Learning to accelerate distributed ADMM using graph neural networks

Henri Doerks* HENRI.DOERKS@MATH.UU.SE
Department of Mathematics, Uppsala University, Sweden

Paul Häusner* PAUL.HAUSNER@IT.UU.SE
Daniel Hernández Escobar DANIEL.HERNANDEZ.ESCOBAR@IT.UU.SE
Jens Sjölund JENS.SJOLUND@IT.UU.SE
Department of Information Technology, Uppsala University, Sweden

Editors: G. Sukhatme, L. Lindemann, S. Tu, A. Wierman, N. Atanasov

Abstract

Distributed optimization is fundamental to large-scale machine learning and control applications. Among existing methods, the alternating direction method of multipliers (ADMM) has gained popularity due to its strong convergence guarantees and suitability for decentralized computation. However, ADMM can suffer from slow convergence and high sensitivity to hyperparameter choices. In this work, we show that distributed ADMM iterations can be naturally expressed within the message-passing framework of graph neural networks (GNNs). Building on this connection, we propose learning adaptive step sizes and communication weights through a GNN that predicts these hyperparameters based on the current iterates. By unrolling ADMM for a fixed number of iterations, we train the network end-to-end to minimize the solution distance after these iterations for a given problem class, while preserving the algorithm's convergence properties. Numerical experiments demonstrate that our learned variant consistently improves convergence speed and solution quality compared to standard ADMM, both within the trained computational budget and beyond.[†]

1. Introduction

The emergence of large-scale interconnected networked systems has driven the need for efficient distributed optimization algorithms across machine learning, control, and signal processing. In these systems, each agent typically has access only to local information, yet must coordinate with neighbors to solve a global optimization problem. The alternating direction method of multipliers (ADMM) is well-suited for this task and has gained increased popularity due to its flexibility and scalability (Boyd et al., 2011). A key feature of ADMM is that it can be implemented in a distributed fashion: agents perform computations locally and in parallel, then communicate updates to their neighbors and incorporate the received information into their local variables. While ADMM has a well-established convergence theory (Eckstein and Bertsekas, 1990), its convergence speed is highly sensitive to hyperparameter choices and can be slow in practice (Zhang et al., 2019).

A promising avenue for addressing these limitations lies in recent advances in machine learning. In particular, the learning-to-optimize framework (Chen et al., 2022) aims to automate the design of faster algorithms by using data-driven techniques that enhance the convergence of iterative methods for a given problem class. Building on this idea, we adopt an unrolling scheme (Monga et al., 2021),

*Equal contributions.
†The code is available at https://github.com/paulhausner/learning-distributed-admm.

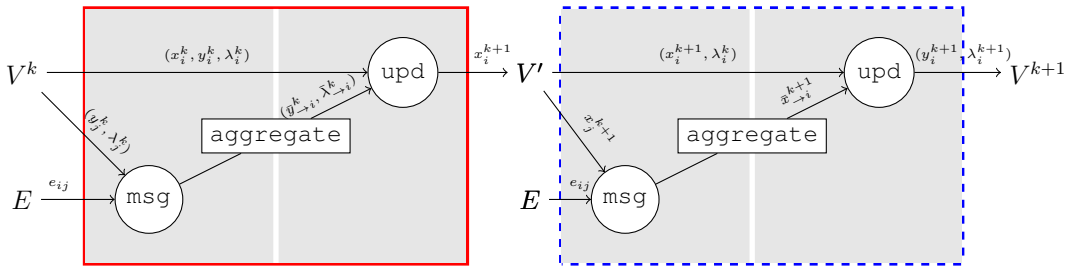

Figure 1: Computation structure of iteration k of the decentralized distributed ADMM algorithm as two message-passing steps. The initial node features V^k contain the previous ADMM iterates $(x_i^k, y_i^k, \lambda_i^k)$ for each node i and the non-zero edge features E describe the weighted pairwise connections in the network which is fixed across iterations. After two message-passing steps – each consisting of a message (msg), an aggregation (agg), and an update (upd) part – the new iterates $(x_i^{k+1}, y_i^{k+1}, \lambda_i^{k+1})$ of all nodes in the network are the output of the GNN.

where network parameters of our model are trained to minimize a loss function that reflects ADMM performance after a fixed number of iterations. In distributed optimization problems, which consist of an interconnected network of agents, graph neural networks (GNNs) (Battaglia et al., 2018; Scarselli et al., 2008) are a natural choice to parameterize the learning method as they can operate on graph-structured data that directly represents the underlying interconnected network.

In this paper, we show that classical decentralized and distributed ADMM iterations (Makhdoumi and Ozdaglar, 2017) can be implemented as a graph network using the message-passing framework (Battaglia et al., 2018) as we showcase in Figure 1 and further expand upon in Section 4.1. We explicitly exploit this connection and utilize GNNs to accelerate distributed ADMM.

In summary, our contributions in this paper are threefold: First, we show the one-to-one correspondence between distributed ADMM iterations (Makhdoumi and Ozdaglar, 2017) and message-passing networks (Gilmer et al., 2017), establishing an explicit algorithmic alignment (Veličković and Blundell, 2021) between ADMM and GNNs. Second, based on these insights, we develop and evaluate different approaches for learning instance-specific hyperparameters, namely the communication matrix and local or global step sizes, to accelerate the distributed ADMM algorithm. These approaches correspond to graph-, node-, and edge-level learning tasks. Finally, we propose a network architecture and a loss function to train the graph neural network in an end-to-end fashion by unrolling the ADMM iterations, ultimately resulting in an automated instance-specific hyperparameter tuning of ADMM. In numerical experiments, we demonstrate that our learning-based approaches reduce the distance from the iterates to the minimizer compared to the baseline methods and can improve the convergence of ADMM even beyond the unrolling steps used during training.

2. Related work

The goal of learning-to-optimize (L2O) approaches is to accelerate the convergence of parameterized optimization problems using machine learning techniques (Amos, 2023; Chen et al., 2022). We focus on accelerating the ADMM algorithm for distributed optimization problems; for a detailed overview of the ADMM algorithm we refer to the survey by Boyd et al. (2011).

While L2O approaches can yield impressive results in practice (Andrychowicz et al., 2016), they typically lack convergence guarantees and may fail to generalize, especially on unseen problem instances or when the underlying data distribution shifts (Amos, 2023). A popular approach

to ensure convergence, which we adopt, is to learn the hyperparameters of an existing optimization algorithm. Along those lines, Sambharya et al. (2024) learn to warm start fixed-point algorithms, including ADMM, by unrolling the iterations and using automatic differentiation to compute the gradient through the iterations. In a follow-up work, Sambharya and Stellato (2024) learn hyperparameters of fixed-point iterations. Similarly, Ichnowski et al. (2021) use reinforcement learning to tune the hyperparameters of OSQP, an ADMM-based solver for convex quadratic programs. These approaches have also been combined to solve quadratic programming problems with distributed ADMM (Saravanos et al., 2025). In contrast, we do not assume the presence of a central communication node. Further, we make no assumptions beyond the convexity of the objective function. A similar unrolling technique for ADMM iterations was successfully applied by Sjölund and Bånkestad (2022) for nonnegative matrix factorization. Another approach by Noah and Shlezinger (2024) learns the algorithm parameters directly through ADMM unrolling on a fixed set of agents. We instead learn a network that predicts instance-specific hyperparameters of the optimization algorithm. Biagioni et al. (2020) use a recurrent neural network to directly predict the solution to ADMM which is used to warm-start the method.

Additionally, we propose a novel way to learn the communication matrix. The predicted weighted Laplacian matrix can be seen as a form of learned preconditioning that stabilizes the iterates by improving problem conditioning (Teixeira et al., 2015). Machine learning methods have previously been applied successfully to learn preconditioners for linear systems (Häusner et al., 2024, 2025), imaging problems (Fahy et al., 2024), and gradient descent (Gao et al., 2025).

GNNs have been used for other L2O purposes previously, as they can represent various optimization problems, such as quadratic programs (Chen et al., 2025; Schmidtobreick et al., 2026), and can simulate the iterations of existing solvers such as interior point methods. However, while GNNs can, in principle, approximate the solution mapping by simulating the solver iterations, they provide no convergence guarantees (Qian et al., 2024). GNNs have also recently been applied to general distributed optimization. In contrast to approaches that start with a learned algorithm and add constraints inspired by theory (He et al., 2024), *we formulate distributed ADMM as a GNN operating on the communication network*. Then, we parameterize the iterations with neural networks, allowing the GNN to learn on the problem class, thereby replacing extensive manual hyperparameter tuning with a data-driven procedure.

3. Background

3.1. Distributed optimization

A distributed optimization problem consists of a network of m interconnected agents, represented by the nodes $\mathcal{V} = \{1, 2, \dots, m\}$ of a connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. An edge $(i, j) \in \mathcal{E}$ indicates that agent i and j can communicate. The agents jointly aim to minimize the sum of local objective functions f_i where each $f_i \colon \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ is only known to agent i and the data parameterizing the objective cannot be communicated. Limited communication between agents can arise, for instance, due to limited networking capacity, as in edge networks (Dal Fabbro et al., 2023), or due to privacy concerns (Chen and Wang, 2024). In such scenarios, we must solve the problem in a distributed manner, which requires decoupling the objective minimization and adding a consensus constraint:

$$\min_{x_1, x_2, \dots x_m} \quad \sum_{i=1}^{m} f_i(x_i) \quad \text{s.t.} \quad x_1 = x_2 = \cdots = x_m \,, \tag{1}$$

where each agent has its own local optimization variable $x_i \in \mathbb{R}^n$. The agents can reach consensus by communicating their local solutions to their neighbors as defined by the graph $\mathcal{G}$.

We assume that all f_i are convex and that the objective (1) is a proper, closed function. While there exist many different algorithms that solve this problem, we focus on distributed ADMM for peer-to-peer networks, i.e., the agents communicate directly with each other. In other words, we consider the *decentralized* setting, which is more general than having a central hub or intermediary.

3.2. ADMM

The alternating direction method of multipliers (ADMM) is an operator splitting technique that solves convex optimization problems of the form

$$\min_{x,z} \; f(x) + g(z) \quad \text{s.t.} \quad Ax + Bz = c\,, \tag{2}$$

where $x \in \mathbb{R}^p$, $z \in \mathbb{R}^q$, $A \in \mathbb{R}^{r\times p}$, $B \in \mathbb{R}^{r\times q}$, and $c \in \mathbb{R}^r$. Originally proposed by Glowinski and Marroco (1975) and Gabay and Mercier (1976), ADMM combines dual ascent, allowing parallel updates, and the method of multipliers, which converges under mild assumptions (Boyd et al., 2011).

Distributed ADMM Following the approach proposed by Makhdoumi and Ozdaglar (2017) we rewrite the distributed problem (1) to match the general ADMM formulation (2). In the rewritten form, the variable $x = [x_i]_{i=1}^m$ corresponds to the stacked individual variables of the agents and the function $f(x) = \sum_{i=1}^m f_i(x_i)$ is the global objective. Further, the consensus constraint in (1) is enforced by the equivalent formulation $(P \otimes I_n)x = 0$, where P is a $m \times m$ communication matrix satisfying $\text{null}(P) = \text{span}\{1_m\}$. This is satisfied by the (weighted) Laplacian $\mathcal{L}$ of a graph.

To allow parallel computation within ADMM, one can introduce an auxiliary variable z and define g as a convenient indicator function. Moreover, by exploiting the inherent symmetry of the optimization variables, the edge-based auxiliary variables z_{ij} can be replaced with corresponding node-based variables y_i, while the original dual variables are reparameterized as related variables λ_i. This reparametrization reduces redundancy and leads to a more compact and efficient formulation.

ADMM iterations Each iteration of the distributed ADMM algorithm updates all variables x_i, y_i, and λ_i for each agent i. The updates can be executed in parallel and require only information about the local objective and previous iterates of the agent's neighbors in the graph denoted by $\mathcal{N}(i) = \{j : (i,j) \in \mathcal{E}\} \subset \mathcal{V}$.[1] The local solution candidates x_i is the solution to a regularized version of the local subproblem of agent i and is computed as the minimizer of

$$x_i^{k+1} = \arg\min_{x_i} f_i(x_i) + \sum_{j\in\mathcal{N}(i)\cup\{i\}} \left((\lambda_j^k)^T (P_{ji}x_i) + \frac{\alpha}{2}\|P_{ji}(x_i - x_i^k) + y_j^k\|_2^2 \right), \tag{3}$$

where $\alpha > 0$ is the global step size hyperparameter. After communicating the updated iterates computed in (3) to all neighbors, the ADMM updates for y_i and λ_i are given by:

$$y_i^{k+1} = \frac{1}{d_i + 1} \sum_{j\in\mathcal{N}(i)\cup\{i\}} P_{ij} x_j^{k+1}, \qquad \lambda_i^{k+1} = \lambda_i^k + \alpha y_i^{k+1}. \tag{4}$$

Note that the variable $y = [y_i]_{i=1}^m$ is a weighted residual for consensus since the constraint $(P \otimes I_n)x = 0$ implies consensus on the components of x_i. The full algorithm is given in Alg. 1.

[1] We do not allow for self-loops, so $i \notin \mathcal{N}(i)$. Further, since $\mathcal{G}$ is undirected $(i,j) = (j,i) \in \mathcal{E}$.

Convergence Under the assumptions on the objective function from Section 3.1, the ergodic average of the computed local solutions x_i converges for every global step size $\alpha > 0$ and any communication matrix P with sublinear rate $\mathcal{O}(1/k)$ to the global solution. However, both the global step size α and the communication matrix P, affect the convergence speed in practice, and optimal choices appear to be instance-dependent (Makhdoumi and Ozdaglar, 2017). The convergence results can be extended to the case where there exists not one but a sequence of step sizes (Eckstein and Bertsekas, 1990), or even node-level step sizes $\alpha_i > 0$ (Barber and Sidky, 2024). Moreover, the solution to (3) does not need to be exact to ensure convergence (Eckstein and Bertsekas, 1990).

3.3. Graph neural networks

Graph neural networks (GNNs) learn a feature representation for each node in a graph based on input features and the underlying graph topology. In this paper, we focus on the framework of message-passing neural networks (MPNNs) which update the node features of a graph by aggregating information from the neighboring nodes and edges (Gilmer et al., 2017). Given edge weights e_{ij} for $(i, j) \in \mathcal{E}$ represented by the weighted adjacency E and initial node features v_i^0 for each node $i \in \mathcal{V}$, the latent node features v_i^{l+1} are iteratively updated in each message-passing step l by computing

$$v_i^{l+1} = \texttt{update}\left(v_i^l, \texttt{aggregate}(\{\texttt{message}(e_{ij} \cdot v_j^l : j \in \mathcal{N}(i))\})\right). \tag{5}$$

Here, the `update` and `message` functions can contain parameters that are learned during training, which are denoted by θ. We denote the node feature matrix in layer l with V^l. The `aggregate` function is typically permutation invariant, as there is no ordering of the graph neighbors, and can be implemented, for example, using the sum or mean function. This makes the overall architecture permutation equivariant, so the ordering of the nodes does not affect the output of the network.

4. Method

Our key observation is that the distributed ADMM iterations introduced in Section 3.2 can be rewritten in the form of two message-passing steps of form (5). In Section 4.1, we explicitly establish the connection between the ADMM iterations and GNNs. Further, we describe how the hyperparameters of the iterations can be parameterized using neural networks. In Section 4.2, we then describe how to train the parameters of this network using unrolled ADMM iterations.

4.1. Learning ADMM via message-passing networks

We start by deriving a one-to-one mapping from each distributed ADMM update step in (3) and (4) to a corresponding message-passing step. Each step consists of a respective `message`, `aggregate`, and `update` function. This correspondence is visualized in Figure 1.

Input representation The distributed optimization problem can be directly encoded as the input to a GNN. The underlying graph structure is naturally given by the communication network $\mathcal{G}$ from the problem formulation (1). The node features of the graph consist of the three ADMM iterates (x_i, y_i, λ_i). Furthermore, the non-negative edge weights $e_{ij} > 0$ of the graph represent the weighted connectivity, that determine the strength of the connection between two nodes in the graph. To obtain a valid communication matrix from the edge features, we construct the weighted Laplacian $P = \tilde{D} - E$ where $\tilde{D}$ is a diagonal $m \times m$ matrix of the weighted degree for each node $\tilde{D}_{ii} = \sum_{j \in \mathcal{N}(i)} e_{ji}$ and E is the symmetric matrix representation of the graph connectivity.

Message-passing steps We now reformulate the ADMM iteration in form of two message-passing steps. Recall that in the MPNN framework in (5), the `update` function only has access to the `aggregation` of the incoming messages and not the individual `messages`, as required in the original ADMM update step (3). To define a valid `update` function, we rewrite (3) equivalently to

$$x_i^{k+1} = \arg\min_{x_i} f_i(x_i) + \left(P_{ii}\lambda_i^k + \bar{\lambda}_{\to i}^k + \alpha(P_{ii}y_i^k + \bar{y}_{\to i}^k)\right)^T x_i + \frac{\alpha}{2} \sum_{j \in \mathcal{N}(i)\cup\{i\}} \|P_{ji}(x_i - x_i^k)\|_2^2. \quad (6)$$

In the rewritten form, $(\bar{\lambda}_{\to i}^k, \bar{y}_{\to i}^k)$ is the summation (`agg`) of all incoming `messages` to node i that we define by $(P_{ji}\lambda_j^k, P_{ji}y_j^k)$. Hence, in accordance with the MPNN framework, the refined `update` function in equation (6) now only depends on the aggregated features and the node's local information that is encoded in the node features. For details of the derivation, we refer to Appendix E. The updates for y and λ in equation (4) can be carried out jointly within a second message-passing step, and their translation is straightforward. It suffices to define the `message` between two adjacent nodes as $P_{ij}x_j^{k+1}$ and replace the respective part in (4) with the `aggregate` of these messages.

Together, this yields a two-step message-passing network that we visualize in Figure 1. Due to the equivalence to the original iteration in (3) and (4), the network's output coincides with the updated iterates of a single distributed ADMM iteration.

Learned components The message-passing steps derived in the previous paragraph rigidly reproduce the decentralized distributed ADMM iterations, without any learnable parameters that could improve performance. In the following, we present several ways to introduce learnable parameters in the message-passing steps that still maintain convergence under assumptions stated in Section 3.1. These approaches correspond to different levels of tasks for the GNN.

- **Graph-level task**: We learn a *global* step size by letting every node predict an α_i based on the local information and using the averaged $\alpha = \frac{1}{m}\sum_{i=1}^m \alpha_i$ for each node. In the distributed setting, this corresponds to introducing additional communication to reach consensus over the step size.
- **Node-level task**: Instead of choosing the same step size α for each node, we can learn an individual *local* α_i for each node. This avoids additional communication between the nodes since the averaging step is omitted. However, each node is required to choose α_i without knowledge about the step sizes of the other nodes, making it conceptually more challenging.
- **Edge-level task**: For each edge in the network, we learn the positive *edge weight* e_{ij} between two connected agents. Based on the edge weights predicted before the first ADMM iteration, we use the corresponding weighted Laplacian P as a fixed communication matrix for all steps.

For the first two tasks, the step size is predicted for each node individually using a neural network that receives as input the corresponding node features (x_i, y_i, λ_i). Additionally, we append the aggregated messages and the total number of nodes in the graph to the input. No further information about the local objective f_i is used as an input. We apply instance normalization before passing the input through the network (Ulyanov et al., 2016). At the beginning of the algorithm, we can predict and change the step size in every iteration, resulting in a separate MLP for every change. However, to ensure asymptotic convergence, we fix the step size after a predefined number of L iterations.[2]

For the predicted edge weights, we use the local degree profile (Cai and Wang, 2018), describing the local connectivity of the graph, as an input to an MLP. The feature representations of the

[2] In practice, we often choose $L = K$.

two nodes get stacked and passed through the network to predict a positive edge weight that preconditions the ADMM iterations. To ensure invariance to the node order and symmetrize the edge weight graph, we sum the predicted edge weights for each directed edge. To ensure the positivity of the learned algorithm hyperparameters e_{ij} and α, we use the `softplus` activation function.

4.2. Network training

In real-world applications, distributed optimization problems are solved repeatedly for similar communication networks $\mathcal{G}$ and local objectives f_i, providing potential for problem-class specific acceleration. We assume access to a dataset $\mathcal{D}$ consisting of problem instances that share an underlying structure to train our model to improve the average performance across the dataset.

Unrolling We apply a technique called algorithm unrolling (Chen et al., 2022), where the network is trained on a fixed number of iterations to maximize performance within this iteration budget. In our case, we construct a GNN, consisting of K distributed ADMM iterations via the message-passing scheme defined in the previous section. This requires $2K$ message-passing steps for the GNN implementation. Depending on the chosen learnable components, several MLPs are incorporated into the GNN to influence the `update` functions. The parameters of all the neural networks are denoted by θ. The final output contains the approximate solution to the optimization problem (1) which now depends on these parameters. The model is then trained using standard gradient-based learning methods that adjust the parameters θ to minimize a suitable loss function.

Loss function The loss function is designed such that it evaluates the GNN output $x^K = [x_i^K]_{i=1}^m$ only after the final K unrolled iterations of ADMM. Thus, we do not consider the learned ADMM performance after $k < K$ steps, nor how the iterates behave in subsequent iterations after K.

To train our model, we utilize a dataset $\mathcal{D}$, where each problem instance consists of a graph $\mathcal{G}$ and the per-node local objective functions f_i. Additionally, we have access to the true minimizer $x_d^\star$ that solves problem (1) for instance $d \in \mathcal{D}$ that can be computed offline efficiently since the problem is convex. Further, we precompute $[\hat{x}_{d,i}^K]_{i=1}^m$, the approximate solution obtained by running the distributed ADMM iteration with default hyperparameters. This is used to normalize the loss function in (7) and avoid over-weighting more involved instances in the training data. We train the network to minimize the empirical risk on the training data by minimizing the loss function

$$\ell(\theta; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{d=1}^{|\mathcal{D}|} \frac{1}{m_d} \left(\sum_{i=1}^{m_d} \frac{\|x_{d,i}^K(\theta) - x_d^\star\|_2^2}{\max\left(\|\hat{x}_{d,i}^K - x_d^\star\|_2^2, \varepsilon\right)} \right), \tag{7}$$

using stochastic gradient-based optimization and automatic differentiation. Here, m_d is the number of agents in instance $d \in \mathcal{D}$, and ε is a small positive constant that prevents division by zero.

This loss function is regression-based as it relies on the true solution to the optimization problem (1) that can be computed apriori and is part of $\mathcal{D}$ (Amos, 2023). As such, the loss penalizes deviations from $x_d^\star$ without explicitly enforcing the consensus constraint or minimizing the global objective (1). However, consensus is implicitly learned since all agents are trained to reach the same global solution. The denominator does not depend on the network parameters but is used to normalize the loss function across problem instances. Moreover, the normalized loss allows a natural interpretation of the network performance as it measures the network outcome relative to the baseline method with fixed hyperparameters. Using an objective-based loss function instead of (7) requires adding an explicit term for consensus among the agents and a careful design of the normalization.

5. Experiments

5.1. Problem setting

We evaluate our method on two different use cases of distributed ADMM that are inspired by real-world problems. More details about the problem generation can be found in Appendix B.

Network average consensus problem The first problem we consider is network consensus where all agents have local information $b_i \in \mathbb{R}^n$ and the goal is to find the mean of all information across the network (Zhang and Kwok, 2014). Despite its simplicity, this problem is a classic benchmark that highlights important design choices in distributed optimization. The objective function for each agent i in the network is $f_i(x_i) = \|x_i - b_i\|^2$.

Distributed least-squares problem For the second use-case, we consider the distributed least-squares problem frequently arising in machine learning (Yang et al., 2020). Each node has access to only c samples of the data given by $B_i \in \mathbb{R}^{c\times n}$ and corresponding labels $b_i \in \mathbb{R}^c$. The goal is to collaboratively find the least-squares solution with local objective:[3] $f_i(x_i) = \|B_i x_i - b_i\|^2$.

5.2. Results

We train all our models for $K = 10$ unrolling steps to minimize the loss function (7) and compare the results with two baseline methods: (i) a version that uses the best performing α as a fixed global step size, and (ii) an adaptive heuristic that adjusts α_i dynamically. Additional details about the model training and baseline selection can be found in Appendix C. In Table 1, we show the results of different learned methods corresponding to individual learning components and a combined version in terms of their ability to minimize the loss function (7) and achieve consensus among the iterates.

In the middle-left column, we show the key results obtained for $K = 10$ iterations, which is the assumed computational budget used for the loss function in (7). We can see that all learned methods outperform the baselines, achieving a lower error across all learned methods in both experimental settings. This also holds for most methods in terms of consensus among the iterates, even though the loss only influences this indirectly. Overall, the combined method where we learn both edge weights and step sizes clearly performs best in both metrics.

Additional iterations In the leftmost column of Table 1, we compare the results for 5 steps. Generally, the gap between the baseline methods and the learned methods is not as big as for the training objective. However, already after 5 iterations, the learned methods outperform the baselines in most cases, even though the training only evaluates the performance after $K = 10$ steps.

In the middle-right column, we show the performance of the (learned) ADMM algorithm after 20 iterations. For the additional iterations, we fix the step size at $\alpha = 1$ to guarantee convergence of ADMM and apply the same learned weighted Laplacian matrix. Except for the global step size in the consensus problem, we observe improvements compared to the baselines. Overall, the combined method demonstrates the strongest improvements. Results for 100 iterations, which is far beyond the training regime, are shown in the rightmost column. Apart from the combined method in the least-squares case, which appears to focus more on reaching consensus, all learned methods improve the baseline metrics. In summary, although our method is trained to perform well only after the trained 10 iterations, both earlier and later iterates still outperform the baseline methods in our experiments.

[3] In machine learning, the data is usually referred to as (x, y) pairs and the parameters of the model as θ. To be consistent with the ADMM framework above, we use $x \in \mathbb{R}^n$ as the regression parameters and (B_i, b_i) as the data.

Table 1: Comparison of different methods of the ADMM algorithm after a fixed number of k steps. Numbers reported are the average across the 100 test instances. The error measures the average distance of the local solutions to the global minimizer $\texttt{avg}\left(\|x_i^k - x^\star\|_2^2\right)$ and the consensus gap measures the degree to which the consensus constraint is violated as $\texttt{avg}\left(\|x_i^k - \bar{x}^k\|\right)$, where $\bar{x}^k$ is the average solution at step k. We compare with a fixed baseline using the best constant step size and an adaptive method which applies a dynamic selection of node-level step sizes. In the combined version, we learn both individual step size parameters α_i and the edge weights simultaneously. Lower is better for all metrics. Best is **bold**, second best underlined.

	$k = 5$		$k = 10$ (**train**)		$k = 20$		$k = 100$	
Consensus	Error	Consensus	Error	Consensus	Error	Consensus	Error	Consensus
Fixed	18.54	11.02	8.99	6.48	2.26	2.02	0.144	0.144
Adaptive	19.02	12.45	8.86	6.98	2.13	1.94	0.140	0.139
Global α	18.74	17.21	4.19	3.65	3.46	3.37	0.104	0.104
Local α_i	14.50	12.26	3.05	2.82	1.17	1.14	0.112	0.112
Weight e_{ij}	17.98	10.96	7.39	5.47	1.52	1.40	0.036	0.036
Combined	**13.39**	**8.05**	**1.96**	**1.76**	**0.76**	**0.65**	**0.001**	**0.001**
Least-squares	Error	Consensus	Error	Consensus	Error	Consensus	Error	Consensus
Fixed	70.89	10.17	53.35	7.68	30.19	2.43	1.28	0.04
Adaptive	70.81	10.64	52.79	7.95	29.58	2.31	1.25	0.04
Global α	61.32	23.41	27.12	9.21	14.93	3.48	0.67	0.04
Local α_i	**56.80**	14.66	23.79	7.37	12.96	1.84	0.62	0.03
Weight e_{ij}	67.83	14.19	43.99	8.20	19.45	2.51	**0.37**	0.03
Combined	58.97	**8.83**	**18.24**	**5.42**	**11.79**	**0.95**	1.94	**0.01**

Function value In addition, we evaluate the results in terms of the objective function value, measured by the relative objective at iteration k, i.e., the difference in objective values $\left|f(x^k) - f(x^\star)\right|$ normalized by the minimal value $|f(x^\star)|$. In Figure 2, the relative objective is plotted for 20 iterations. We can clearly see that the performance benefit of the learned methods is largest after the 10 unrolling steps used in training. When additional iterations are executed, the model using a global α shows a large initial increase in the objective, likely due to overfitting on the number of unrolling iterations and freezing the step size. In the least-squares case, we also observe that the method using the learned communication matrix leads to a higher objective value for most of the iterations, even though the distance to the minimizer decreases. However, when combining the communication matrix with the learned step size, the performance significantly improves, beyond simply adding the two individual contributions of these hyperparameters. This shows that preconditioning still benefits the ADMM convergence, especially in synergy with learned step sizes. More results comparing wall-clock times and out-of-distribution generalization of the methods can be found in Appendix D.

6. Discussion & conclusion

We have shown how the distributed ADMM algorithm can be accelerated by expressing the updates in form of message-passing steps that can be parametrized to learn the step size and communication matrix. Several other approaches could further exploit this connection. In the following we highlight some of these potential extensions and discuss limitations of our approach.

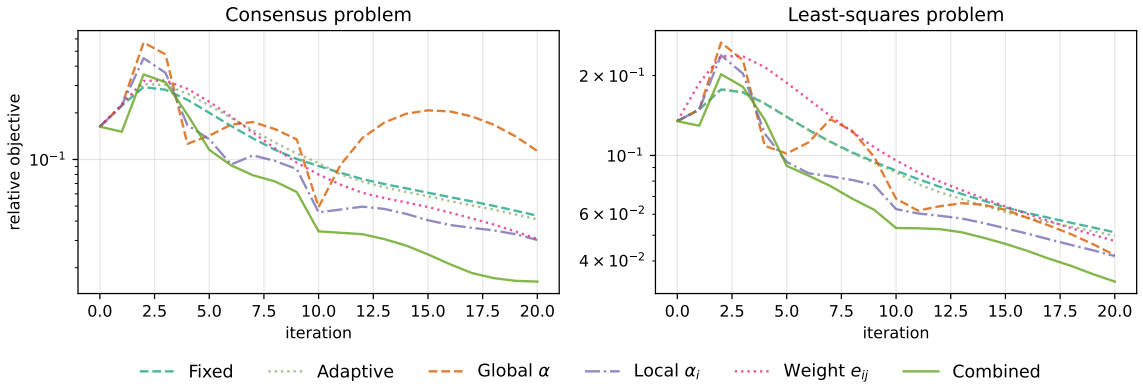

Figure 2: Semi-log plots of the mean relative objective over all 100 test instances for 20 iterations. The learned methods are only trained on the first 10 iterations as in previous results.

Limitations ADMM has robust guarantees but is known to suffer from slow convergence when approaching the optimum. Since we intentionally stay within the ADMM framework, we cannot expect to overcome this inherent weakness. We only train the network to perform well within a fixed number of iterations K across all instances. This does not necessarily translate into improved performance on every individual instance, or faster global convergence beyond the K iterations, which is a general limitation of unrolling (Arisaka and Li, 2023). We observe this in our experiments, where the combined method leads to a larger error than the baselines when increasing the iterations significantly. However, practical applications typically have a computational budget, making unrolling a useful technique for real-world problems.

Future work We only augment ADMM by learning to predict the hyperparameters. However, it is also possible to replace some of the ADMM steps completely. For example, the update of the x-variable requires solving a potentially time-consuming optimization problem (Li et al., 2023). While it is tempting to replace this with the output of a neural network directly, ensuring convergence for such inexact methods is significantly more challenging (Banert et al., 2024). Additionally, the exchange of messages involving all iterates can be limited by the network bandwidth. ADMM can be extended to only communicate compressed information without losing convergence (Chen et al., 2026).

We focus on decentralized distributed ADMM (Makhdoumi and Ozdaglar, 2017) while many other formulations of distributed ADMM exist that can leverage graph neural networks. In particular, distributed ADMM with a centralized communication hub can be mapped to message-passing with a global node, opening up many more synergies between distributed optimization and GNNs.

Conclusion In this paper, we demonstrate an equivalence between distributed ADMM and message-passing networks that, to our knowledge, has not been explicitly recognized before. Based on this insight, we describe a learning-to-optimize approach that predicts instance-specific hyperparameters for distributed ADMM. The model is trained by unrolling the algorithm and minimizing the normalized distance between the true minimizer and the network output for a specific problem distribution. These learning tasks can be naturally expressed as different graph learning tasks leveraging the connection between ADMM and GNNs. In numerical experiments, we validate that the learned algorithm outperforms the baseline methods and leads to faster convergence of ADMM.

Acknowledgments

The authors thank Daniel Arnström for helpful discussions and suggestions and the anonymous reviewers for their thoughtful comments. This research was funded in part by Sweden's Innovation Agency (Vinnova), grant number 2022-03023, the Centre for Interdisciplinary Mathematics at Uppsala University (CIM), and supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

Brandon Amos. Tutorial on amortized optimization. *Foundations and Trends in Machine Learning*, 16(5):592–732, 2023.

Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International conference on machine learning*, pages 136–145. PMLR, 2017.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

Sohei Arisaka and Qianxiao Li. Principled acceleration of iterative numerical methods using machine learning. In *International Conference on Machine Learning*, pages 1041–1059. PMLR, 2023.

Sebastian Banert, Jevgenija Rudzusika, Ozan Öktem, and Jonas Adler. Accelerated forward-backward optimization using deep learning. *SIAM Journal on Optimization*, 34(2):1236–1263, 2024.

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

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

David Biagioni, Peter Graf, Xiangyu Zhang, Ahmed S Zamzam, Kyri Baker, and Jennifer King. Learning-accelerated admm for distributed dc optimal power flow. *IEEE Control Systems Letters*, 6:1–6, 2020.

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011. ISSN 1935-8237.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Chen Cai and Yusu Wang. A simple yet effective baseline for non-attributed graph classification. *arXiv preprint arXiv:1811.03508*, 2018.

Chuyan Chen, Yutong He, Pengrui Li, Weichen Jia, and Kun Yuan. Greedy low-rank gradient compression for distributed learning with convergence guarantees. *IEEE Transactions on Signal Processing*, 2026.

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *Journal of Machine Learning Research*, 23(189):1–59, 2022.

Ziang Chen, Xiaohan Chen, Jialin Liu, Xinshang Wang, and Wotao Yin. Expressive power of graph neural networks for (Mixed-integer) quadratic programs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 13–19 Jul 2025.

Ziqin Chen and Yongqiang Wang. Privacy-preserving distributed optimization and learning. *arXiv preprint arXiv:2403.00157*, 2024.

Nicolò Dal Fabbro, Michele Rossi, Luca Schenato, and Subhrakanti Dey. Q-shed: Distributed optimization at the edge via hessian eigenvectors quantization. In *ICC 2023-IEEE International Conference on Communications*, pages 4403–4408. IEEE, 2023.

Jonathan Eckstein and Dimitri P. Bertsekas. An alternating direction method for linear programming. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1990.

P Erdös and A Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

Patrick Fahy, Mohammad Golbabaee, and Matthias J Ehrhardt. Greedy learning to optimize with convergence guarantees. *arXiv preprint arXiv:2406.00260*, 2024.

Daniel Gabay and Bertrand Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & mathematics with applications (1987)*, 2(1):17–40, 1976. ISSN 0898-1221.

Wenzhi Gao, Ya-Chi Chu, Yinyu Ye, and Madeleine Udell. Gradient methods with online scaling part i. theoretical foundations. *arXiv preprint arXiv:2505.23081*, 2025.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

R. Glowinski and A. Marroco. 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, 1975. ISSN 0397-9342.

Jonathan Godwin, Thomas Keck, Peter Battaglia, Victor Bapst, Thomas Kipf, Yujia Li, Kimberly Stachenfeld, Petar Veličković, and Alvaro Sanchez-Gonzalez. Jraph: A library for graph neural networks in jax., 2020. URL http://github.com/deepmind/jraph.

Paul Häusner, Ozan Öktem, and Jens Sjölund. Neural incomplete factorization: learning preconditioners for the conjugate gradient method. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

Paul Häusner, Aleix Nieto Juscafresa, and Jens Sjölund. Learning incomplete factorization preconditioners for GMRES. In *Proceedings of the 6th Northern Lights Deep Learning Conference (NLDL)*, volume 265 of *Proceedings of Machine Learning Research*, pages 85–99. PMLR, 2025.

Bing-Sheng He, Hai Yang, and SL Wang. Alternating direction method with self-adaptive penalty parameters for monotone variational inequalities. *Journal of Optimization Theory and applications*, 106(2):337–356, 2000.

Yutong He, Qiulin Shang, Xinmeng Huang, Jialin Liu, and Kun Yuan. A mathematics-inspired learning-to-optimize framework for decentralized optimization. *arXiv preprint arXiv:2410.01700*, 2024.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL http://github.com/google/flax.

Jeffrey Ichnowski, Paras Jain, Bartolomeo Stellato, Goran Banjac, Michael Luo, Francesco Borrelli, Joseph E Gonzalez, Ion Stoica, and Ken Goldberg. Accelerating quadratic optimization with reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 21043–21055, 2021.

Meiyi Li, Soheil Kolouri, and Javad Mohammadi. Learning to optimize distributed optimization: Admm-based dc-opf case study. In *2023 IEEE Power & Energy Society General Meeting (PESGM)*, pages 1–5. IEEE, 2023.

Ali Makhdoumi and Asuman Ozdaglar. Convergence rate of distributed ADMM over networks. *IEEE transactions on automatic control*, 62(10):5082–5095, 2017. ISSN 0018-9286.

Vishal Monga, Yuelong Li, and Yonina C Eldar. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. *IEEE Signal Processing Magazine*, 38(2):18–44, 2021.

Yoav Noah and Nir Shlezinger. Distributed learn-to-optimize: Limited communications optimization over networks via deep unfolded distributed ADMM. *IEEE Transactions on Mobile Computing*, 2024.

Chendi Qian, Didier Chételat, and Christopher Morris. Exploring the power of graph neural networks in solving linear optimization problems. In *International Conference on Artificial Intelligence and Statistics*, pages 1432–1440. PMLR, 2024.

Rajiv Sambharya and Bartolomeo Stellato. Learning algorithm hyperparameters for fast parametric convex optimization. *arXiv preprint arXiv:2411.15717*, 2024.

Rajiv Sambharya, Georgina Hall, Brandon Amos, and Bartolomeo Stellato. Learning to warm-start fixed-point optimization algorithms. *Journal of Machine Learning Research*, 25(166):1–46, 2024.

Augustinos D Saravanos, Hunter Kuperman, Alex Oshin, Arshiya Taj Abdul, Vincent Pacelli, and Evangelos Theodorou. Deep distributed optimization for large-scale quadratic programming. In *The Thirteenth International Conference on Learning Representations*, 2025.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

Ella J Schmidtobreick, Daniel Arnström, Paul Häusner, and Jens Sjölund. Warm-starting active-set solvers using graph neural networks. In *8th Annual Conference on Learning for Dynamics and Control*, 2026.

Jens Sjölund and Maria Bånkestad. Graph-based neural acceleration for nonnegative matrix factorization. *arXiv preprint arXiv:2202.00264*, 2022.

Andre Teixeira, Euhanna Ghadimi, Iman Shames, Henrik Sandberg, and Mikael Johansson. The admm algorithm for distributed quadratic problems: Parameter selection and constraint preconditioning. *IEEE Transactions on Signal Processing*, 64(2):290–305, 2015.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7), 2021.

Tao Yang, Jemin George, Jiahu Qin, Xinlei Yi, and Junfeng Wu. Distributed least squares solver for network linear equations. *Automatica*, 113:108798, 2020.

Juyong Zhang, Yue Peng, Wenqing Ouyang, and Bailin Deng. Accelerating admm for efficient simulation and optimization. *ACM Transactions on Graphics (TOG)*, 38(6):1–21, 2019.

Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1701–1709, Bejing, China, 22–24 Jun 2014. PMLR.

Appendix A. Glossary of Notation

Table 2: Summary of the notation used throughout the paper.

Symbol	Description	Comments
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Graph with set of vertices $\mathcal{V}$ and edges $\mathcal{E}$	$\mathcal{V} = \{1, 2, \ldots, m\}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$
$\mathcal{N}(i)$	Set of neighboring nodes of i	By default we assume $i \notin \mathcal{N}(i)$
d_i	Degree of node i	Equal to $\lvert\mathcal{N}(i)\rvert$
$D, \tilde{D}$	Degree matrix and weighted degree matrix	
m	Number of agents / nodes in the graph	
n	Dimension of the local optimization problem	
f_i	Local objective function of node / agent i	
b_i, B_i	Local objective function information	
x_i	Local optimization variable of node / agent i	
I_n	Identity matrix of size n	
1_n	Vector of all ones of size n	
$V = [v_i]_{i=1}^m$	Node features in GNN	
$E = [e_{ij}]$ for $(i, j) \in \mathcal{E}$	Edge features in GNN	Typically implemented as a sparse matrix
l	Layer in GNN	Twice as many GNN layers as ADMM steps
θ	All learnable network parameters	
$\mathcal{D}$	Dataset used for training	
ε	Small constant to avoid division by zero	
α	ADMM step size	
$\mathcal{L}$	Weighted Laplacian matrix	$\mathcal{L} = D - E$
P	Communication matrix	
x, z	ADMM primal variables	Stacked and slack distributed variables
y	ADMM distributed variable	This is a reformulation of the z variable
λ	ADMM dual variable	
r, s	Primal and dual ADMM residuals	
μ, τ	Adaptive step size parameters	
A, B	ADMM linear constraint matrices	
k	Iterations of the ADMM algorithm	
K	Total number of ADMM steps in the unrolling	
L	Number of steps to switch to fixed step size	Often $L = K$

Appendix B. Problem details

Here, we describe in more detail how we generate the different problem classes used in the experiments in Section 5 and describe how to generate ground-truth labels for them which are used for the supervised training as well as provide details on how the subproblem in equation (3) is solved in practice. Further, we provide the additional information to describe how the distribution of problems is generated for each problem.

Across all problem domains, the underlying graph structure is based on the Erdős-Rényi random graph model (Erdös and Rényi, 1959) with $m = 8$ nodes. The edge probability is fixed to $p = 0.4$ in the experiments. We discard sampled graphs that are not connected. Moreover, the dimensionality of the solution $x^\star \in \mathbb{R}^n$ is fixed at $n = 2$ in all experiments.

For each problem class, we generate a total of 900 instances for training, 100 instances for validation and 100 instances for testing purposes. All results in the main text are evaluated on the testing instances which are not seen by the network during training.

B.1. Network average consensus

For the network average consensus problem, each node or agent $i \in \{1, \dots, m\}$ in the network has a local objective function that minimizes the distance between the local optimization variable x_i and its local private data b_i.

Problem generation This local objective function is parameterized for every agent by a vector $b_i \in \mathbb{R}^n$. The vector is sampled independently from a n-dimensional normal distribution $\mathcal{N}(0, \Sigma)$ with covariance matrix $\Sigma = 100 I_n$.

Subproblem solution The solution to the subproblem (3) for this use-case is a special case of the result presented for the distributed least-squares problem below with $B_i = I_n$. Since the resulting linear system for the problem that we derive in equation (9) is diagonal, the problem can be solved more efficiently in practice requiring only $\mathcal{O}(n)$ steps and does not require expensive matrix inversion techniques. Instead, we can take the reciprocal of the diagonal elements to obtain the inverse. Note that these elements are always non-zero by the construction of the linear system, and thus the inverse is well-defined.

Global solution For training, we assume access to the global solution $x^\star$ for all problem instances. We can efficiently generate these solutions by utilizing the underlying problem structure and the direct access to all problem data. The minimizer is given by

$$x^\star = \frac{1}{n} \sum_{i=1}^{n} b_i \tag{8}$$

In other words, the agents are collaborating to find the mean of their vectors b_i without sharing the actual data.

B.2. Distributed least squares problem

In the distributed least squares problem, the local objective function solves the least-squares problem for the privately available data. In addition to the data vector b_i from the network average consensus problem, each agent also has a private feature transformation matrix $B_i \in \mathbb{R}^{n \times n}$. The latter is used to weight the linear regression coefficients of x_i before comparing with b_i.

Problem generation The local objective function is parametrized for every agent by the pair $(B_i, b_i) \in \mathbb{R}^{n \times n} \times \mathbb{R}^n$. Each entry of the matrix B_i is sampled from a continuous uniform distribution $\mathcal{U}[0, 1]$ on the unit interval. In addition, we assume in our experiments that B_i is of full rank to obtain better convergence behavior of ADMM and ensure closed-form solutions. We enforce this by discarding and resampling B_i in case the magnitude of an eigenvalue is below $1/10$. The data vector b_i is sampled as before.

Subproblem solution In the distributed least squares problem, the solution to the optimization problem to update the x-variable from (3) has a closed form solution and can be solved as the solution to the following linear equation system:

$$x_i^{k+1} = (2B_i^T B_i + \alpha M_{ii})^{-1} \big(2B_i^T b_i - P_{ii} \lambda_i^k - \bar{\lambda}_{\to i}^k + \alpha (M_{ii} x_i^k - P_{ii} y_i^k - \bar{y}_{\to i}^k)\big), \tag{9}$$

where M_{ii} is a diagonal matrix defined by

$$\begin{aligned} M_{ii} :&= \left(\sum_{j \in \mathcal{N}(i)} P_{ji}^T P_{ji} + P_{ii}^T P_{ii} \right) \otimes I_n \\ &= \left(\sum_{j \in \mathcal{N}(i)} e_{ji}^2 + \Big(\underbrace{\sum_{j \in \mathcal{N}(i)} e_{ji}}_{\text{weighted degree of } i} \Big)^2 \right) \otimes I_n \end{aligned} \tag{10}$$

since $P_{ii} = \tilde{D}_{ii} = \sum_{j \in \mathcal{N}(i)} e_{ji}$ is the weighted degree in the case where we choose the weighted Laplacian as our communication matrix for the algorithm. Note that the (weighted) degree and the sum of the weighted connections can be easily computed as a message in the MPNN.

We can observe in particular that the matrix here is positive definite since $B_i^T B_i$ is positive (semi)-definite and M_{ii} is a diagonal matrix with positive diagonal entries. Therefore, we can use the iterative conjugate gradient method to compute the solution to the equation system which scales computationally better than inverting or factorizing the matrix for large-scale problems.

Global solution The global minimizer is given by

$$x^\star = \left(\sum_{i=1}^m B_i^T B_i \right)^{-1} \left(\sum_{i=1}^m B_i^T b_i \right) . \tag{11}$$

Note that the inverse exists since again $B_i^T B_i$ is positive definite for every i, due to the construction of the training data, and hence is the sum described in the problem generation paragraph.

Appendix C. Implementation details

In the following, we discuss the details of the training scheme and the chosen hyperparameters and tuning steps in order to ensure reproducibility of our presented results.

Framework We use the JAX framework (Bradbury et al., 2018) to implement the differentiable distributed ADMM algorithm as a message-passing network. In particular, we are using the `jraph` toolkit to implement the message-passing steps (Godwin et al., 2020) and integrate it with the `flax` library to design neural network components using the novel `linen` API (Heek et al., 2024).

Optimization subproblem There is one obvious difficulty in making the GNN fully differentiable. In each update of the node features x_i, an optimization problem of the form (6) needs to be solved, and the gradient of the network parameters θ needs to be computed with respect to the solution to this optimization problem. In the case of step size learning, all of α_i, the previous iterates, and incoming messages depend on the parameters of the network through previous updates. To still compute the gradient, we need to differentiate the solution to the optimization problem with respect to the parameters of the network $\frac{\partial x_i^{k+1}}{\partial \theta}$ which can be decomposed via the chain rule $\frac{\partial x_i^{k+1}}{\partial \alpha_i} \cdot \frac{\partial \alpha_i}{\partial \theta}$.

Computing the first part can be achieved in closed form by the means of implicit differentiation (Amos and Kolter, 2017). However, implicit differentiation can in practice be costly and requires additional overhead. Instead, we solve the subproblem (6) approximately by unrolling

an iterative solver[4] or use the closed-form solution to (6) if it exists. In both cases, we compute its gradient using standard automatic differentiation techniques. Along the same lines, the partial derivative of the optimization problem (3) with respect to the learned edge weights e_{ij} is computed. The remaining update steps can be differentiated using standard automatic differentiation tools provided by JAX (Bradbury et al., 2018).

Network architectures For the network components to predict the hyperparameters, within the message-passing steps, we are using a 2-layer multi-perceptrons. The hidden layer has 32 neurons and there is 1 output which gets passed through a `softplus` activation function to ensure positivity. We use a `ReLU` non-linearity between the two layers.

- We use a single network for all iterations from $k = 2$ until $K = 10$. For the first step, we do not use learning but use the default step size. Thus, there are a total of 9 learned steps in the unrolling scheme. The input to the network is given by the stacked optimization variables of the current iteration. Thus, the total number of trainable parameters is 3 753.
- The input for the edge weight prediction is given by the concatenated local degree profiles of the adjacent nodes (Cai and Wang, 2018). Each of these consists of 5 features containing statistics about the the neighborhood structure including the degree, maximum and minimum neighboring degree, as well as the mean and variance. To ensure that the communication matrix is symmetric, we sum up the edge predictions. Further, we use the same predicted communication matrix for all iterations. Thus, the number of parameters in the model is 385.
- In the combined model, we use both the step size network and the edge weight network together. Thus, the total number of parameters is 4 138.

Training hyperparameters For the training, we are using the Adam optimizer with a learning rate of 10^{-4} and global gradient clipping of with radius 1.0. We are training the model for 100 epochs using a batch size of 5 via gradient accumulation. Thus, during training a total of 18 000 parameter update steps are executed. To avoid division by zero, we include the hyperparameter ε into the loss function in (7). In the experiments, we set $\varepsilon = 10^{-5}$.

ADMM hyperparameters If not learned or further specified, we apply the naive step size $\alpha = 1$ and the Laplacian matrix $\mathcal{L}$ of the graph as the communication matrix P in all algorithm steps. Like the Laplacian, the degree d_i of each node i depends on the respective graph instance.

ADMM initialization In our experiments, we initialize all x_i by $\mathbf{0} \in \mathbb{R}^n$ and hence also all y_i by $\mathbf{0} \in \mathbb{R}^n$. While it is generally possible to use warm starting with the proposed framework, e.g. through learning (Sambharya et al., 2024), additional algorithm steps are required in this case to ensure that the consensus variables y_i are appropriately initialized. For details, see lines 2-6 in Algorithm 1.

Fixed baseline The fixed baseline method uses a constant, global α across all iterations, which is chosen via grid search on the validation data. We allow α to be in the interval $[0.001, 10]$ using linear spacing for 100 different values.[5] This is a form of black-box optimization. To choose the α

[4] Here, we unroll the iterative solver within each unrolled iteration of ADMM.

[5] We choose the initial interval such that it covers the empirically observed learned step sizes across the unrolled iterations.

that performs best on average for all instances, we use the same objective as in training, defined in equation (7). For both problem classes, we select the α in our grid that performs best in this metric on the validation data after $K = 10$ iterations, and apply it on the test data in the reported results. This leads to selecting $\alpha = 1.112$ for the consensus problem, and $\alpha = 1.011$ for the least squares problem. This approach is similar to the method proposed by Noah and Shlezinger (2024) in the case where only one constant step size is learned, but it relies on black-box optimization rather than gradient-based optimization to determine the parameters.

Adaptive baseline In the adaptive baseline, the local node-level step sizes α_i gets adjusted dynamically based on the balance of primal and dual residuals (Boyd et al., 2011). Intuitively, if the primal residual is too large, we increase α to shift the focus towards enforcing the consensus, while if the dual residual is too large, we decrease the step size to allow for more flexibility in the local node updates. We utilize the common heuristic introduced by He et al. (2000) to update the step size in each step as:

$$\alpha^{k+1} = \begin{cases} \tau\alpha^k & \text{if } \|r^k\| > \mu\|s^k\|, \\ \tau^{-1}\alpha^k & \text{if } \|s^k\| > \mu\|r^k\|, \\ \alpha^k & \text{otherwise.} \end{cases} \tag{12}$$

Here, $\tau \in \mathbb{R}_+$ and $\mu \in \mathbb{R}_+$ are parameters of the adaptive scheme. Typical default choices are $\tau = 2$ and $\mu = 10$ (He et al., 2000). The primal and dual residual at iteration k are denoted by r^k and s^k respectively and defined as:

$$\begin{aligned} r_{ij}^{k+1} &= P_{ij}x_j^{k+1} - z_{ij}^{k+1} = y_i^{k+1}, \\ s_j^{k+1} &= \alpha \sum_{i=1}^{m} \left[P_{ij}^2 \left(x_j^{k+1} - x_j^k \right) + P_{ij} \left(y_i^{k+1} - y_i^k \right) \right] \\ &= \alpha \left(\left[\sum_{i=1}^{m} P_{ij}^2 \right] (x_j^{k+1} - x_j^k) + d_j(y_j^{k+1} - y_j^k) + \bar{y}_{\to j}^{k+1} - \bar{y}_{\to j}^{k} \right) \end{aligned} \tag{13}$$

The second formulation of the dual residual is readily available in our message-passing framework. Note that the primal residual is an edge-based version of the newly introduced y^k in the distributed ADMM framework as defined in equation (4), measuring the constraint violation of the primal problem. The dual residual is a measure of change in this auxiliary variable and the local solution x_i. In order to compute a node-based primal residual for predicting a node-based α_i with the step size heuristic, we use

$$\|r_i\|_2 = \sqrt{\sum_{j \in \mathcal{N}(i)} r_{ij}^T r_{ij}} = \sqrt{d_i \, y_i^T y_i} = \sqrt{d_i} \, \|y_i\|_2. \tag{14}$$

As for the fixed baseline, we choose the hyperparameters of the adaptive scheme using grid search over the validation data, based on average performance after $K = 10$ iterations. For the μ parameter, that controls how frequently the step size parameter is changed, we search over the choices $\{1, 5, 10, 15, 20, 25, 30, 25, 30, 35, 40\}$. For the scaling parameter, that adjusts the step size we pick values in the interval $(1, 2]$. The grid search picks 20 different values from the interval with logarithmic spacing. Thus, in total we evaluate 200 different combinations of parameters for the tuning of the adaptive step size scheme. From this, we obtain the optimal values $\mu = 10$ and

$\tau = 1.32$ for the consensus problem, and similarly $\mu = 10$ and $\tau = 1.36$ for the least-squares problem. In order to avoid numerical breakdowns, we reset the step size to the fixed value of $\alpha = 1$ after the first $K = 10$ iterations, as in the other methods.

Computational resources All reported experiments are executed on a single NVIDIA TITAN Xp GPU with 12 GB of memory. Training a single epoch requires 10 minutes plus additional time for the validation process. In total, the training of one model for 100 epochs requires around 20 hours.

Appendix D. Additional results

Here, we provide additional results to complement the empirical evaluation of our methods based on the convergence time, the distribution of solving times, and the out-of-domain (OOD) performances.

D.1. ADMM convergence and solver time

In Table 3, we compare the different methods based on achieving a certain threshold in relative objective value within a given computational budget of one second. For every instance on which a certain method reaches the threshold within the budget, we measure the required number of iterations and wall-clock time. In the table, we report these values as averages across successful instances per method. Further, we report the fraction of successfully solved instances within the budget. Note that this leads to a dependent metric since the iterations and time are conditional on solving. Thus, we need to evaluate the different methods based on their Pareto front instead of look-

Table 3: Timing and convergence comparison of the different methods until the relative objective reaches a threshold ε. The total computational budget for each problem is 1 second. For each solver we check how many instances achieve the desired accuracy within this budget and report the resulting success rate (↑ higher is better). For the successfully solved instances per method, we measure the average number of iterations and required time in seconds (↓ lower is better). Best is **bold**, second best underlined.

	$\varepsilon = 0.05$			$\varepsilon = 0.01$			$\varepsilon = 0.001$		
Consensus	Iterations	Time	Success	Iterations	Time	Success	Iterations	Time	Success
Fixed	11.64	0.23	0.94	16.36	0.30	0.87	29.42	0.50	0.76
Adaptive	11.30	0.30	0.93	15.76	0.36	0.83	26.85	0.57	0.71
Global α	7.82	0.20	0.96	19.37	0.39	0.91	35.31	0.64	0.78
Local α_i	**5.61**	**0.16**	0.96	12.72	0.28	0.89	26.88	0.51	0.80
Weight e_{ij}	11.19	0.24	0.96	15.62	0.32	0.92	27.38	0.46	0.82
Combined	**5.61**	0.17	**0.98**	**10.47**	**0.25**	**0.96**	**23.56**	**0.44**	**0.90**
Least-squares	Iterations	Time	Success	Iterations	Time	Success	Iterations	Time	Success
Fixed	9.61	0.30	0.85	14.72	0.44	0.60	21.23	0.62	0.26
Adaptive	9.20	0.32	0.85	13.29	0.44	0.59	20.23	0.62	0.26
Global α	**5.31**	0.22	**0.94**	11.30	0.36	0.76	18.47	0.58	0.34
Local α_i	6.27	0.22	0.89	11.67	0.39	0.76	18.97	0.60	0.35
Weight e_{ij}	10.88	0.33	0.86	15.68	0.47	0.63	19.54	0.56	0.39
Combined	5.41	**0.21**	**0.94**	**10.13**	**0.34**	**0.83**	**17.90**	**0.53**	**0.41**

ing at the metrics individually. A method is on the Pareto front if no other method achieves both a higher success rate and lower iteration/time performance at the same time.

Across both problem classes, all learned solvers are able to solve more instances within the computational budget than the fixed and adaptive baseline. Moreover, for every threshold, there exist multiple learned methods that are Pareto improvements to the two baselines, i.e. more instances are solved while requiring fewer average iterations and solving them in faster average time. In particular, the combined method appears to be the strongest choice as it is always a Pareto improvement over the baselines, solves the most instances, and is on the Pareto front for all thresholds and problems. Only for $\varepsilon = 0.05$ there exists a learned method that requires marginally fewer iterations or less time.

D.2. Distribution of solving times

In the previous results, we only report the mean performance across all problem instances to summarize the performance in a single number. Here, we additionally show the distribution of the performance across the 100 individual test instances. For each learned ADMM solver, we compare the distribution of relative objectives after $K = 10$ iterations in Figure 3. Overall, we can see that there is a large spread of relative objectives achieved after the iterations showing that the problems are vastly different in difficulty.

One observation is that the learned ADMM method can lead to slightly worse performance for individual instances for which the fixed baseline method works very well. This is due to the fact that the training aims to improve the mean performance across all instances (Arisaka and Li, 2023).

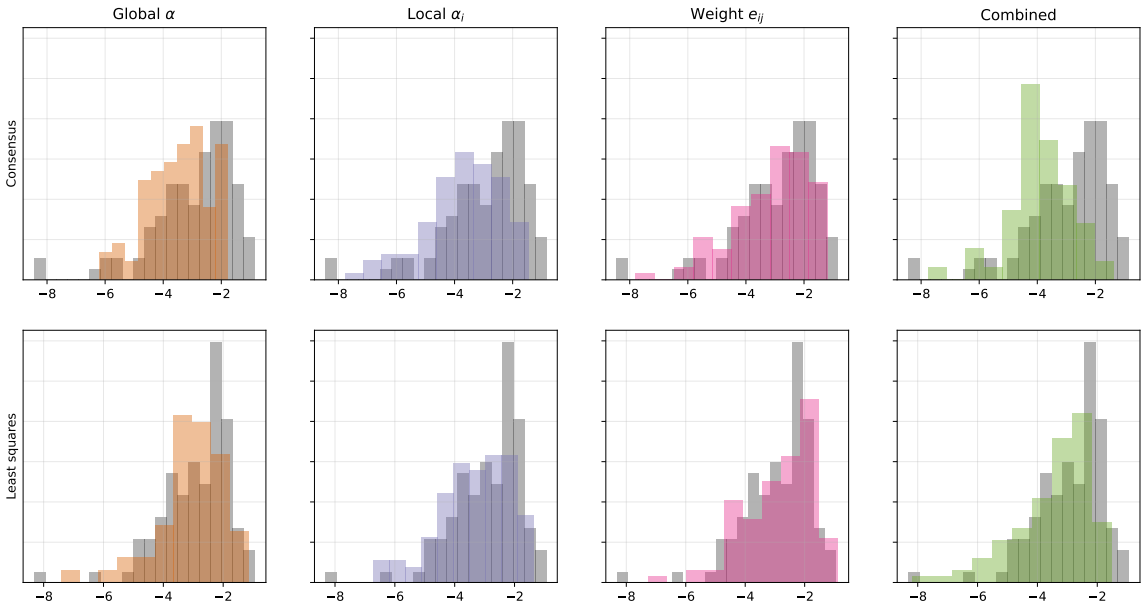

Figure 3: Pairwise comparison of the the distributions of normalized objective value across the test set between the fixed baseline method (gray) and the learned methods (colored). The x-axis shows the log-scaled normalized residual after 10 iterations.

Table 4: Results for OOD generalization with increasing number of nodes in the distributed system after $K = 10$ steps. For each size, we solve 20 problem instances and report the mean performance. All models are trained using problem instances with $m = 8$ nodes. Best method is **bold**, second best underlined.

	m=16		m=32		m=64		m=128	
Consensus	Error	Consensus	Error	Consensus	Error	Consensus	Error	Consensus
Fixed	22.92	21.26	18.04	17.24	16.90	16.51	20.07	19.88
Adaptive	25.01	24.00	21.37	20.89	20.56	20.36	24.58	24.48
Global α	18.91	18.16	16.11	15.82	16.30	16.15	19.19	19.13
Local α_i	15.90	15.74	13.16	13.04	13.74	13.69	16.37	16.34
Weight e_{ij}	19.63	17.90	17.99	17.29	16.21	15.88	19.76	19.59
Combined	**12.93**	**12.47**	**10.91**	**10.58**	**11.65**	**11.54**	**13.95**	**13.90**

D.3. Generalization behavior of learned methods

OOD generalization To examine the robustness and generalization ability of our learned methods, we evaluate their performance on larger graphs unseen during training, and compare the results with the behavior of the two baselines. For the out-of-distribution samples, we slightly change the data distribution by adjusting the edge weight parameter p in the random graph generation to keep the approximate neighborhood structure fixed. Given the initial value for $p = 0.4$ the expected number of neighbors in a graph with $m = 8$ nodes is 2.8. We adjust the edge probability for a graph with $m \in \{16, 32, 64, 128\}$ nodes to $p = 2.8/(m - 1)$. Thus, the expected degree of each node remains the same even when increasing the graph size. This is more realistic than scaling the degree of the nodes as well, as in practical applications, the degree often represents a physical communication limit, which does not increase when more neighbours are in the network.

The obtained results are shown in Table 4. Although the network is only trained on small graphs with $m = 8$ nodes, we observe that all learned methods generalize well and outperform the baselines even on the OOD data. The performance of all methods drops compared to the original data shown in Table 1, but does not continue to worsen when increasing the graph size further.

Additional ADMM iterations In order to show the performance of our method when increasing the number of iterations even further beyond the training regime, we showcase the performance for 100 iterations in Figure 4. Figure 2 showed the same results until 20 iterations are reached.

The results show that the learned methods continue to outperform the baselines, even far beyond the trained iterations. The only exception to this is the combined method in the least-squares problem, which slows down after 50 iterations. Particularly noteworthy is the strong performance of the method with learned communication matrix, as the learned matrix is used in all iterations and not only in the trained ones. However, using the learned communication matrix together with the learned step sizes performs best in the consensus example while achieving the worst in the least squares problem.

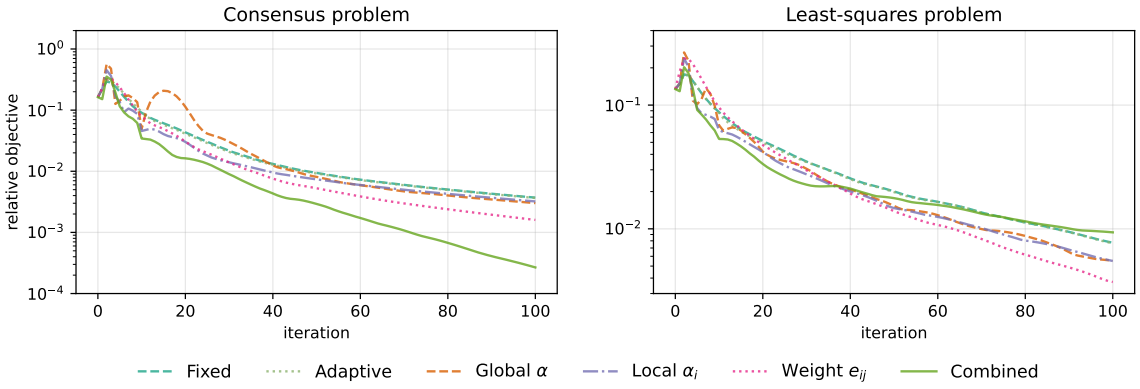

Figure 4: Semi-log plots of the mean relative objective over all 100 test instances in the two problem classes for 100 iterations. Note that this is far beyond the trained unrolling of $K = 10$ iterations.

Appendix E. Algorithms

E.1. Decentralized, distributed ADMM

In the following, we state the decentralized, distributed ADMM algorithm (Algorithm 1) from Makhdoumi and Ozdaglar (2017) adapted to our notation. As described in Section 3.2, each iteration updates all three iterates (x_i, y_i, λ_i) and includes two communication steps to pass information about the current iterates to the neighbors in $\mathcal{G}$.

E.2. Message-passing formulation of distributed ADMM

When it comes to defining a message-passing network that performs one iteration of Algorithm 1, it is critical to reformulate the update steps in such a way that they only require knowledge of the aggregation of incoming messages and not of all individual messages. We pointed this out in Section 4.1 and defined respective `message`, `aggregate`, and `update` functions that respect the MPNN framework. The two resulting message-passing steps are presented in Algorithm 2.2. To highlight the one-to-one correspondence, we compare it with the equivalent Algorithm 2.1, which is identical to a single iteration of the original distributed ADMM algorithm (see lines of 8-14 of Algorithm 1).

Equivalence between updates. For completeness, we will briefly show the equivalence between the original x-update step from (3) and the rewritten step in (6) here. Details for the y and λ update are omitted since the equivalence follows immediately from the definition of the messages.

Equation (3) (or line 8 of Algorithm 1) includes a sum over features from neighboring nodes that we need to revise. The first summand in this sum can be rewritten easily by the definition of the aggregated messages $\bar{\lambda}^k_{\to i}$:

$$\sum_{j \in \mathcal{N}(i) \cup \{i\}} (\lambda_j^k)^T (P_{ji} x_i) = (P_{ii} \lambda_i^k)^T x_i + \underbrace{\left(\sum_{j \in \mathcal{N}(i)} P_{ji}^T \lambda_j^k \right)^T}_{=: \bar{\lambda}^k_{\to i}} x_i \,, \tag{15}$$

Algorithm 1 Decentralized, distributed ADMM

Hyperparameters: $\alpha > 0$, communication matrix P ($= \mathcal{L}$ here), d_i degree of node i
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 $j \in \mathcal{N}(i)$
for $i = 1, \dots, m$ **do**
    Initialize y_i^0

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

end for
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 $j \in \mathcal{N}(i)$
        Update local solution candidate x:

$$x_i^{k+1} = \arg\min_{x_i} \left(f_i(x_i) + \sum_{j \in \mathcal{N}(i) \cup \{i\}} \left((\lambda_j^k)^T (P_{ji} x_i) + \frac{\alpha}{2} \| P_{ji}(x_i - x_i^k) + y_j^k \|_2^2 \right) \right) .$$

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

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

        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: $x^k = [\, x_i^k \,]_{i=1}^m$

Note that P_{ji} must be included in the message since weighting is no longer possible once all λ_j^k are aggregated. The second summand, the penalty term for consensus, splits into the following

$$\sum_{j \in \mathcal{N}(i) \cup \{i\}} \| P_{ji}(x_i - x_i^k) + y_j^k \|_2^2 = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \Bigg(\| P_{ji}(x_i - x_i^k) \|_2^2 + \underbrace{\| y_j^k \|_2^2 + 2(P_{ji}(x_i - x_i^k))^T y_j^k}_{\text{includes features from neighbors}} \Bigg) . \tag{16}$$

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

$$2 \cdot \sum_{j \in \mathcal{N}(i) \cup \{i\}} (P_{ji}(x_i - x_i^k))^T y_j^k = 2(x_i - x_i^k)^T \left(P_{ii}^T y_i^k + \underbrace{\sum_{j \in \mathcal{N}(i)} P_{ji}^T y_j^k}_{=: \bar{y}_{\to i}^k} \right), \tag{17}$$

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 involving x_i^k out in the optimization.

Note that this rule can be modified further if the edge weights e_{ij} are not fixed over the whole algorithm and are not accessible for node i. In this case, one must add (P_{ji}, P_{ji}^2) to the message.

MLP Integration. For step size learning (both global and node-level), we can predict and use a new step size in each iteration. To achieve this, we insert each MLP that predicts a new step size into the first node `update` function of each iteration, before the x variable is updated. In Algorithm 2.2, this is exactly between lines 5 and 6 of the pseudocode. The predicted step size is then used during an entire iteration.

Conversely, in the edge-level task, we predict the edge-weights e_{ij} only once and keep them fixed during the whole $K = 10$ unrolled iterations. The corresponding MLP is placed before the first iteration of Algorithm 2.2.

Algorithm 2.1 One iteration of decentralized, distributed ADMM

for $i \in \mathcal{V}$ **in parallel do**

Communicate y_i^k and λ_i^k to all neighbors $j \in \mathcal{N}(i)$

Update local solution candidate x:

$$x_i^{k+1} = \arg\min_{x_i} \Bigg(f_i(x_i) + \sum_{j \in \mathcal{N}(i) \cup \{i\}} \left((\lambda_j^k)^T (P_{ji} x_i) + \frac{\alpha}{2} \| P_{ji}(x_i - x_i^k) + y_j^k \|_2^2 \right) \Bigg) .$$

Communicate x_i^{k+1} to all neighbors $j \in \mathcal{N}(i)$

Update local deviation-variable y:

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

Update local dual-variable λ:

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

end for

Algorithm 2.2 One iteration of decentralized, distributed ADMM **as 2-block GNN**

for $(j, i) \in \mathcal{E}$ **in parallel do**

$m_{ji} = \left(P_{ji} \lambda_j^k, P_{ji} y_j^k \right)$ ▷ `message 1`

end for

for $i \in \mathcal{V}$ **in parallel do**

$\left(\bar{\lambda}_{\to i}^k, \bar{y}_{\to i}^k \right) = \sum_{(j,i) \in \mathcal{E}} m_{ji}$ ▷ `aggregation 1`

$v_i = \left(x_i^{k+1}, y_i^k, \lambda_i^k \right)$, where ▷ `update 1`

$$x_i^{k+1} = \arg\min_{x_i} \Bigg(f_i(x_i) + \left(P_{ii} \lambda_i^k + \bar{\lambda}_{\to i}^k + \alpha (P_{ii} y_i^k + \bar{y}_{\to i}^k) \right)^T x_i + \frac{\alpha}{2} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \| P_{ji}(x_i - x_i^k) \|_2^2 \Bigg) .$$

end for

for $(j, i) \in \mathcal{E}$ **in parallel do**

$m_{ji} = P_{ij} x_j^{k+1}$ ▷ `message 2`

end for

for $i \in \mathcal{V}$ **in parallel do**

$\bar{x}_{\to i}^{k+1} = \sum_{(j,i) \in \mathcal{E}} m_{ji}$ ▷ `aggregation 2`

$v_i = \left(x_i^{k+1}, y_i^{k+1}, \lambda_i^{k+1} \right)$, where ▷ `update 2`

$$y_i^{k+1} = \frac{1}{d_i + 1} \left(\bar{x}_{\to i}^{k+1} + P_{ii} x_i^{k+1} \right) ,$$
$$\lambda_i^{k+1} = \lambda_i^k + \alpha y_i^{k+1} .$$

end for