{ji}^Ty_j^k\right), \quad (74)$$

such that $\bar{\mathbf{e}}_{\to i}$ is of the desired form. The subsequent node update function ϕ_1^v on the x-component of $\mathbf{v}_i$ is given by equation (72), or explicitly by (73) for our data. On the y- and λ-components, ϕ_1^v is defined as the identity, which completes the definition of the first GN block.

For the second one, we define

$$\phi_2^e\left(\mathbf{e}_{ji}, \mathbf{v}_j, \mathbf{v}_i\right) := \left(A_{ij}x_j^{k+1}, \star\right), \quad (75)$$

where $\star$ indicates that the second n-dimensional entry of the new edge feature $\mathbf{e}'_{ji}$ is irrelevant. Applying the aggregation function $\rho^{e\to v}$ gives

$$\bar{\mathbf{e}}_{\to i} = \left(\bar{x}_{\to i}^{k+1}, \star\right), \qquad \text{where} \quad \bar{x}_{\to i}^{k+1} := \sum_{j\in N(i)\setminus\{i\}} A_{ij}x_j^{k+1}, \quad (76)$$

so we can define ϕ_2^v on the y- and λ-components sequentially by

$$\begin{aligned} y_i^{k+1} &= \frac{1}{d_i+1}\left(\bar{x}_{\to i}^{k+1} + A_{ii}x_i^{k+1}\right), \\ \lambda_i^{k+1} &= \lambda_i^k + \alpha y_i^{k+1}. \end{aligned} \quad (77)$$

The just defined Algorithm 3 is a GN based on our defintion in Section 3.1 that performs a single, deterministic iteration of DD-ADMM on a given instance. Multiple iterations, as needed in our algorithm unrolling approach, can be performed by simply layering several of these GNs behind each other. This marks the first crucial step towards constructing a learnable GNN capable of improving the performance of DD-ADMM. As a side effect of strictly following the structure of a GN block, we have now also defined how messages are exchanged and processed.

4.3 Architecture of GNNs for different learnable parts

After defining and implementing the single iteration ADMM-GN from the previous section, we can now develop our unrolling approach. For the latter, we place $K = 10$ of these 2-block GNs behind each other, resulting in a bigger GN that performs $K = 10$ iterations of the algorithm. In this GN, we can include learnable parts by incorporating MLPs into certain update functions, as in our theoretical introduction of GNNs in Section 3.3. The key question to be addressed in this section is which parts we are trying to learn and how we can incorporate an MLP to do so.

The first question opens up a wide range of learning possibilities, of which we present two opposing approaches that are explored in this thesis. On the one hand, we could stick rigorously to the update rules of Algorithm 1 and only tune certain hyperparameters of the algorithm. This has the advantage that the trained algorithm can also be applied even after the $K = 10$ unrolling steps since the algorithm maintains its convergence property presented in Section 2.2.3. Possible hyperparameters are the step size $\alpha > 0$ or the communication matrix $A \in \mathbb{R}^{mn \times mn}$, which we set to $A = \mathcal{L} \otimes I_n$ in all versions. On the other hand, we could decide to be less restrictive in the iteration rule and let the GNN learn an entire update step (or a deviation from a step within a certain bound) while still performing the other deterministic algorithm steps. This approach offers the potential for larger improvements through greater freedom of choice but does not preserve the convergence property. A convergence guarantee can generally only be obtained by switching back to the original algorithm at some point.

The following Section 4.3.1 deals with the possibility of tuning the step size $\alpha > 0$ for DD-ADMM. Section 4.3.2 discusses the architecture of a GNN that learns the entire update step of the x-iterate. Both approaches share the similarity that the MLPs are incorporated in the node update function ϕ_1^v of the first GN block from Algorithm 3, which is performing the update of the x-iterate. Note again that both are completely generic approaches, not bound to the specific problem formulation from Section 4.1. Consequently, their idea can be transferred to other distributed optimization algorithms as long as they can be written as a GN. Therefore, the following sections also serve as an illustrative introduction to potential applications of GNNs in model-based Learning-to-Optimize.

4.3.1 Step size of distributed ADMM

As mentioned in Section 2.2.3, the step size $\alpha > 0$ has a huge impact on the convergence speed of distributed ADMM. Since it is the factor in front of the penalty term for consensus with the neighbors (see (21)), the step size determines the balance between how fast consensus is reached and how strong the influence of the local objective f_i is. A good choice for α can vary highly even for similar problems, making it hard to fine-tune this hyperparameter without further assumptions or specific knowledge about the problem. For example, Ghadimi et al. (2012) proved an explicit formula for the optimal step size $\alpha^* > 0$ in the consensus problem defined by $f_i(x_i) = (x_i - b_i)^2$, where $x_i, b_i \in \mathbb{R}$, on a k-regular graph G (compare to our chosen optimization function from (65)). Dropping the restrictive k-regular assumption on G already destroys the result, and to our knowledge, no optimal step size has yet been found for this generalized problem.

For our specific class of optimization problems defined in Section 4.1, we observe the big impact of the step size on the convergence speed in Figure 7. Although these problems are of a certain form and share many similarities, Figure 10 from our experiments in Chapter 5 shows how much the best choice of α varies over the first 30 instances of our training data set. Therefore, a general approach that selects a good step size for a given distributed optimization problem, similar to a heuristic, and in the best case automatically adapts to different distributions of problems, is of great interest and value. Recently, Ichnowski et al. (2021) applied a reinforcement learning approach to achieve this on quadratic programs, emphasizing the relevance of the targeted task. As introduced before, our approach is to train a GNN that automatically selects an appropriate step size for DD-ADMM. This section presents the architecture of the applied GNN in detail and discusses potential future enhancements.

Overall architecture of the GNN

The central idea of our algorithm unrolling approach is to introduce multiple MLPs into the GN, which consists of $K = 10$ iterations of Algorithm 3. As mentioned before, we always place the MLP into the node update function ϕ_1^v since α should stay the same over an iteration and ϕ_1^v marks the actual

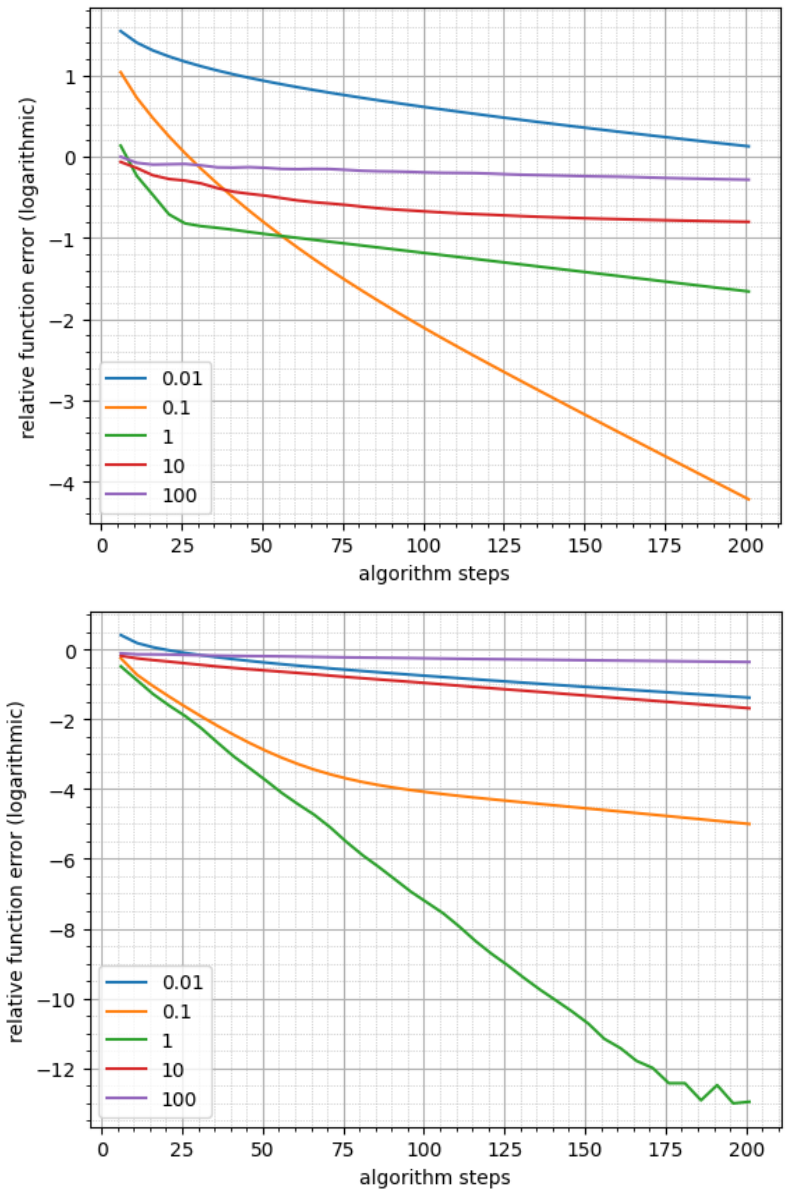

Figure 7: Convergence behavior of the relative function error $\overline{\Delta}_{\mathrm{rel},f}$ for different step sizes α on two training instances. Note the different magnitudes of the error between the instances and that the α with the lowest error after $K = 10$ steps is not necessarily equal to the best α.

start of one. Technically, we can adapt α in every iteration of the algorithm but our initial experiments showed that this barely improved convergence speed. Moreover, the training of the GNN becomes more complex and time-consuming, so we decided to make only two adaptations of the step size in total. Thus, the parameter space of our GNN is given by

$$\Theta = \Theta^1_{\text{MLP}} \times \Theta^2_{\text{MLP}} \,, \tag{78}$$

where the size of Θ_{MLP} depends on the architecture of the MLPs. Each MLP can potentially take all arguments available in a given node as input for its feature update. Further details are provided at the end of the section. Since we initialize all three iterates with a $0 \in \mathbb{R}^n$ vector, it has proven beneficial to first run one deterministic step of Algorithm 3 with a naively chosen step size, e.g. $\alpha = 1$. The first adjustment of α with an MLP is then done in the second iteration since the input already shows more information variability at this point and one also avoids possible issues with the computed gradients. Therefore, we place an MLP in the update function $\phi^v_{1,2}$ of the second iteration and one in $\phi^v_{1,6}$ of the sixth iteration of our algorithm. Figure 8 shows a schematic illustration of this procedure. The dashed frame around the MLPs and the GNs indicates that the MLPs are in fact part of the respective GN's node update function ϕ^v_1 and thus part of the GN itself.

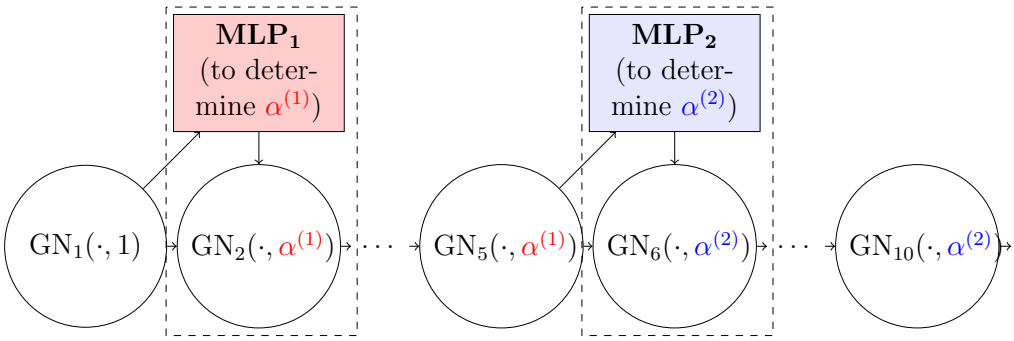

Figure 8: Visualization of the applied algorithm unrolling approach, in particular of the influence of the MLPs on the GNN.

The adapted step size after the second MLP, notated as $\alpha^{(2)}$, can be used for the remainder of the algorithm, even beyond the $K = 10$ unrolling steps, if additional deterministic steps are performed afterward. This motivates us to

investigate whether $\alpha^{(2)}$ can be advantageously applied in further algorithm steps, potentially resulting in a fine-tuned step size beyond unrolling. We address this additional research question through our experiments in Chapter 5.

Placing an MLP into a node update function means that every node routes its selected input feature through the identical MLP and not that every node is equipped with its own MLP since the node update function is identical for all nodes in a layer. Note that in a strictly distributed setting, this raises a need for distributed or federated learning to train the GNN, marking another important research direction of neural networks. We relax this restriction here while still ensuring that only the permitted information is used at a certain node. JAX provides the option to declare the updates as vectorized functions such that the same function is applied to all nodes in parallel.

The idea of the two learnable update functions is that the MLP first determines a new step size based on the input of the node update function and then performs the usual deterministic update $\phi_1^v(\,\cdot\,;\alpha)$ of the x-iterate depending on the new α. Currently, this would give a different step size for each node. Although using different α_i in the same iteration is also feasible (Barber and Sidky, 2024), the results showed considerable fluctuations. Therefore, we incorporate an averaging step over the calculated step sizes. For simplicity of the algorithm, we average over all α_i to obtain a global step size α for all nodes, which is technically an undesired central communication step. An alternative approach would be to implement an averaging step over the neighboring choices, which should already lead to improved results compared to independent, individual step sizes. The globally determined α is stored in each node for the subsequent update functions until the next adjustment of the step size. Figure 9 illustrates how the two designed learnable update functions $\phi_{1,2}^v$ and $\phi_{1,6}^v$ work. One might ask how the GNN knows that it is learning an optimized step size α for the algorithm. This is accomplished automatically by the way the output is incorporated in the next algorithm steps and by the upcoming definition of the loss function given in Section 4.4, which defines the overall goal of the GNN.

Architecture details of the included MLPs

Having specified the computational structure of the GNN, we may now focus on the design of the included MLPs itself. Although different architectures

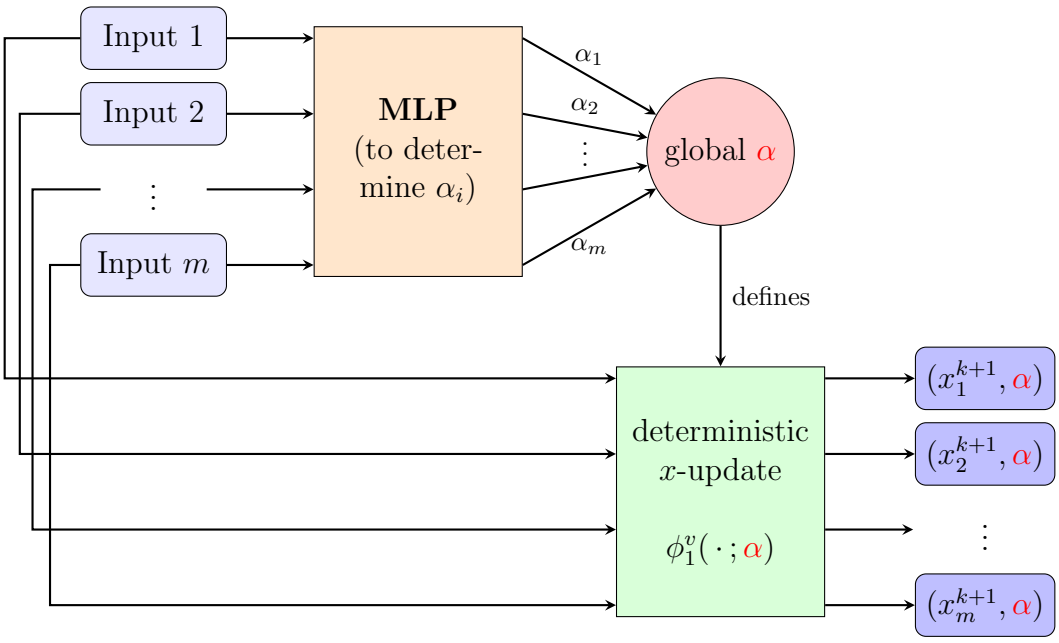

Figure 9: Visualization of the learnable node update functions $\phi_{1,2}^v$, $\phi_{1,6}^v$ that consists of 3 steps: (1) The input of each node $i \in V$ is processed by the for all nodes identical MLP to determine their local $\alpha_i > 0$. (2) The global step size α is calculated by averaging. (3) The input of each node is processed by the deterministic update function $\phi_1^v(\cdot;\alpha)$, which is defined by α and again identical for all nodes. The output is the updated x_i^{k+1} along with α itself, as it determines the subsequent update functions.

could be applied, we construct both of them in the same way for simplicity. The amount of neurons in the input layer is equal to the combined size of the input arguments in each node that are considered to be valuable for the adaptation of α. This number depends on the chosen class of problems from Section 4.1, which is why we set it along with the specific input arguments for our problem class at the end of this section. The number cannot vary for certain nodes since the identical MLP and the same function ϕ_1^v is applied to all nodes.

Between the input and the output layer, we can place any number of hidden layers of any possible size. At least one layer should be selected to allow non-linearity and sufficient learning capabilities in the network. Since each MLP

should output a single real number $\alpha_i > 0$, the output layer must consist of a single neuron and the exponential function proves to be a natural choice for its activation function to ensure strict positivity. As an activation function for the hidden layers of the MLP, we selected the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \,. \tag{79}$$

Other functions like ReLU or SeLU could also be used here. All layers are fully connected as in any MLP. Note that we could also incorporate any other type of neural network in the GN since the use of an MLP is not required. Many other tuning techniques for neural networks, such as dropout or layer normalization, can be considered but were not employed here. In conclusion, the only architectural questions that remain unanswered for now are those regarding the number of hidden layers and the number of neurons included in these layers. We address this matter in Section 5.2.1 and present results for different choices.

We now conclude this section by defining the features that we considered as input for our MLPs. Naturally, it receives all the features defined as input for the deterministic x-update function ϕ_1^v being

$$(\bar{\mathbf{e}}_{\to i}, \mathbf{v}_i) = \big((\bar{y}^k_{\to i}, \bar{\lambda}^k_{\to i}), (x^k_i, y^k_i, \lambda^k_i)\big) \tag{80}$$

at iteration k. Also, the old value for the global $\alpha \in \mathbb{R}^+$ and the degree $d_i \in \mathbb{N}$ are transmitted. The index of the node should not be considered since we want the result of the MLPs to be invariant to graph isomorphism. Specifically for the problem class from 4.1, we additionally input the private information vector $b_i \in \mathbb{R}^n$ and the value $Bx^k_i \in \mathbb{R}^n$ of the old x-iterate to learn about the private objective f_i. In total, we therefore need $7n + 2$ neurons in the input layer of each MLP.

Of course, the use of additional input features about the private objective f_i could be considered. Including local degree features, such as the variance in the degree of neighboring nodes or the maximum and minimum degree of neighbors, may also be valuable. These features were successfully employed by Häusner et al. (2024). Moreover, one could include global graph attributes like the maximal distance between two nodes in the graph, as this affects the speed at which information is shared across the graph.

4.3.2 Update step for local solution candidate x_i

A contrasting approach to learning suitable hyperparameters for distributed ADMM is to modify an entire step of the algorithm, which gives greater freedom to the MLPs. The update step of the x-iterate is the most computationally demanding step since it requires solving an optimization problem in x_i itself (see (21)). Moreover, for the convergence of the algorithm, it can be sufficient to solve this optimization problem exactly only to a certain extent (Eckstein and Bertsekas, 1990, Thm. 1). Thus, we chose the entire x-update step ϕ_1^v as the second learnable component of DD-ADMM in this thesis.

Similarly to the structure for learning α in Figure 8, we incorporate two MLPs into the GN, which consists of $K = 10$ iteration of Algorithm 3. Unlike the previous section, these MLPs determine the entire output of the associated ϕ_1^v. All other x-updates, so every function besides $\phi_{1,2}^v$ and $\phi_{1,6}^v$, remain deterministic as defined in Algorithm 3. The step size is naively set to $\alpha = 1$ for the entire algorithm. Therefore, the GNN performs a total of K iteration of DD-ADMM, during which two x-updates are carried out in an unrestricted manner without following any algorithm rules. Alternatively, one could learn a deviation within a certain range from each x-update while still computing the correct deterministic update step, similar to Banert et al. (2020).

In contrast to the previous section, the output layer of the MLPs has to consist of n neurons since the output should represent the updated $x_i^{k+1} \in \mathbb{R}^n$. Moreover, the activation function of the output layer has to be removed or set as the identity since each entry of x_i^{k+1} could be any real number. Besides that, we use the same architecture of the MLP, in particular the same input arguments as specified in 4.3.1 except for the now constant step size $\alpha = 1$.

An alternative approach, often referred to as layer-wise weight tying or weight sharing, is to run the same parametrized GNN in all K unrolling steps (or again starting with the second iteration). This means that only a single MLP is incorporated into the whole GNN to learn the x-update but it is employed repeatedly before the performance is evaluated with the loss function. Consequently, weight-sharing more rigorously realizes the learning of a modified iteration rule as the same transformation is applied in each step. Due to the time constraints of the thesis, it was not explored.

The approaches from this and the previous section could also be combined to additionally learn the step size α. In this case, the output layer of each MLP would consist of $n + 1$ neurons. The neuron corresponding to α would then define the step size of the algorithm until a new adjustment is made as in Figure 8.

4.4 Loss function

While the architecture of the GNN influences the structure and capability of learning, the loss function of the network is the crucial factor that determines the direction in which the parameterization $\theta \in \Theta$ of the GNN is optimized in the training process. Ideally, this should be a function that can be calculated without much computational effort. Our application opens up the possibility to use many different loss functions. In this section, we specify and explain the rationale behind our two proposed candidates. Hereby, we address the question of what we consider to be an improved algorithm.

Traditionally, one distinguishes between so-called supervised and unsupervised loss functions. In supervised learning, each instance used for training is associated with an (output) label. Here, the goal of the training procedure is to describe the relation between the input of each instance, or in our case between an entire problem instance, and the label in the best possible way such that the network can predict the label on unseen data. In our case, it is not obvious how to define such a label for our problem instances from Section 4.1. This is because we aim to outperform the original algorithm on these instances and, especially in terms of the x-update, do not know a priori how to achieve this. Therefore, our loss functions do not have a classic supervised character but may still include a label.

In unsupervised learning, we have no true solution or label but still attempt to adapt $\theta \in \Theta$ such that it lowers a loss function both on training and unseen instances. This typically involves identifying patterns in the data that are relevant to the value of the loss function (Lindholm et al., 2022).

Since our overall aim is to improve the convergence speed of our DD-ADMM algorithm, the performance should be captured in the loss function as the loss defines our learning objective. Before we can find a suitable function, it is essential to specify what we consider to be an improvement in terms

of convergence. This question is closely related to the following Section 4.5 where we discuss possibilities to measure an improvement of the GNN after training. A desirable goal is to compute more accurate solutions using the same computational cost or to reach the same accuracy while using fewer computational resources. Once trained, the $K = 10$ steps of our GNN are essentially as expensive as $K = 10$ steps of the deterministic GN with naive step size $\alpha = 1$, so we naturally use the same computational resources in all parameterized GNNs. Thus, we evaluate the GNN by the accuracy of its solutions after the K unrolling steps and recognize a more accurate solution as an improvement.

But in the context of distributed optimization, it is even necessary to define what we consider as a higher level of accuracy. This is because in distributed optimization we have to minimize the function f but also ensure the consensus among the solution candidates x_i (primal feasibility). A first approach to measuring this and designing a loss function is via the size of the residuals for primal and dual feasibility from Section 2.2.4. However, we take a slightly different approach to measuring the optimality of the solution.

Instead of using the norm of the dual residual, we exploit the fact that we can easily compute the global solution $x^* \in \mathbb{R}^n$ for our problems from Section 4.1. Therefore, we can calculate both the relative localization error of the solution candidates x_i^K and of their respective global function values $f(x_i^K)$. If the average over all m agents of both relative errors is reduced, we certainly have a solution that is closer to optimality. We propose the following loss function

$$\begin{aligned}\ell_{\beta,\gamma,\delta}(\theta, \mathbf{X}) &= \beta\, \overline{\Delta}_{\mathrm{rel},x} + \gamma\, \overline{\Delta}_{\mathrm{rel},f} + \frac{\delta}{m_{\mathbf{X}}} \sum_{i=1}^{m_{\mathbf{X}}} \|\tilde{r}_i^K\|_2^2 \qquad (81)\\ &= \frac{\beta}{m_{\mathbf{X}}} \sum_{i=1}^{m_{\mathbf{X}}} \frac{\|x_i^K - x^*\|_2^2}{\|x_i^0 - x^*\|_2^2} + \frac{\gamma}{m_{\mathbf{X}}} \sum_{i=1}^{m_{\mathbf{X}}} \left(\frac{f(x_i^K) - f(x^*)}{f(x_i^0) - f(x^*)} \right)^2 + \frac{\delta}{m_{\mathbf{X}}} \sum_{i=1}^{m_{\mathbf{X}}} \|y_i^K\|_2^2 \,,\end{aligned}$$

where $\beta, \gamma, \delta \geq 0$ are three hyperparameters, $x^* \in \mathbb{R}^n$ is the unique global solution of problem $\mathbf{X}$ and $x_i^K, y_i^K \in \mathbb{R}^n$ are given by the GNN output

$$\mathrm{GNN}(\mathbf{X}; \theta) = (G, \mathbf{E}, \mathbf{V} = [(x_i^K, y_i^K, \lambda_i^K)]_{i=1}^m) \,. \qquad (82)$$

Hence, evaluating $\ell_{\beta,\gamma,\delta}$ includes running the GNN, and thereby DD-ADMM for K steps, to measure the accuracy of the final output. In each training step,

$\theta \in \Theta$ is adapted in such a way that the loss is reduced and thus the accuracy of the output is increased. Since this is the goal in each learning approach, it does not matter which part of the algorithm the GNN is supposed to learn and we can employ $\ell_{\beta,\gamma,\delta}$ in all GNNs from Section 4.3. The GNN explores the influence of each learnable part on the final output and adjusts $\theta \in \Theta$ accordingly. By the way the MLP output is used in the subsequent algorithm steps, it recognizes which MLP output leads to a lower loss on a certain instance. Consequently, with this loss function, the GNN is forced by the algorithm design and its own structure to assign the MLP outputs to values that correspond to the specific parts that we try to learn. To illustrate, in case of step-size learning, the loss decreases only if the MLP output improves compared to the previous step size $\alpha > 0$. Therefore, its learned MLP output is indeed an improved step size if the GNN has a lower loss.

Obviously, x^* serves as a label for each instance $\mathbf{X}$ and adds a supervised component to $\ell_{\beta,\gamma,\delta}$. However, at least for learning the step size, x^* is not the desired output of the MLPs. Moreover, it cannot be expected that the final output of the GNN already corresponds to the true solution, as it only performs $K = 10$ algorithm steps. Thus, in contrast to a typical supervised output label, x^* only implicitly affects the desired output.

The loss function $\ell_{\beta,\gamma,\delta}$ is sensitive to the selection of the hyperparameters β, γ and δ. A good choice of these parameters depends on the selected class of problems and can be identified by fine-tuning on the validation set. In particular, the loss should be aligned with the evaluation metric in Section 4.5, such that a low loss leads to a low metric value. Note that we kept the norm of the primal residual in the loss function to measure the consensus after the K iterations. It would also be reasonable to set $\delta = 0$ since the relative localization error of the x_i^K implicitly measures consensus. Additionally, it is inadvisable to set both $\beta = \gamma = 0$ as this would result in a measurement of consensus alone.

Especially for problem classes where it is harder or not possible to compute x^*, we could leave it out of the loss function. This leads to the entirely unsupervised loss

$$\tilde{\ell}_\delta(\theta, \mathbf{X}) = \frac{1}{m_\mathbf{X}} \sum_{i=1}^{m_\mathbf{X}} f(x_i^K) + \frac{\delta}{m_\mathbf{X}} \sum_{i=1}^{m_\mathbf{X}} \|y_i^K\|_2^2 \,, \tag{83}$$

which is universally applicable for all problem classes. The downside of $\tilde{\ell}_\delta$ is that it just measures the average function values and not a relative function error as before. Therefore, it is potentially sensitive to instances with higher fluctuations of the function value. Moreover, the localization error cannot be measured anymore and thus is not considered during training. Hence, it is necessary to set $\delta > 0$ here, otherwise consensus is not evaluated. In Chapter 5, experiments are performed with $\ell_{\beta,\gamma,\delta}$ and $\tilde{\ell}_\delta$. A regularization term of the parameterization $\theta \in \Theta$ can be added to both losses.

In the case of learning the step size, one could also consider a classic supervised loss function. For example, each instance $\mathbf{X}$ could be labeled with the best possible step size $\alpha_{\mathbf{X}} > 0$ for convergence. The loss is then calculated by the squared difference between the step size proposed by the GNN and $\alpha_{\mathbf{X}}$ to learn a relation between instances and their suiting step sizes. Labeling the instances requires expensive preprocessing, such as testing a grid of step sizes for each instance, and the results after K iterations must be evaluated by an improvement measure from the following section. This approach has not been investigated in the thesis, but it presents an interesting direction for future research on step-size tuning.

4.5 Evaluation metric for trained GNN

In this section, we discuss how we can evaluate whether the training of the GNN has led to an improved version of DD-ADMM compared to the baseline algorithm. In the previous section, we have already specified that we consider both a more accurate solution at the same computational cost and a solution of the same accuracy at lower computational cost to be an improvement in terms of convergence speed. The first was the natural choice to time-efficiently evaluate the K iterations of the GNN. While the loss function remains an important tool for the evaluation of the trained GNN, the second type of improvement becomes increasingly relevant, particularly for practical applications of the algorithm.

This is because the desired solution often needs to meet certain error bounds in practice. The latter means that the algorithm cannot be terminated until the desired level of accuracy for the relative errors has been reached. The improvable quantity is then the computational cost. In addition, the algorithm can usually only rely on the stopping criteria for termination since the

true solution is unknown in real applications. Therefore, we propose running the deterministic ADMM-GN after the trained GNN and counting the steps needed to reach the desired bounds to measure a possible improvement in computational costs. Since this involves running the algorithm far beyond K steps, the measure is too expensive to be applied as a loss function in every training step, but it adds an important dimension to measure faster convergence of the trained GNN.

Both improvement measures are expected to be correlated, otherwise our loss function would be useless for improving the algorithm in the second direction. However, one should note that it is not guaranteed that the algorithm reaches the desired error bound faster when starting from a more accurate iterate (according to the loss). We have already seen this in Figure 7, where after $K = 10$ iterations $\alpha = 1$ shows the fastest decrease in the relative function error, but $\alpha = 10^{-1}$ reaches the desired error bound earlier. Thus, we use the following two measures to evaluate the performance of the trained GNN on each instance

(i) value of the loss function on the GNN output $\mathbf{V} = [(x_i^K, y_i^K, \lambda_i^K)]_{i=1}^m$,

(ii) steps needed to reach accuracy bounds or stopping criteria starting from the GNN output.

The size of the certain bounds can depend on personal preference, i.e. if we only care about the size of the function value there is no need for us to measure the distance to the global solution x^*.

It is worth noting that in the case of learning the entire x-update, it can also be a desirable goal to reduce the computation time rather than the number of steps to reach a certain accuracy. This is because a trained x-update might lead to an acceleration of this computationally intensive step. Such an improvement was not considered in Chapter 5.

5 Results of GNN-based ADMM Learning

This chapter explores the two unrolling approaches presented in Section 4.3 and tests them on our synthetic problem instances from Section 4.1. We start by evaluating the behavior of our candidate loss functions given in Section 4.4 and set a specific one to be used in all our experiments. Since tuning a neural network is a highly iterative process, we outline the procedure for identifying an appropriate model in the case of step-size learning. Finally, we present the results of our final models for both approaches and compare them with suitable baseline algorithms. In particular, the performance is evaluated in terms of the loss and the needed steps evaluation metric from Section 4.5. All methods are implemented in JAX (Bradbury et al., 2018) and Jraph (Godwin et al., 2020), JAX's package for GNNs.

5.1 Different loss functions

As a first step of building our GNN model, we must fix the loss function and its hyperparameters. In Section 4.4, we have proposed the two candidates $\ell_{\beta,\gamma,\delta}$ (81) and $\tilde{\ell}_{\delta}$ (83). There, we have also pointed out that the loss should be designed in a manner that positively influences the evaluation metric, which is the value we aim to minimize overall. To be more precise, the parameterization with the lowest loss function value on the data set should be as close as possible to the one with the lowest metric value.

In the case of step-size learning, we are able to compare these two measures and adjust the values of β, γ and δ for the loss $\ell_{\beta,\gamma,\delta}$ accordingly. To find a good combination of hyperparameters, we created a grid $\mathrm{A} \subset \mathbb{R}^{+}$ of possible step sizes and tested the different loss candidates on our training data in a deterministic ADMM-GN. For simplicity, we only changed the step size once to a constant step size $\alpha \in \mathrm{A}$ after the first step with $\alpha = 1$. For each problem instance $\mathbf{X}$ and certain combinations of β, γ and δ, we identified the step size $\tilde{\alpha}_{\mathbf{X},(\beta,\gamma,\delta)} \in \mathrm{A}$ with the lowest loss function value after $K = 10$ steps in this deterministic algorithm. The three hyperparameters were selected from a coarse grid of different powers of 10. We then calculated the number of steps needed with $\tilde{\alpha}_{\mathbf{X},(\beta,\gamma,\delta)}$ to bring both relative errors[2] $\overline{\Delta}_{\mathrm{rel},x}$ and $\overline{\Delta}_{\mathrm{rel},f}$

[2]We used the relative errors as a deciding factor here since they showed slightly fewer

below $\epsilon^{\text{rel}_f} = \epsilon^{\text{rel}_x} = 10^{-3}$. This number was limited to a maximum of 500 steps since it is a common issue of ADMM to converge very slowly on certain instances and we did not want to give these instances a high weight. Summed over all considered instances $\mathbf{X}$ the boundary configuration $\beta = 1$ and $\gamma = \delta = 0$ showed the lowest number of total steps. In particular, this number was lower than with a constant naive step size $\alpha = 1$.

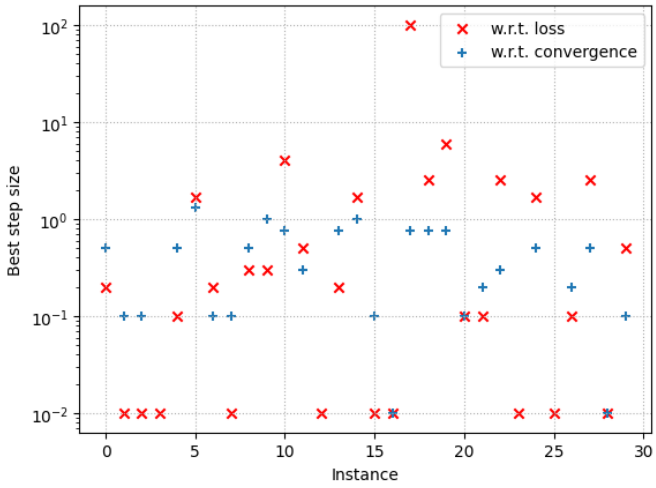

Figure 10: Comparison between the step size $\alpha_{\mathbf{X}}$ that leads to the fastest convergence in both relative errors and the best step size $\tilde{\alpha}_{\mathbf{X},(1,0,0)}$ according to the loss function $\ell_{1,0,0}$ (lowest relative localization error after $K = 10$ algorithm steps) on 30 training instances.

We still observed several deviations between $\tilde{\alpha}_{\mathbf{X}(1,0,0)}$ and the step size $\alpha_{\mathbf{X}} \in \mathrm{A}$ that required the smallest number of steps on each instance to reach the relative error bounds (see Figure 10). However, the number of deviations remained below the number observed for other tested hyperparameter combinations. Since it seemed to be the most promising loss function candidate to lower the number of needed steps, we fixed

$$\ell_{1,0,0}(\theta, \mathbf{X}) = \frac{1}{m_{\mathbf{X}}} \sum_{i=1}^{m_{\mathbf{X}}} \frac{\|x_i^K - x^*\|_2^2}{\|x_i^0 - x^*\|_2^2} \tag{84}$$

outliers compared to the two residuals. However, we have seen combinations of upper (error) bounds, e.g. $\epsilon^{\text{rel}} = 10^{-4}$ and $\epsilon^{\text{pri}} = \epsilon^{\text{dual}} = 10^{-5}$, for which the needed steps to reach these bounds were similar.

as the primary loss function for our GNN model in this chapter, both for learning the step size and the x-update. We expect that this result depends on the specific distribution of problems from which the instances are drawn, as well as on the applied accuracy bounds ϵ^{rel_f} and ϵ^{rel_x}, which do not have to be identical for both relative errors. Moreover, it is expected that the coarseness of our grid for β, γ and δ affected the result. This is because $\ell_{1,0,0}$ showed a tendency to underestimate $\alpha_{\mathbf{X}}$ and $\ell_{0,1,0}$ to overestimate it suggesting that a combined measure of both could be beneficial.

Later, we also ran experiments with the purely unsupervised loss function $\tilde{\ell}_{1/10}$, showing that it can also lead to an improved loss on the validation data. In terms of alignment with our evaluation metric, it is comparable to $\ell_{0,1,1/10}$, with the additional disadvantage that the measured function error is no longer relative. Due to the superiority of $\ell_{1,0,0}$ and the high computational cost of training, we focused on $\ell_{1,0,0}$ and only present results with this loss.

5.2 Different learnable parts

With the loss function now fully defined, we can start the process of finding a suitable GNN model for learning the step size and the x-update step of DD-ADMM. As previously stated, tuning a (Graph) Neural Network is a highly iterative procedure. Therefore, we first sketch this process of model selection in the context of learning the step size. In the subsequent sections, the results of the final model for the two different learnable parts are presented, along with the hyperparameter choices that led to these models.

Due to some initial design choices in the code, all results are limited to a batch size of 1. Moreover, the Adam optimizer was applied in all major experiments because of its promising initial results.

5.2.1 Step size of distributed ADMM

To assess the performance of our step size approach and obtain comparable values, we first define a baseline algorithm, a benchmark, and a lower threshold for the loss function.

The baseline algorithm to which we compare our method is a deterministic ADMM-GN with naive step size $\alpha = 1$, which is run for the usual $K = 10$

steps and performs well on average. In addition, we calculated the constant step size $\bar{\alpha} = 7.54 \in \mathrm{A}$, which gives the lowest average loss over all training instances when used after the second step[3]. The value of our selected loss $\ell_{1,0,0}$ with $\bar{\alpha}$ serves as a benchmark that we expect to reach. Moreover, as a lower threshold for a near-optimal model, we calculated, analogously to Section 5.1, the value of $\ell_{1,0,0}$ with the best possible $\tilde{\alpha}_{\mathbf{X},(1,0,0)} \in \mathrm{A}$ regarding this loss on all problem instances $\mathbf{X}$. A double change in the step size, which does result in a slightly lower loss, was not considered here, as we did not expect to reach this lower bound in a simple model, at least not on the validation and test data.

Process of model selection

To find a suitable model and a good configuration of the remaining hyperparameters, we followed the following informal heuristic to tackle the bias-variance trade-off.

1. Select a sufficient size and complexity of the model (number of parameters and hidden layers) such that it is capable of reaching a desired accuracy bound (on the training data).
2. Utilize tools to reduce the generalization gap between the loss on training and validation data.
3. Evaluate the model, reconsider other hyperparameter choices, and restart the process from 1.

The first step, which aims to find a model with an acceptable bias, deals with the architecture of our GNN. In Section 4.3.1, we have already specified the majority of its design and computational structure. Remember, in particular, that we decided to include two MLPs of the same architecture into the GNN that determine new step sizes (see Figure 8). The remaining open choices that affect the model complexity are the number of hidden layers in both MLPs and the number of incorporated neurons in these layers. These two choices should be made so that the model can achieve a certain target accuracy.

[3]In fact, we allowed for a second adjustment of the step size, analogous to our GNN, after the sixth step. However, keeping $\bar{\alpha} \in \mathrm{A}$ constant proved to be optimal here.

In order to gain a first impression about the learning capabilities of different model complexities, we ran tests on small training data sets containing only 90 instances due to the extensive training time and the limited computing resources. On average, a simple model with a single hidden layer and $4n$ neurons in this layer already exceeded both our benchmark and our lower threshold on the loss function on the training data (see Figure 11).

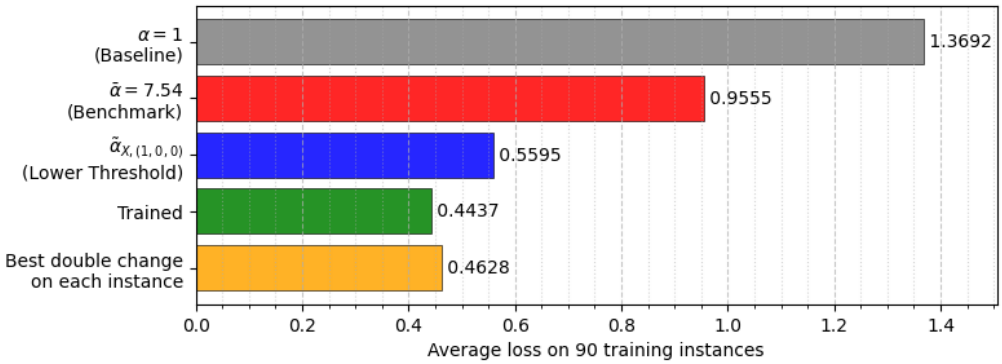

Figure 11: Average loss for different step sizes on the 90 instances used for training. Both applied MLPs have the architecture $[7n + 2, 4n, 1]$.

It even reached the best loss value when allowing a double change in the step size[4]. Of course, we observe heavy overfitting here since the number of parameters in this model is

$$2\big(((7n + 2)4n + 4n) + (4n + 1)\big) = 56n^2 + 32n + 2 \tag{85}$$

and thus larger than the number of used training instances. However, the result confirms that the overall structure of the GNN together with the loss function works as desired since the parameterization is adapted towards the optimal step sizes. We also tested whether it makes more sense to use a deeper architecture for the MLPs, i.e. increase the number of hidden layers, or a wider architecture that has more neurons in the existing layer. For this purpose, we compared models with similar numbers of parameters (for $n = 2$), e.g. an architecture $[7n + 2, 8n, 1]$ against $[7n + 2, 5n, 5n, 1]$ with an additional hidden layer. These experiments showed a better performance of

[4]In the diagram, the average loss with a trained step size is slightly lower than the value with a double change in the step size due to the applied discrete grid A.

wider MLPs with only one hidden layer, so we only considered such architectures for our final model. Figure 12 shows how the training and validation loss behaves when increasing the number of neurons in the hidden layer. As expected, the model starts to overfit with an increasing number of neurons, and thus parameters, leading to a lower training loss and a larger generalization gap.

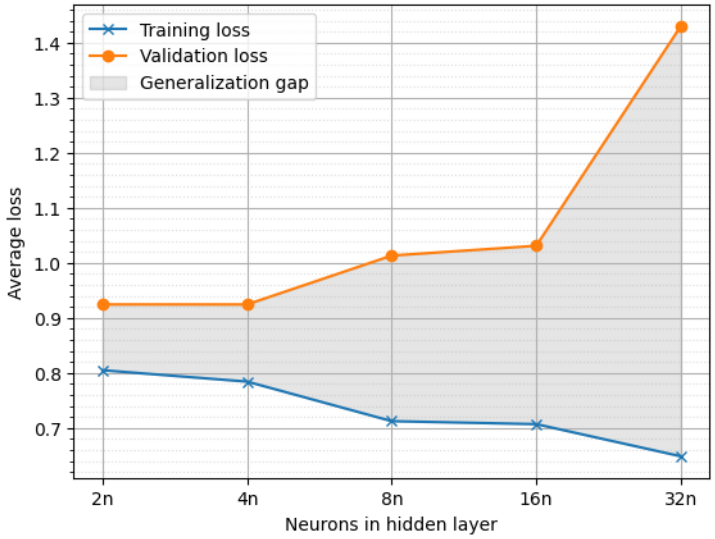

Figure 12: Behaviour of training and validation loss for an increasing number of neurons in the hidden layer (900 training instances).

To prevent overfitting and to reduce the generalization gap, the second step of our heuristic, we essentially have two tools: increasing the number of training instances and including regularization into the loss function. Figure 13 shows the effect of using more instances for training our model. It also aligns with the theory that increasing the number of training instances leads to a decrease in the generalization gap while slightly increasing the training loss. In addition, we used the regularized loss function

$$\ell^{\text{reg}}_{1,0,0}(\theta, \mathbf{X}) = \ell_{1,0,0}(\theta, \mathbf{X}) + \lambda^{\text{reg}} \|\theta\|^2_\star \tag{86}$$

in our model to further improve generalization. After various experiments on small training data sets, the L^2-norm seems to be more promising for regularization in our setting. A good choice for the regularization parameter

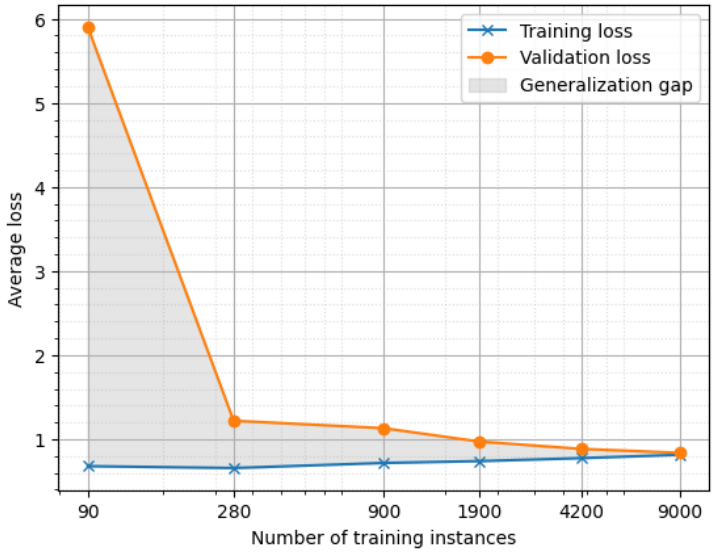

Figure 13: Behaviour of training and validation loss for an increasing number of used training instances ($16n$ neurons in the hidden layer).

$\lambda^{\text{reg}} \geq 0$ must be fine-tuned with the final model but appears to be between 10^{-3} and 10^{-5}.

In the third step, we evaluate and compare the performance of a chosen model on training and validation data to further improve the design and adapt it in the more pressing direction. For example, if we observe a low training but high validation loss we should consider adding more training data or increase λ^{reg}. In case these possibilities are exhausted, it may be beneficial to simplify the model complexity in order to prevent overfitting. This procedure is applied iteratively until an appropriate model is found.

In the evaluation step, it is also important to ensure that the training process actually converges to a stable model. To verify this, we increase the number of epochs and observe if the performance of the model stabilizes. In addition, the same model should be trained multiple times starting from different initializations to assess whether the observed performance is reliable. If a model fails to converge, certain hyperparameters should be reconsidered, in particular, the learning rate and the number of epochs.

Before fixing the $7n + 2$ input features in Section 4.3.1, we also ran initial experiments on what information about f_i to include. For example, we considered using $f_i(x_i^k) \in \mathbb{R}$ directly as an input. However, our experiments have shown that additionally including $f_i(x_i^k)$ led to no notable improvement and excluding e.g. Bx_i^k led to significantly worse results. Also, the use of the degree d_i seemed to be valuable, motivating our choice of input features.

In general, it can also be valuable to gain insights into why certain instances of the specific problem distribution show an unusually large loss. We identified two major reasons for this:

(i) Some instances contain a matrix B such that an eigenvalue of B^TB is numerically very small (although it cannot be 0 with probability 1).

(ii) Some instances have a unique solution x^* that is very close to the origin.

The first issue, which can also cause B^TB to be ill-conditioned, makes it harder for any optimization algorithm to converge to the unique solution x^* since f has only marginal differences along the span of the respective eigenvector. This can lead to a situation where the algorithm has almost converged to the optimal function value but is far away from the actual solution x^*. The second point can also lead to an extraordinarily high localization error $\overline{\Delta}_{\text{rel},x}$ since we initialize the algorithm at $x^0 = 0$ and thus divide by the norm of x^*. As $\ell_{1,0,0}$ only measures $\overline{\Delta}_{\text{rel},x}$ it can be sensitive to minor changes on such instances and potentially influence the model notably. We still decided to keep these instances in our datasets but want to raise awareness for their potential influence.

During the tuning process, a model was trained that achieved an average validation loss of 0.6801, which was significantly better than the performance of other models. However, all attempts to reproduce this result, even with identical hyperparameters, were unsuccessful since no fixed random seed was yet applied for the reshuffling of the instances in each epoch. Especially, after reducing the learning rate to stabilize the results, the loss values obtained in later models were higher in general. This suggests that the superior performance of the model was likely due to a fortunate and serendipitous combination of factors. Thus, it is not included in the final evaluation, but its existence is noteworthy as it indicates potential for further improvements.

Performance of the final model

By following this procedure and after several adjustments and experiments, we arrived at the following hyperparameter choices that led to the most promising performances on the validation data:

- architecture of MLPs: $[7n + 2, 16n, 1]$ (one hidden layer with $16n$ neurons),
- size of training data: 9,000 (maximal considered amount),
- regularization: L^2-regularization with $\lambda^{\text{reg}} = 10^{-4}$,
- learning rate: 10^{-5},
- epoch: maximal 5,000 with early stopping (check validation loss every 50 epochs and stop if not improved after 5 consecutive checks).

Moreover, we utilized the Xavier (or Glorot) initialization, which is beneficial for MLPs that use the sigmoid activation function (Goodfellow et al., 2016). Figure 14 shows the range of loss values obtained by this setting on 5 different random seeds and compares it to the case where no regularization is used. We observe that including regularization, if at all, only marginally improves the model in this setting. This finding differs from older experiments which is why we still kept it in the final model configuration. Reasons for the lower influence of regularization are discussed in Chapter 6. However, we still observe a small effect of regularization as it shows a tendency to have a slightly higher training loss but an equal or even marginally lower validation and test loss.

Both models perform consistently better than our benchmark, which uses $\bar{\alpha}$ on all instances, and thus also better than the baseline algorithm, as shown in Figure 15. Our lower threshold, which selects $\tilde{\alpha}_{\mathbf{X},(1,0,0)}$ on each instance, is clearly not reached.

The variation between different outcomes of the trained models remains considerable. The randomness within the same dataset lies in the initialization of the parameters and the order in which the instances are reshuffled in each epoch. Also noteworthy is that the performance of a trained model varies between the different datasets. For example, in case of no regularization, the model trained on seed 5 shows the second lowest loss on the validation data

Figure 14: Comparison between the average loss values of the final model with and without regularization on the training, validation, and test data set. Each model is trained on 5 different random seeds.

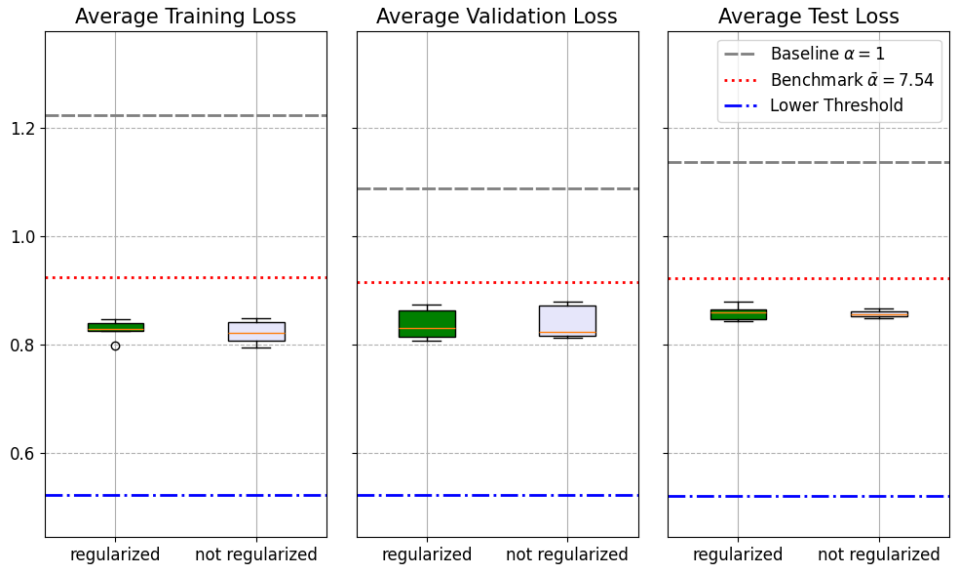

Figure 15: Same comparison, now including the baseline and lower threshold.

but has the highest loss of all 5 seeds on the test data. This suggests that the problem instances in the different datasets vary notably despite being drawn from the same distribution of problems.

Based on the regularized model with the median performance in terms of validation loss, we calculated the needed steps metric from Section 4.5 on the test instances. In this experiment, we applied the deterministic ADMM-GN on the GNN output, once with the second step size $\alpha^{(2)}$ chosen by the GNN on each instance and also with the naive step size $\alpha = 1$. To assess whether we can achieve faster convergence with the trained model, we compared both results with our three deterministic comparison algorithms. Again we capped the number at a maximum of 500 steps on each instance.

The results are shown in Figure 16. We observe that the trained algorithm performs better when it switches back to $\alpha = 1$ after the unrolling steps. This version is slightly faster than our baseline algorithm. Also, it outperforms the benchmark algorithm with $\bar{\alpha} = 7.54$, which is notably inferior in this metric relative to the others. Our lower threshold is again not reached.

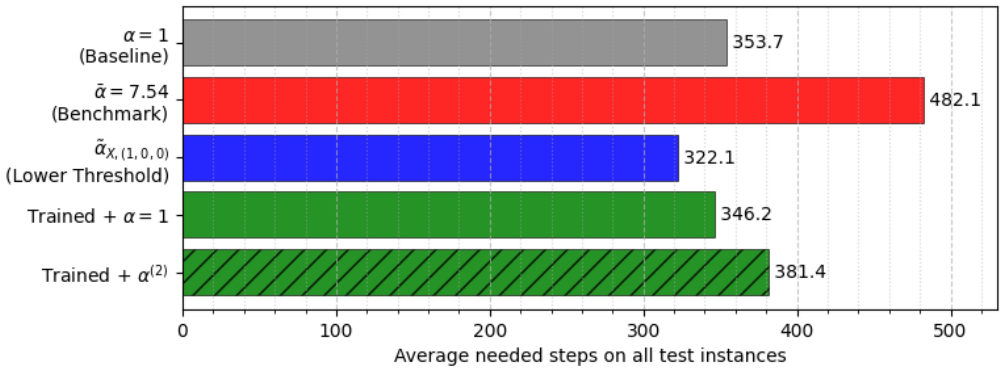

Figure 16: Comparison of average needed steps to reach accuracy bound. The trained model uses (1) the learned $\alpha^{(2)}$ and (2) $\alpha = 1$ after the $K = 10$ unrolling steps.

5.2.2 Update step for local solution candidate x_i

Due to the time limitation of the thesis, the same hyperparameter configuration of the final model for step-size learning was also applied for learning

the x-update. Note that, despite using the same hyperparameters, the architecture of the included MLPs is slightly different. As specified in Section 4.3.2, the now constant step size $\alpha = 1$ is no longer considered as an input argument. Moreover, the output layer consists of n neurons since the output represents the updated local variable $x_i^{k+1} \in \mathbb{R}^n$. Therefore, the here applied MLPs have the architecture $[7n + 1, 16n, n]$. For $n = 2$, this leads to only one additional parameter per MLP in this approach, resulting in two more parameters in total.

By design, all but 2 of the unrolling steps are now performed using the deterministic ADMM-GN with naive step $\alpha = 1$. Additionally, the MLPs receive no guidance on how to update x correctly. Thus, we initially considered the deterministic ADMM-GN, which uses $\alpha = 1$ and runs for only 8 steps, as a fair baseline algorithm since the MLPs should be able to learn the identity function on the x-iterate as a possible update step. However, given the results, we kept the same comparison measures from the previous section.

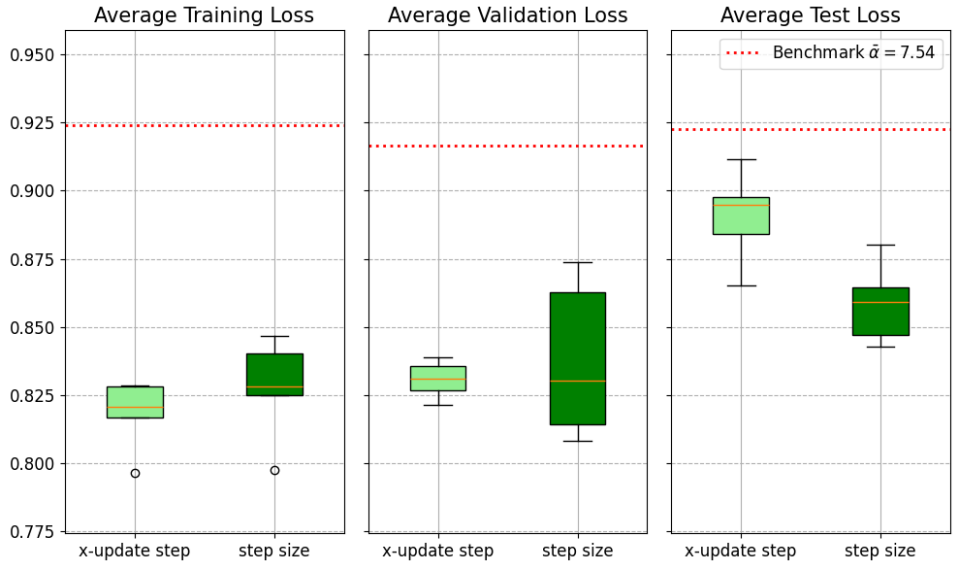

Figure 17: Comparison between the average loss values of the final model for the x-update and the step-size learning approach on the training, validation, and test data set. Each model is again trained on 5 different random seeds.

Figure 17 shows the average loss values obtained by the trained models and compares them to the performance of the step size approach. Clearly, both

the baseline algorithm and the loss benchmark with $\bar{\alpha}$ are still beaten on all data sets. Compared to the step size models, it shows a similar performance. The range of losses obtained for the different random seeds is even more concentrated. However, the performance on the test set is worse and shows a notable drop compared to the training and validation loss.

Analogous to the previous section, we selected the model with the median performance on the validation data for the needed steps metric. Following the unrolling steps, we directly applied the deterministic ADMM-GN with $\alpha = 1$ to maintain the convergence property of the algorithm. For consistency, we switched to the naive step size after $K = 10$ steps in all algorithms. Thus, we investigate in this experiment which algorithm provides the best initial point for DD-ADMM with $\alpha = 1$ and possibly also for other constant step sizes.

Figure 18 presents the obtained results. In comparison to the trained step size, the trained x-update only needs around 3 more steps on average despite not being fine-tuned. It remains slightly faster than the naive baseline and the benchmark algorithm. Compared to Figure 16, we observe that both the benchmark and the lower threshold algorithm improve significantly when switching back to $\alpha = 1$ after the first $K = 10$ steps. This is consistent with the behavior we observed for the trained step size, where it was also superior to use $\alpha = 1$ than to continue with $\alpha^{(2)}$.

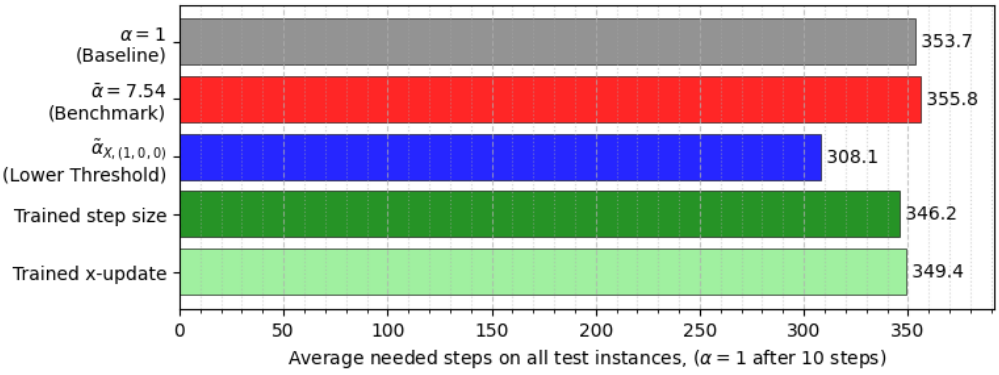

Figure 18: Comparison of average needed steps to reach accuracy bound. Every algorithm uses $\alpha = 1$ after the first $K = 10$ steps.

6 Discussion

This chapter discusses the results obtained in Chapter 5 and their limitations. In addition, we highlight possible improvements and future research directions since the thesis only represents an initial exploration of the topic.

6.1 Evaluation of the results and limitations

In this section, we briefly discuss the results from Chapter 5 and connect them to our two open research questions. Moreover, we point out unexpected results and potential reasons for their occurrence. Where possible, we discuss directly how to address the issues encountered in future experiments.

Performance of the step size approach

In the beginning of our experiments, we have observed in Figure 11 that the proposed unrolling approach for the step size in combination with the loss function works as desired and leads to optimal step sizes regarding the loss on the training data. However, this was not the central goal of our training procedure as we want to investigate whether the approach is able to accelerate DD-ADMM on unseen problem instances from the same distribution. This central research question was successfully answered by the results of the final model for step-size learning, which demonstrated a generalization ability to unseen instances. Both the naive baseline and the benchmark algorithm were beaten in terms of the loss (Figure 15) and the needed steps (Figure 16).

But the results also raised new questions. First of all, we observed considerable fluctuations in the obtained results for different random seeds. Possible explanations are a high sensitivity of the loss function, in particular to the two types of instances discussed in Section 5.2.1, variation between the datasets despite being drawn from the same distribution, poor and improvable hyperparameter choices for the final model and a generally hard optimization landscape for learning the step size. Clearly, this needs to be investigated in further experiments and addressed to achieve a more stable performance.

Moreover, we observed in Figure 14 that the effect of regularization was almost not visible anymore, in contrast to previous experiments where it

led to improved generalization and a lower validation loss. We believe that this effect is mainly caused by the addition of early stopping in the final configuration, which is also a regularization technique to avoid overfitting and therefore leads to a vanishing effect of L^2-regularization.

Alignment between loss and needed steps

What is most striking about the results is the fact that the trained step size model only needs marginally fewer steps than the baseline algorithm to reach the accuracy bounds, despite being significantly superior in terms of the loss. It could be the case that capping the steps at a maximum of 500 steps shifts the results to this outcome. However, we believe that the difference in the two metrics arises mainly because they do not align well enough. This could be improved by a more comprehensive fine-tuning of the parameters β, γ, δ associated with the loss function, when they are chosen from finer grids than just different powers of 10 as in Section 5.1. As mentioned in this section, the here applied loss $\ell_{1,0,0}$ tends to underestimate the step size $\alpha_{\mathbf{X}}$ that leads to the fastest convergence. Since $\ell_{0,1,0}$ overestimates $\alpha_{\mathbf{X}}$, this suggests setting $\gamma > 0$ to also include the relative function error $\overline{\Delta}_{\mathrm{rel},f}$ for a better alignment and faster convergence.

The argument of insufficient alignment is supported by the poor performance of the benchmark algorithm in the needed step metric. Although the benchmark uses the step size $\bar{\alpha}$ that gives the lowest average loss on the training data, it is significantly worse than the baseline algorithm in terms of needed steps. In particular, this is displayed in Figure 18. Compared to the baseline algorithm, the benchmark clearly achieves the lower loss, measured by definition after the $K = 10$ unrolling steps. Thus, it indicates that the computed iterates are closer to the true solutions on average. In said Figure 18, we apply the same deterministic algorithm to both sets of iterates and still observe a faster convergence of the baseline algorithm, despite starting from the seemingly inferior iterates. Therefore, our applied loss function $\ell_{1,0,0}$ does not necessarily lead to iterates from which we achieve faster convergence, indicating that the relative localization error $\overline{\Delta}_{\mathrm{rel},x} = \ell_{1,0,0}$ alone is not sufficient to decide whether a set of iterates is a superior starting point for DD-ADMM. In addition, it is reasonable to include $\overline{\Delta}_{\mathrm{rel},f}$ in the loss, at least to some extent, since we require both relative errors to converge in our

evaluation metric. However, in light of the experiments from Section 5.1 and the slight but noticeable improvement of the trained algorithm, $\ell_{1,0,0}$ is also not entirely unsuitable.

Applying the learned step size after unrolling

Closely related, we can answer the research question of whether the learned step sizes generalize beyond the unrolling steps so that $\alpha^{(2)}$ can be advantageously applied in further deterministic steps. We observe that it is significantly better to switch back to the naive step size $\alpha = 1$ than to continue with $\alpha^{(2)}$, as visualized in Figure 16. This observation is not completely unexpected since we have already seen the main reason for this behavior. Figure 7 showed us that the step size leading to the lowest relative error after $K = 10$ steps is not necessarily the step size leading to the fastest convergence. Thus, we cannot expect $\alpha^{(2)}$, which has been optimized for steps 6 to 10, to perform optimally on subsequent algorithm steps. To improve this generalization behavior and get a better estimate for a well-suited step size beyond unrolling, we could increase the number of unrolling steps. This should automatically also improve the alignment between the loss and the evaluation metric as the relative errors are observed over a larger number of steps.

Obviously, this adjustment would cause a more expensive training process leading to a natural tradeoff between a potentially higher accuracy and the expected training time. However, note that only the time for training is affected since once trained every step still requires the same computational resources. Consequently, we recommend investigating a higher number of unrolling steps if the increased training time is acceptable.

Performance of the x-update and comparison to step-size learning

Additionally, we briefly investigated the capabilities of the x-update approach. Figure 17 and Figure 18 demonstrate that it also successfully generalizes from training problems to validation and test instances. Again, the baseline and the benchmark algorithm are outperformed in terms of both loss and needed steps. Given the fact that the same hyperparameters as for the step size were used without further fine-tuning, the good performance of the x-update approach offers even greater potential for further enhancements.

Note again that updating x is a non-trivial step in DD-ADMM, as it includes solving an optimization problem itself. Two of these steps are now performed freely by the MLP without any restriction or guidance and the naive $\alpha = 1$ is applied in all steps of this approach. Thus, the obtained performance is even more impressive since the GNN seems to learn a meaningful update rule for x and affects only 2 algorithm steps directly, unlike the step size approach where the two learned step sizes are used over 9 algorithm steps.

In comparison, we observed a very similar performance of both approaches, with a slight advantage of the step-size learning approach on the test data. Figure 17 shows a slight jump in the test loss for both approaches, which is not uncommon due the fine-tuning of hyperparameters on the validation set. In particular, the use of an early stopping mechanism that tracks the validation loss leads to a small overfitting to the validation data, explaining this observation. The jump is higher in the x-update model, which might indicate a stronger sensitivity to outlier instances.

6.2 Future work

In this section, we discuss possible future work, including further potential improvements of the obtained results besides those already mentioned, broader experiments, and deeper research directions related to the topic. These considerations are not meant to be exhaustive, but to give an overview of other research ideas we came across during the thesis.

6.2.1 Further possible improvements of the results

Clearly, the presented results are just an initial exploration of the performance achievable with the proposed unrolling approaches, given the limited computational resources and time. In light of the only marginal gain in needed steps, further improvements in the results are also required to justify real-world implementation.

In both approaches, the simple architecture of the incorporated MLPs and the use of only a few techniques to guide and stabilize the training process provide opportunities for such improvements. For example, we believe that batching and learning rate scheduling can stabilize the results. The pos-

sibility for further enhancements is also indicated by the existence of the serendipitous model, which demonstrates that good parameterizations for learning the step size exist. It remains open and should be further investigated if we can use more advanced techniques and find better hyperparameter choices such that similar performing models are obtained regularly.

Beyond that, it should be considered to include more advanced input features for the MLPs, giving them more arguments to capture possibly relevant differences between the problem instances. This could be features that encode the connectivity of the underlying graph, as used by Häusner et al. (2024) and mentioned at the end of Section 4.3.1, or ADMM and problem-related features.

Besides more exhaustive fine-tuning of the hyperparameters β, γ, and δ associated with the loss function, a completely different loss could be evaluated and compared to $\ell_{\beta,\gamma,\delta}$. For example, in Section 4.4 we proposed using a supervised loss function for step-size learning that maps each instance $\mathbf{X}$ to $\alpha_{\mathbf{X}}$.

Finally, also the unrolling approach itself could be refined by e.g. including more MLPs or unrolling more steps as already suggested. Also, weight sharing across the layers could be a possibility for learning the x-update. Moreover, the approaches for learning the step size and the x-update could be combined, as described in Section 4.3.2. Learning could be even extended to other parts of the algorithm. In particular, we could focus on learning the communication matrix A, which is the other hyperparameter of DD-ADMM. For simplicity, we applied the same $A = \mathcal{L} \otimes I_n$ in all algorithms. However, the experiments from Makhdoumi and Ozdaglar (2017) and the convergence proof indicated that A also has a strong influence on the convergence speed, leaving room for further improvements compared to the baseline algorithm.

6.2.2 Extending the experimental scope

In addition to investigating further improvements of the results, the scope of the experiments should be extended. This includes broadening the distribution of possible graphs G and using more challenging local functions f_i. For example, we could start by relaxing the strict convexity of f_i. Later, it would be interesting to train the GNN even on non-convex functions and observe its behavior, given that DD-ADMM is not guaranteed to converge on such f_i.

For the applicability of our unrolling approach, it is also crucial to test it on real-world problems that contain non-artificial f_i and actual agent networks represented by G.

Related to this, it would be interesting to examine the case where we have no access to the true solution x^* and are forced to follow an unsupervised learning approach. We have mentioned already in Section 5.1 that utilizing our suggested loss $\tilde{\ell}_\delta$ for this case also leads to a lower average loss on the validation data and thus to a lower global objective on average. However, we should study whether $\tilde{\ell}_\delta$ can be tuned such that it also leads to faster convergence.

Further, the unrolling approach should be applied on other algorithms for distributed optimization to compare whether different algorithms allow for stronger gains in convergence speed and explore if certain algorithm components have stronger potential for improvements. For this to work, it must only be possible to bring the algorithm into the computational form of the GN framework, as we have done with DD-ADMM in Algorithm 3 (visualized in Figure 6). In particular, it is essential that the iteration rule can be transformed in such a way that messages from neighboring nodes are aggregated into a single incoming feature by an aggregation function. Note that in general, global features can also be incorporated in a GN block.

6.2.3 Further research directions

The topic also allows for deeper research in the areas of Learning-to-Optimize with GNNs and distributed optimization.

First, it would be interesting to study the explainability of the GNN models. In particular, one could explore what is captured by the GNN, which input features are particularly relevant for the prediction, and what features are created to decide on a step size or the x-update. This could lead to the development of additional advanced input features for the included MLPs or to possible advances in fine-tuning distributed optimization algorithms.

Moreover, a probabilistic approach could be taken to analyze the influence of specific problem distributions on the unrolling results. More precisely, the optimal solutions x^* and possibly the optimal step sizes $\alpha_{\mathbf{X}}$ are expected to follow a certain distribution, influenced by the problem distribution itself. We

could investigate if and how the distribution of optimal solutions affects the unrolling performance. This analysis might help identify scenarios where the approach underperforms and guide targeted research to adjust the network for better performance in those cases. The latter can significantly enhance the approach's applicability and overall performance, or it could reveal that the current results are positively affected by favorable conditions within our synthetic distribution.

Regarding the applicability, it is also important to study the robustness of the model when we apply it to out-of-distribution problems, which differ significantly from the problems on which the GNN was trained.

Finally, we can focus on the limitations that GNNs have themselves, despite the earlier discussed algorithm alignment. For example, simple GNN architectures, such as the one we applied, have issues distinguishing certain non-isomorphic graphs and capturing global information like shortest paths between nodes. In addition, certain generalization bounds exist that might influence the outcome (Cappart et al., 2023). We must study these limitations more thoroughly and investigate whether they offer potential for further enhancements in the approach through even more advanced networks.

7 Conclusion

In our efforts to improve the convergence speed of distributed optimization on specific problem distributions, the thesis introduced and investigated two novel L2O approaches that use Graph Neural Networks. This chapter summarizes the key considerations that guided the development of these approaches and highlights the main contributions of the thesis. In particular, we revisit and address the central research questions.

Since we wanted to guide the GNN with an already existing and convergent algorithm instead of letting it freely discover how to solve distributed optimization, we followed a model-based L2O approach in the thesis. In Chapter 2, we therefore began to derive a specific version of ADMM that met our needs. In particular, the final DD-ADMM algorithm from Makhdoumi and Ozdaglar (2017) allows all agents to compute their iterates in a distributed manner, relying only on decentralized communication over the given edges. We concluded the chapter by proofing that the ergodic average of the locally computed solutions converges to an optimal solution with a sublinear convergence rate.

In the subsequent chapter, we stepped away from distributed optimization and introduced Graph Neural Networks following the framework of Battaglia et al. (2018). Especially, we focussed on the computational structure of its two components: (i) the deterministic Graph Network which defines the overall update structure and (ii) the trainable Multi-Layer Perceptron which adds learning capabilities to the update functions of the GNN.

The central reason why we selected GNNs as our trainable network is the algorithmic alignment between DD-ADMM and the computational structure of GNNs. In Chapter 4, we showed that they are not only similar. In fact, we defined a deterministic GN that performs a single iterations of the algorithm. Crucial to this result was the fact that DD-ADMM's update steps could be rewritten to receive only a single aggregated message feature instead of all separate incoming messages. This allowed the use of an aggregation function.

The finding provided a way to address the main research question of how GNNs can be used to learn distributed optimization. The central idea is to unroll K steps of the algorithm on which the GNN is trained. Hereby, the included MLPs influence certain algorithm components and thus the

outcome after the K steps. In our case, the GNN becomes the algorithm itself, since it is constructed from K consecutive ADMM-GNs. We proposed two simple GNN architectures for learning pivotal parts of the algorithm: (a) the step size α and (b) the x-update step. The main difference is that step-size learning only tunes one parameter of DD-ADMM and thus remains convergent, whereas the x-update is performed freely without any restriction. Here, convergence is regained by switching back to the original algorithm and the outcome after unrolling could be considered as an improved initialization.

The direction of learning is determined by a loss function, which led to our second constructional research question: How can improvements be measured in terms of distributed optimization? Our loss candidate $\ell_{\beta,\gamma,\delta}$ portrays an initial answer to this, as it time-efficiently measures the average accuracy of the local iterates after the unrolling steps. However, the ultimate goal of training is to achieve faster convergence. Since each iteration requires the same computational resources, we proposed to also count the steps needed by DD-ADMM until convergence to evaluate a trained model. In contrast to the loss, this is computationally expensive and thus cannot be used during training.

Clearly, the development of our two unrolling approaches, combined with the definition of a suitable loss function, represents the main contribution of this thesis. The idea offers potential for further refinements and could even be transferred to other graph algorithms involving local computations if the exchanged messages can be aggregated by an aggregation function.

In Chapter 5, we then explored the capabilities of the approaches through experiments on synthetic problem instances. Both approaches beat the baseline and the benchmark algorithm in terms of loss and needed steps on the test instances. Therefore, the crucial research question - whether we can train a GNN to learn something about DD-ADMM that generalizes to unseen problems from the same distribution - is clearly answered in the affirmative. Although the enhancements in needed steps were minor, the simplicity of the applied models, the non-optimal alignment between loss and evaluation metric, and a possible increase of the number of unrolling steps leave room for significant improvements of the results, justifying the effort to implement the approaches in a distributed setting.

As discussed in Chapter 6, the approaches must be investigated further, for example on real-world problems to ensure their applicability. Future research

directions include studying the explainability and robustness of the models.

Overall, the results demonstrated significant potential for speed-up and automatic fine-tuning of DD-ADMM when similar problems need to be solved repeatedly. Although the training process can be computationally intensive, once trained, the algorithm outperforms the baseline and achieves faster convergence. Given the aforementioned demand for fast and efficient distributed optimization algorithms, the proposed unrolling technique holds substantial value for the industries in need, serving as a foundation for future developments and advances in the field.

A Appendix

A.1 Algorithms

Algorithm 4 Centralized, distributed ADMM

Initialize $z^0 \in \mathbb{R}^n$, $x_i^0 \in \mathbb{R}^n, \lambda_i^0 = 0 \in \mathbb{R}^n$ for $i = 1, \ldots, m$, and $k = 0$
while stopping criterion is not reached **do**
  for $i = 1, \ldots, m$ **in parallel do**
    Update local x-iterates:

$$x_i^{k+1} = \arg\min_{x_i} \left(f_i(x_i) + (\lambda_i^k)^T (x_i - z^k) + \frac{\alpha}{2} \|x_i - z^k\|_2^2 \right)$$

    Communicate $x_i^{k+1} + \lambda_i^k / \alpha$ to central node
  end for
  Update central z-iterate:

$$z^{k+1} = \frac{1}{m} \sum_{i=1}^{m} x_i^{k+1} + \frac{1}{\alpha} \lambda_i^k$$

  Communicate z^{k+1} to all nodes
  for $i = 1, \ldots, m$ **in parallel do**
    Update local λ-iterates:

$$\lambda_i^{k+1} = \lambda_i^k + \alpha (x_i^{k+1} - z^{k+1})$$

  end for
  $k \leftarrow k + 1$
end while
return $x^k = [\, x_i^k \,]_{i=1}^m$

A.2 Proofs

Proof of Lemma 2.1. We start by proving part (a) of the Lemma, i.e. the equivalence between the system $x_i = x_j$ from (2) to $Ax = 0$.

'$\Leftarrow$': Let $Ax = 0$. For $1 \leq \ell \leq n$ consider the vector $x^{(\ell)} := (x_{1,\ell}, \ldots, x_{m,\ell})$ consisting of the ℓ-th coordinate of all x_i. Because of the definition of $A := \mathcal{L} \otimes I_n$, $Ax = 0$ implies $\mathcal{L}x^{(\ell)} = 0$. Since G is connected $\ker(\mathcal{L}) = \text{span}(\mathbf{1})$ and since $x^{(\ell)} \in \ker(\mathcal{L})$ it follows that $x_i^{(\ell)} = x_j^{(\ell)}$ for any $i, j \in \{1 \ldots, m\}$.
'$\Rightarrow$': Let $x_i = x_j$ for any $i, j \in \{1 \ldots, m\}$. In particular for $1 \leq \ell \leq n$ it is $x_i^{(\ell)} = x_j^{(\ell)}$. Consequently, $\mathcal{L}x^{(\ell)} = 0$ for every ℓ implying that $Ax = 0$.

Part (b) will be proven by first showing the equivalence between $Ax = 0$ and $A^T D^{-1} Ax = 0$.

'$\Rightarrow$': Let $Ax = 0$. Trivially, $x \in \ker(A^T D^{-1} A)$.
'$\Leftarrow$': Let $A^T D^{-1} Ax = 0$. Since the inverse of a diagonal matrix is still of diagonal shape, left multiplying with x^T gives

$$\begin{aligned} 0 = x^T \left(A^T D^{-1} A\right) x &= \left(x^T A^T D^{-1/2}\right) \left(D^{1/2} Ax\right) \\ &= \left(D^{1/2} Ax\right)^T \left(D^{1/2} Ax\right) = \|D^{1/2} Ax\|_2^2 \,, \end{aligned} \tag{87}$$

implying that $D^{1/2} Ax = 0$. Since D (and therefore also $D^{1/2}$) is invertible, it follows that $x \in \ker(A)$.

Now lets consider the singular value decomposition of $A^T D^{-1} A = U \Sigma U^T$ and the definition of $Q := U \Sigma^{1/2} U^T$. Note that both use the same basis transformation U. Since $\sigma = 0$ if and only if $\sqrt{\sigma} = 0$ for any singular values $\sigma \in \mathbb{R}^+$ on the diagonal of Σ, $x \in \ker(U \Sigma U^T) = \ker(A^T D^{-1} A)$ is equivalent to $x \in \ker(Q)$. Applying part (a) concludes the proof. □

Proof of Lemma 2.4. As stated before in section 2.2.2, the y-update step can be written compactly as $y^k = D^{-1} Ax^k$, which gives

$$\sum_{j \in N(i)} A_{ji} y_j^k = \sum_{j \in N(i)} A_{ji} \left[D^{-1} Ax^k\right]_j = \left[A^T D^{-1} Ax^k\right]_i \,, \tag{88}$$

since $A_{ji} = 0$ if $j \neq N(i)$. Note as well that $A_{ji}^T = A_{ji}$ since it is diagonal per definition (here for $n = 1$ it is even $A_{ji} \in \mathbb{R}$). We can also rewrite the

update step for λ^k, $k \geq 0$

$$\lambda^k = \underbrace{\lambda^0}_{=0} + \alpha \sum_{s=1}^{k} y^s = \alpha \sum_{s=1}^{k} D^{-1} A x^s \,. \tag{89}$$

Similarly to before, we can use this relation to obtain

$$\sum_{j \in N(i)} A_{ji} \lambda_j^k = \sum_{j \in N(i)} A_{ji}\, \alpha \sum_{s=1}^{k} [D^{-1} A x^s]_j = \alpha \sum_{s=1}^{k} [A^T D^{-1} A x^s]_i \,. \tag{90}$$

Note that, as in (88), the right hand side depends on the full vector x^k and not just on x_i^k. For each agent i, the under Assumption 1 (i) and (ii) solvable update step for x_i^{k+1} can be written by the optimality condition

$$h_i(x_i^{k+1}) + \sum_{j \in N(i)} A_{ji}^T \lambda_j^k + \alpha \sum_{j \in N(i)} A_{ji}^T \left(y_j^k + A_{ji}(x_i^{k+1} - x_i^k) \right) = 0 \,, \tag{91}$$

for some $h_i(x_i^{k+1}) \in \partial f_i(x_i^{k+1})$. Incorporating (88) and (90) gives

$$h_i(x_i^{k+1}) + \alpha \left(\sum_{s=1}^{k} [A^T D^{-1} A x^s]_i + [A^T D^{-1} A x^k]_i + \underbrace{\sum_{j \in N(i)} A_{ji}^T A_{ji}}_{= M_{ii}} (x_i^{k+1} - x_i^k) \right) = 0 \,. \tag{92}$$

Note that M is invertible since $d_{\min} \geq 1$, which holds in particular since G is assumed to be connected. Therefore, (92) is equivalent to

$$x_i^{k+1} = \left(x_i^k - M_{ii}^{-1} [A^T D^{-1} A x^k]_i \right) - M_{ii}^{-1} \sum_{s=1}^{k} [A^T D^{-1} A x^s]_i - \frac{1}{\alpha} M_{ii}^{-1} h_i(x_i^{k+1}) \,, \tag{93}$$

which proves the statement. □

Proof of Proposition 2.5. Essentially, Proposition 2.5 is an for us important Corollary of another Proposition which was omitted from section 2.2.3 due to its technical nature. We start this proof by stating and proving this other statement. For a by Algorithm 1 constructed sequence of x-iterates, we define the auxilary sequence

$$r^K := \sum_{k=1}^{K} Q x^k \,. \tag{94}$$

In order to obtain a more concise result, we introduce another auxiliary variable $q^k \in \mathbb{R}^{2mn}$ that combines x^k and r^k in one vector and an auxiliary matrix $H \in \mathbb{R}^{2mn \times 2mn}$ to measure the size of q

$$q^k := \begin{pmatrix} r^k \\ x^k \end{pmatrix}, \qquad H := \begin{pmatrix} I_{mn} & 0 \\ 0 & M - A^T D^{-1} A \end{pmatrix}, \tag{95}$$

leading to the seminorm $\|q^k\|_H^2 = \|r^k\|_2^2 + \|x^k\|_{M-A^T D^{-1} A}^2$.

Proposition A.1. *(Makhdoumi and Ozdaglar, 2017, Prop. 1)*
Let $r \in \mathbb{R}^{mn}$ be arbitrary and x^* an optimal primal solution of (24). Under Assumption 1, it holds for the sequence of q-iterates created by Algorithm 1 and for any $k \geq 0$ that

$$f(x^{k+1}) - f(x^*) + \alpha r^T Q x^{k+1} \leq \frac{\alpha}{2} \left(\|q^k - \tilde{q}^*\|_H^2 - \|q^{k+1} - \tilde{q}^*\|_H^2 - \|q^k - q^{k+1}\|_H^2 \right), \tag{96}$$

where $\tilde{q}^* := \begin{pmatrix} r \\ x^* \end{pmatrix}$.

Proof of Proposition A.1. By Lemma 2.4 we have

$$M \left(x^{k+1} - x^k \right) + \left(A^T D^{-1} A \right) x^k = -\frac{1}{\alpha} h(x^{k+1}) - \left(A^T D^{-1} A \right) \sum_{s=1}^{k} x(s) \tag{97}$$

Subtracting $(A^T D^{-1} A) x^{k+1}$ on both sides gives

$$\begin{aligned} \left(M - A^T D^{-1} A \right) \left(x^{k+1} - x^k \right) &= -\frac{1}{\alpha} h(x^{k+1}) - \left(A^T D^{-1} A \right) \sum_{s=1}^{k+1} x(s) \\ &= -\frac{1}{\alpha} h(x^{k+1}) - Q r^{k+1} \end{aligned} \tag{98}$$

where we used the definition of r^k and that $Q^2 = A^T D^{-1} A$. Rearanging (98) leads to

$$\frac{1}{\alpha} h(x^{k+1}) = -Q r^{k+1} - \left(M - A^T D^{-1} A \right) \left(x^{k+1} - x^k \right) \tag{99}$$

By the definition of a subgradient $h_i(x_i) \in \partial f_i(x_i)$ for a convex function f_i we have for any $y \in \mathbb{R}^n$, $i \in \{1, \ldots, m\}$ and any $k \geq 0$

$$h_i(x_i^{k+1})^T \left(x_i^{k+1} - y \right) \geq f_i(x_i^{k+1}) - f_i(y). \tag{100}$$

Summing this inequality up over all i gives

$$(x^{k+1} - \mathrm{y})^T h(x^{k+1}) \geq f(x^{k+1}) - f(\mathrm{y})\,. \tag{101}$$

for any $\mathrm{y} \in \mathbb{R}^{mn}$, in particular for $\mathrm{y} = x^*$. By first applying (101) with x^* and then the identity (99) we obtain

$$\begin{aligned}
&\frac{2}{\alpha}\left(f(x^{k+1} - f(x^*)\right) + 2r^T Q x^{k+1} \\
&\quad \leq \frac{2}{\alpha}\left(x^{k+1} - x^*\right)^T h(x^{k+1}) + 2r^T Q x^{k+1} \qquad (102)\\
&\quad = -2\left(x^{k+1} - x^*\right)^T \left(Qr^{k+1} + (M - A^T D^{-1} A)(x^{k+1} - x^k)\right) + 2r^T Q x^{k+1} \\
&\quad = -2\left(x^{k+1} - x^*\right)^T \left((M - A^T D^{-1} A)(x^{k+1} - x^k)\right) + 2\left(r - r^{k+1}\right)^T \left(r^{k+1} - r^k\right)
\end{aligned}$$

where we used in the last equality that $Qx^{k+1} = r^{k+1} - r^k$ and $Qx^* = 0$ since x^* is an optimal primal solution and therefore in particular feasible. Expanding the scalar product gives

$$2\left(r - r^{k+1}\right)^T \left(r^{k+1} - r^k\right) = \|r^k - r\|_2^2 - \|r^{k+1} - r\|_2^2 - \|r^k - r^{k+1}\|_2^2 \tag{103}$$

and analogously for the terms involving x

$$\begin{aligned}
&-2\left(x^{k+1} - x^*\right)^T \left((M - A^T D^{-1} A)(x^{k+1} - x^k)\right) \qquad (104)\\
&\quad = \|x^k - x^*\|^2_{M - A^T D^{-1} A} - \|x^{k+1} - x^*\|^2_{M - A^T D^{-1} A} - \|x^k - x^{k+1}\|^2_{M - A^T D^{-1} A}.
\end{aligned}$$

Inserting (103) and (104) into (102) and applying the definitions of $q^k, \tilde{q}^*$ and H proves Proposition A.1. □

We continue proving Proposition 2.5 by summing up the inequality of Proposition A.1 from $k = 0$ to $k = K - 1$, which gives for any $r \in \mathbb{R}^{mn}$

$$\begin{aligned}
&\sum_{k=0}^{K-1} f(x^{k+1}) - f(x^*) + \alpha r^T Q x^{k+1} \\
&\qquad \leq \frac{\alpha}{2}\left(\|q^0 - \tilde{q}^*\|_H^2 - \|q^K - \tilde{q}^*\|_H^2 - \sum_{k=0}^{K-1} \|q^k - q^{k+1}\|_H^2\right) \\
&\qquad \leq \frac{\alpha}{2}\|q^0 - \tilde{q}^*\|_H^2\,, \qquad (105)
\end{aligned}$$

where we resolved the telescoping sum and omitted the positive seminorm terms. At next, we lower bound the left-hand side by using the convexity of f from Assumption 1 and Jensen's inequality with $\lambda = 1/K$

$$\sum_{k=0}^{K-1} f(x^{k+1}) - f(x^*) + \alpha r^T Q x^{k+1} = K \sum_{k=1}^{K} \frac{1}{K} \overbrace{\left(f(x^k) + \alpha r^T Q x^k\right)}^{\text{convex function}} - K f(x^*)$$
$$\geq K \left(f(\hat{x}^K) + \alpha r^T Q \hat{x}^K - f(x^*)\right) , \quad (106)$$

where we used the definition of the ergodic average $\hat{x}^K$. Therefore, we obtain

$$f(\hat{x}^K) - f(x^*) + \alpha r^T Q \hat{x}^K \leq \frac{\alpha}{2K} \left\| q^0 - \tilde{q}^* \right\|_H^2 , \quad (107)$$

for any $r \in \mathbb{R}^{mn}$ by combining (105) and (106). Inserting the defintion of $\tilde{q}^*, q^0$ and H concludes the proof of Proposition 2.5. Note that $r^0 = 0$ per defintion. □

Proof sketch of Corollary 2.3. The proof follows directly from Theorem 2.2, the initialization of $x^0 = r^0 = 0$ and the following Lemma, which bounds the norms of x^* and of a specific optimal dual solution $\tilde{r}^*$ of (24).

Lemma A.2. *(Makhdoumi and Ozdaglar, 2017, Lem. 5+8)*
Let x^* be an optimal primal solution of (24). Then there exists a related optimal dual solution $\tilde{r}^*$ of (24) such that

$$\|\tilde{r}^*\|_2^2 \leq \frac{u^2}{\alpha^2 \tilde{\lambda}_{min}} , \quad (108)$$

where $\tilde{\lambda}_{min} \in \mathbb{R}$ is the smallest non-zero eigenvalue of $A^T D^{-1} A$ and u an upper bound on the norm of the subgradients of f at x^*, i.e. $\|h\|_2 \leq u$ for any $h \in \partial f(x^*)$. Moreover, we can bound

$$\|x^*\|_{M - A^T D^{-1} A}^2 \leq \lambda_{max} \|x^*\|_2^2 \leq \left(\frac{d_{min} + 2}{d_{min} + 1}\right) \|A\|_2^2 \|x^*\|_2^2 , \quad (109)$$

where $\lambda_{max} \in \mathbb{R}$ denotes the largest eigenvalue of $M - A^T D^{-1} A$.

For the proof of Lemma A.2 we refer to (Makhdoumi and Ozdaglar, 2017, Lem. 5 +8). □

B References

Banert, S., Ringh, A., Adler, J., Karlsson, J., and Öktem, O. (2020). Data-driven nonsmooth optimization. *SIAM Journal on Optimization*, 30(1):102–131.

Barber, R. F. and Sidky, E. Y. (2024). Convergence for nonconvex admm, with applications to ct imaging. *Journal of Machine Learning Research*, 25(38):1–46.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y., and Pascanu, R. (2018). Relational inductive biases, deep learning, and graph networks.

Bertsekas, D. (2009). *Convex Optimization Theory*. Athena Scientific optimization and computation series. Athena Scientific.

Boyd, S. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

Boyd, S. P., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

Cappart, Q., Chételat, D., Khalil, E., Lodi, A., Morris, C., and Veličković, P. (2023). Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61.

Chen, T., Chen, X., Chen, W., Heaton, H., Liu, J., Wang, Z., and Yin, W. (2022). Learning to optimize: A primer and a benchmark. *Journal of Machine Learning Research*, 23(189):1–59.

Eckstein, J. and Bertsekas, D. P. (1990). An alternating direction method for linear programming. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology.

Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hungary. Acad. Sci.*, 5:17–61.

Gabay, D. and Mercier, B. (1976). A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers mathematics with applications (1987)*, 2(1):17–40.

Ghadimi, E., Teixeira, A., Shames, I., and Johansson, M. (2012). On the optimal step-size selection for the alternating direction method of multipliers*. *IFAC Proceedings Volumes*, 45(26):139–144.

Glowinski, R. and Marroco, A. (1975). Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique*, 9(R2):41–76.

Godwin, J., Keck, T., Battaglia, P., Bapst, V., Kipf, T., Li, Y., Stachenfeld, K., Veličković, P., and Sanchez-Gonzalez, A. (2020). Jraph: A library for graph neural networks in jax.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Gori, M., Monfardini, G., and Scarselli, F. (2005). A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2.

Häusner, P., Öktem, O., and Sjölund, J. (2024). Neural incomplete factorization: learning preconditioners for the conjugate gradient method.

Ichnowski, J., Jain, P., Stellato, B., Banjac, G., Luo, M., Borrelli, F., Gonzalez, J. E., Stoica, I., and Goldberg, K. (2021). Accelerating quadratic optimization with reinforcement learning. *NeurIPS 2021*.

Lindholm, A., Wahlström, N., Lindsten, F., and Schön, T. (2022). *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press.

Makhdoumi, A. and Ozdaglar, A. (2017). Convergence rate of distributed ADMM over networks. *IEEE transactions on automatic control*, 62(10):5082–5095.

Marsden, A. (2013). Eigenvalues of the laplacian and their relationship to the connectedness.

Moore, N. S., Cyr, E. C., Ohm, P., Siefert, C. M., and Tuminaro, R. S. (2023). Graph neural networks and applied linear algebra.

Nedić, A. and Liu, J. (2018). Distributed optimization for control. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(Volume 1, 2018):77–103.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

Shi, W., Ling, Q., Yuan, K., Wu, G., and Yin, W. (2014). On the linear convergence of the ADMM in decentralized consensus optimization. *IEEE transactions on signal processing*, 62(7):1750–1761.

Sjölund, J. and Bånkestad, M. (2022). Graph-based neural acceleration for nonnegative matrix factorization.

Wei, E. and Ozdaglar, A. (2012). Distributed alternating direction method of multipliers. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 5445–5450. IEEE.

Wei, E. and Ozdaglar, A. (2013). On the o(1=k) convergence of asynchronous distributed alternating direction method of multipliers. In *2013 IEEE Global Conf. on Signal and Information Processing*, pages 551–554. IEEE.

Wright, S. and Recht, B. (2022). *Optimization for Data Analysis*. Cambridge University Press.

Xu, K., Li, J., Zhang, M., Du, S. S., Kawarabayashi, K., and Jegelka, S. (2020). What can neural networks reason about?

Yang, Y., Sun, J., Li, H., and Xu, Z. (2020). Admm-csnet: A deep learning approach for image compressive sensing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(3):521–538.