UPPSALA
UNIVERSITET

IT mDA 25 003
Degree project 30 credits
May 2025

Accelerating Active-set Solvers using Graph Neural Networks

Ella Johanna Schmidtobreick

Master's Program in Data Science

UPPSALA
UNIVERSITET

Accelerating Active-set Solvers using Graph Neural Networks

Ella Johanna Schmidtobreick

Abstract

This thesis explores the use of machine learning to accelerate the solution of Quadratic Programs (QPs), a widely used class of optimization problems with a quadratic objective and linear inequality constraints. Specifically, it investigates the integration of Graph Neural Networks (GNNs) into the active-set method, an iterative algorithm that identifies the subset of constraints active at the optimal solution.

Instead of performing all iterations of the active-set method, a GNN is trained to predict the active set directly, leveraging a graph-based representation of QPs to capture problem structure. This prediction enables warm-starting the solver, potentially reducing the number of iterations needed to find the solution, ideally to a single step.

Experimental results demonstrate that this approach can significantly reduce both iteration count and computational time while maintaining high prediction accuracy. The results highlight the potential of learning-to-optimize strategies in structured optimization problems and motivate further research on scalability and generalization across problem sizes.

Faculty of Science and Technology

Uppsala University, Uppsala

Supervisor: Paul Häusner, Daniel Arnström Subject reader: Jens Sjölund

Examiner: Filip Malmberg

Contents

Chapter 1

Introduction

A common class of optimization problems are Quadratic Programs (QPs), which appear in a range of applications, including robotics [1], control [2], and finance [3, 4]. These problems are typically solved using numerical optimization methods, which can be computationally expensive and therefore unsuitable for real-time applications. To overcome these limitations, recent research has explored integrating machine learning techniques in conventional solvers [5–8].

Extending this line of work, the thesis investigates how Graph Neural Networks (GNN), a graph-based machine learning technique, accelerate the active-set method for solving QPs on synthetic data. This technique leverages the underlying problem structure using graphs. Instead of computing each iteration precisely, the proposed approach predicts the active set, allowing the solver to omit multiple iterations and thereby significantly reduce computation time.

The main contributions of this thesis are therefore

- designing a GNN-based approach to accelerate QP solving by learning the active set.
- a comprehensive evaluation of the proposed method in terms of scalability to different input sizes and its effectiveness in reducing the number of iterations and overall solving time through various experiments.

The thesis begins by presenting related work and formulating concrete research questions. It then provides the necessary background on Quadratic Programs, active-set solvers and Graph Neural Networks. These foundations are used to develop a GNN-based approach for solving QPs, which is subsequently evaluated through experimental analysis.

Ethical considerations. As this thesis applies machine learning to a mathematical optimization problem using synthetic data, ethical considerations are not applicable and are therefore omitted.

1.1 Related work

This thesis draws on relevant work from both fundamental optimization methods and machine learning approaches. The first section explores various solution methods for Quadratic Programs (QPs), emphasizing why the active-set solver is the most suitable choice for this thesis. The second section reviews different learning-to-optimize approaches on active-set solvers using various machine learning techniques, providing a strong foundation for this work.

1.1.1 Numerical optimization methods for QPs

Commonly used methods for solving Quadratic Programs include interior-point methods [9, 10], active-set methods [11–14], gradient projection methods [15, 16] and first-order methods including operator-splitting methods [17–20].

Interior-point methods solve quadratic programs by applying Newton's method to a sequence of reformulated problems. By introducing barrier functions, the constrained optimization problem is transformed into an unconstrained problem while still ensuring feasibility. The method follows an interior path towards the optimal solution, adjusting step length and search direction to maintain feasibility while optimizing the objective function. Interior-point methods are particularly effective for large-scale problems but require expert knowledge to efficiently exploit the problem structure [9], [21].

Active-set solvers iteratively identify the inequality constraints of an optimization problem that hold with equality, called active constraints, to then transform a inequality-constraint problem into one with a reduced set of equality constraints. In each iteration, a constraint is added to or removed from the working set, approximating the true active set. These solvers are particularly effective for small- and medium-sized problems and at detecting problem infeasibility [21]. Given their ability to efficiently warm-start a series of closely related QPs, and thereby accelerate the solving process through predictions made by a graph neural network, this thesis focuses on active-set solvers [22].

Like the active-set method, the gradient projection method iteratively identifies the active set. The method follows the steepest gradient descent direction while ensuring feasibility by projecting the gradient onto the feasible region whenever a boundary is encountered. This allows the method to detect multiple constraints as active in each iteration, making it computationally efficient for large-scale problems. Due to the projections, this approach is particularly well-suited for problems with simple constraints, such as only variable bounds, where the feasible region forms a rectangular shape, allowing for efficient projections. Since the feasible region of general QPs is defined by more complex constraint boundaries, this method is less efficient [21].

One well-known class of first-order methods is operator-splitting. Common

techniques include Douglas-Rachford splitting [17, 18] and the Alternating Direction Method of Multipliers (ADMM) [19]. These methods decompose the given quadratic program into structured subproblems, solving them iteratively while updating variables to approach the optimal solution. First-order methods are computationally inexpensive and well-suited for large-scale problems. However, they are highly sensitive to tuning parameters, such as step length, and typically do not detect primal or dual infeasibility. Additionally, the solution is only of limited accuracy, making this method unsuitable for high-precision optimization tasks [20].

1.1.2 Learning-to-optimize

An alternative approach to traditional methods is to apply machine learning in the optimization process. This approach, known as learning-to-optimize [23–27] combines data with classical algorithms to improve solution strategies. Different targets can be learned, such as predicting the primal and dual solutions directly, as in [28]. Since this is computationally more expensive and less robust than predicting the active constraints, this thesis focuses on learning the active set at the optimum, which is then used to warm-start the method.

Learning-to-optimize can be applied to various optimization methods, such as operator-splitting techniques, which benefit from this approach. In [5], a neural network is used to warm-start the Douglas-Rachford splitting method, resulting in an effective end-to-end framework. Similarly, in [6], reinforcement learning is used to accelerate the first-order solver OSQP, leading to significant performance improvements.

In addition to these methods, several studies have also applied learning-to-optimize to the active-set method. Klaučo et al. [7]compares the performance of k-nearest neighbors and classification trees to warm-start active set methods, where k-nearest neighbors proved superior in accelerating the primal active-set solver for the classification problem. Here, the underlying problem is framed as a regression task, to not only identify the active set but also quantify the model's confidence regarding the different constraints.

Chen et al. [8] investigates the application of a neural network to QPs. The neural network's output serves as a warm-start for the primal active-set method, leading to a twofold reduction in solution time, even for large-scale systems.

In the paper of [22], the objective is to learn the active constraints using a Transformer architecture. The active set is learned not only to warm-start the problem but also to provide a good initial guess for the optimal solution. The work compares several solvers, including DAQP, the dual active-set solver that is the focus of this thesis, and evaluates the performance of various machine learning models, with the Transformer architecture proving to be particularly effective.

Most existing machine learning approaches applied to active-set methods

ignore the specific structure inherent in optimization problems. To address this limitation and explore the potential benefits of incorporating the problem structure into machine learning models, this thesis investigates the application of Graph Neural Networks (GNNs) for solving quadratic optimization problems. GNNs offer two key advantages for capturing optimization problems: they are invariant to node permutations, which corresponds to reordering constraints, and they have the ability to effectively exploit sparsity, as discussed in [29]. Chen et al. [30] presents the transformation of QPs, represented as matrices, into graphs, focusing on whether GNNs can efficiently map the problem to its optimal solution. While Cappart et al. [29] examines the reasoning of GNNs for solving combinatorial optimization problems, and Chen et al. [30] emphasizes theoretical results over extensive training and generalization, both lack experimental validation, which will be addressed in this thesis.

1.2 Research questions

The goal of this thesis is to accelerate quadratic programming solving by effectively capturing the structure of the optimization problem. Specifically, it aims to predict the active set of constraints using a GNN, enabling to warm-start the active-set solver and potentially reducing the number of required iterations. The research questions addressed in this thesis are given in the following.

- How accurately can a GNN predict the active constraints of a QP?
- How does incorporating the structure of the optimization problem affect predictive performance?
- How does the predictive model affect optimization performance?

This approach bridges the gap between optimization and machine learning by effectively capturing the structure of the underlying optimization problem, while leveraging the strengths of machine learning through the use of GNNs.

Chapter 2

Background

This chapter presents the theoretical background necessary to understand the learned warm-starting strategy and to evaluate the experimental results. It first introduces quadratic programming, followed by a detailed discussion of active-set solvers, which form a state-of-the-art class of QP solvers. Finally, Graph Neural Networks are introduced, which operate on graph-structured representations of QPs.

2.1 Quadratic programming

The following optimization problem is referred to as a **Quadratic Program (QP)** and is given by

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & J(x) \coloneqq \frac{1}{2} x^T H x + f^T x \\ \text{subject to} \quad & Ax \leq b. \end{aligned} \tag{2.1}$$

The objective function $J : \mathbb{R}^n \to \mathbb{R}$ is composed of a quadratic term defined by the matrix $H \in \mathbb{R}^{n \times n}$ and a linear term $f \in \mathbb{R}^n$. The feasible solution space is given by a polyhedron defined by m linear constraints, which are represented in matrix form by $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. The problem is **convex** if H is positive semi-definite, guaranteeing the existence of a global minimum. If H is positive definite, this minimum is unique, whereas for a positive semi-definite H multiple optimal solutions might exist [31].

The resources required to solve a QP depend on the objective function and number of constraints. Convex QPs are comparable in difficulty to linear problems and can be solved efficiently, while nonconvex QPs, defined by an indefinite matrix H, are more challenging due to potential stationary points and local minima [21].

Example 2.1.1. Consider the following quadratic program

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & x_1^2 + x_2^2 + x_1x_2 - 4x_1 - 8x_2 \\ \text{subject to} \quad & x_1 + \ x_2 \leq 3 \\ & -x_1 + 2x_2 \leq 0. \end{aligned} \tag{2.2}$$

This problem can be written in the standard QP form 2.1 with

$$H = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad f = \begin{pmatrix} -4 \\ -8 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 1 \\ -1 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} 3 \\ 0 \end{pmatrix}.$$

Figure 2.1 illustrates the QP, showing the level sets of the objective function as ellipsoids and highlighting the feasible region, defined by the inequality constraints. Additionally, the unconstrained optimal solution $-H^{-1}f = \begin{pmatrix} 4 & 0 \end{pmatrix}^T$ and the constrained minimizer $x^* = \begin{pmatrix} 2 & 1 \end{pmatrix}^T$ are indicated in the figure.

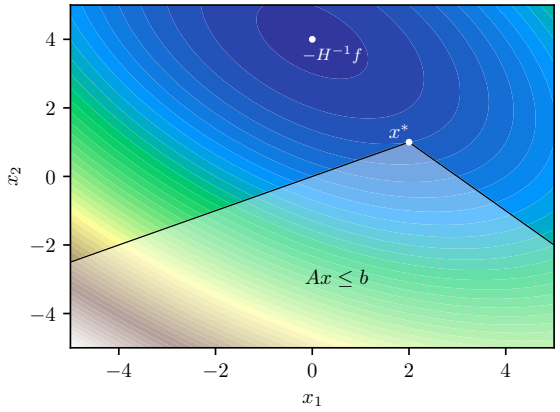

Figure 2.1: Visualization of the two-dimensional QP 2.2 in Example 2.1.1 with the constraint optimum x^*. The objective function is illustrated by the colored background and the level curves (warmer colors correspond to higher function values) and the unconstrained minimum is marked by $-H^{-1}f$. The gray area represents the feasible region defined by the constraints.

Definition 2.1.2. Let a QP as defined in Equation (2.1) be given.

(a) If there exists a variable $x \in \mathbb{R}^n$ such that $J(x) < \infty$ and $Ax \leq b$, the QP is **feasible**, such that at least one solution exists. Otherwise, if no such x

exists, the problem is **infeasible**. Infeasibility results from two different cases.

(b) If there exists no point that satisfies the constraints $Ax \leq b$, such that the feasible set is empty $\{x \in \mathbb{R}^n \mid Ax \leq b\} = \emptyset$, the problem is called **primal infeasible**.

(c) If the matrix H is singular, it can occur that there exist primal feasible points where the objective function has no lower bound

$$\inf_{x \in \mathbb{R}^n} J(x) = -\infty.$$

Since no minimum exists, the problem is called **unbounded** or **dual infeasible**.

Since a closed-form solution for QPs, as defined in 2.1, generally does not exist, iterative methods are used to solve the problem [32]. The result of a QP can be obtained in a finite number of steps by either finding an optimal solution or proving infeasibility [21].

A useful tool for finding the optimal solution is **duality**, which states that every QP of the form 2.1, referred to as the **primal problem**, has a corresponding dual problem. The optimal values of both problems are related, such that the primal solution can be found by solving the dual problem, and conversely. Consequently, solving the dual problem provides an effective alternative to directly solving the primal problem [33].

As stated in [21], the dual problem is constructed by applying the **Lagrangian function**

$$L(x, \lambda) \coloneqq J(x, \lambda) - \lambda^T c(x) \tag{2.3}$$

to problem 2.1, where the function c represents the constraints and is defined as

$$0 \leq c(x) \coloneqq b - Ax. \tag{2.4}$$

This results in the following expression for the Lagrangian of problem 2.1

$$L(x, \lambda) = \frac{1}{2} x^T H x + f^T x + \lambda^T (Ax - b), \tag{2.5}$$

where $\lambda \in \mathbb{R}^m$ represents the dual variable.

The objective function of the dual problem is then obtained by taking the infimum of the Lagrangian function with respect to x, resulting in the **dual problem** formulation

$$\begin{aligned} \underset{\lambda \in \mathbb{R}^m}{\text{maximize}} \quad & \inf_{x \in \mathbb{R}^n} \frac{1}{2} x^T H x + f^T x + \lambda^T (Ax - b) \\ \text{subject to} \quad & \lambda \geq 0. \end{aligned} \tag{2.6}$$

It is important to note that the Hessian matrix of the dual problem is not necessarily positive definite, and thus an unconstrained optimum of the dual problem does not always exist.

If an optimal solution $x^* \in \mathbb{R}^n$ exists for the primal problem, it satisfies the **Karush-Kuhn-Tucker (KKT) conditions** defined by

$$Hx^* + A^T\lambda = -f \tag{2.7a}$$
$$Ax^* \leq b \tag{2.7b}$$
$$\lambda \geq 0 \tag{2.7c}$$
$$([b]_i - [A]_i x^*)[\lambda]_i = 0, \quad \forall\, i \in \mathbb{N}_m \tag{2.7d}$$

for some dual variable $\lambda \in \mathbb{R}^m$. The notation $[\,\cdot\,]_i$ refers to the i-th row of a matrix. Equation 2.7a represents the **stationarity condition**, ensuring that the optimal solution minimizes the objective function J. Equations 2.7b and 2.7c correspond to the constraints for the primal and dual problem and are referred to as **primal** and **dual feasibility**. Finally, the **complementary slackness condition** 2.7d defines the relationship between the primal variable x^* and the dual variable λ. It states that either the inequality constraint holds with equality, or the corresponding component of λ must be zero, or both [21]. This condition is the only non-linear condition in 2.7, making the QP nontrivial to solve [31].

As proven in [34], an optimal solution λ^* to the dual problem 2.6 satisfies the same KKT conditions as an optimal solution x^* to the primal problem 2.1, thereby demonstrating equivalence between solving either problem. The corresponding value of the other optimal solution is determined by solving the stationarity condition 2.7a for the respective variable. Specifically, when solving the dual problem, the primal solution x^* is given by

$$x^* = -H^{-1}(A^T\lambda + f). \tag{2.8}$$

The conditions in 2.7 are necessary for primal and dual optimal points in general QPs, while in the convex case they are also sufficient [33]. To obtain an optimal solution, the KKT system

$$\begin{pmatrix} H & A^T \\ A^T & 0 \end{pmatrix} \begin{pmatrix} x^* \\ \lambda \end{pmatrix} = \begin{pmatrix} -f \\ b \end{pmatrix}, \tag{2.9}$$

is solved, incorporating the stationarity condition 2.7a and primal feasibility 2.7b. The solution for this system provides candidate values for x^* and λ^*, which must then be verified against the dual feasibility condition 2.7c and the complementary slackness condition 2.7d, as these are not explicitly enforced within the linear system.

One commonly used method for solving QPs is the active-set method. It ensures that the KKT conditions hold at the optimum and is presented in more detail in the next section.

2.2 Active-set method

An active-set method aims to identify the active set at an optimal solution x^* in order to simplify the system.

Definition 2.2.1. Let $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}$ and $x \in \mathbb{R}^n$.

(a) An inequality constraint $[A]_i x \leq [b]_i$ is **active** at $\hat{x} \in \mathbb{R}^n$ if it holds with equality $[A]_i \hat{x} = [b]_i$ at that point.

(b) The **active set** of a point x is the set of all inequality constraints that are active at point x. This set is denoted by $\mathcal{A}(x) \coloneqq \{i \in \mathbb{N}_m \mid [A]_i x \leq [b]_i\}$. For the active set at an optimal solution x^*, the notation $\mathcal{A}^*$ is used instead of $\mathcal{A}(x^*)$.

Example 2.2.2. In Example 2.1.1 both constraints are active and therefore hold with equality at the optimum. The active set at x^* is given by $\mathcal{A}^* = \{1, 2\}$. If an additional constraint,

$$-3x_1 + x_2 \leq 10$$

is introduced, the optimal solution remains unchanged, since this new constraint is inactive at the optimum, as illustrated in Figure 2.2. In this case, the active set at the optimum remains $\mathcal{A}^* = \{1, 2\}$, even though the problem includes three constraints.

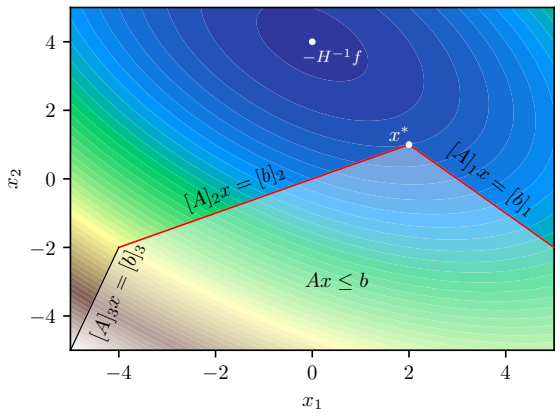

Figure 2.2: Visualization of Example 2.2 with one additional inactive constraint ($[A]_3 x = [b]_3$). The active constraints are highlighted in red.

The following Lemma, presented in [31], states that a QP with inequality constraints can be reduced to a problem with only equality constraints if the active set is known.

Lemma 2.2.3. Let x^* be an optimal solution of Problem 2.1. Then x^* is also an optimal solution for the reduced equality constrained problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & J(x) \coloneqq \frac{1}{2} x^T H x + f^T x \\ \text{subject to} \quad & [A]_i x = [b]_i, \quad \forall i \in \mathcal{A}^*. \end{aligned} \tag{2.10}$$

Proof. Since all conditions in 2.7 must hold for an optimal solution, it follows that

$$[\lambda]_i = 0, \quad \forall i \in \mathbb{N}_m \setminus \mathcal{A}^* \tag{2.11}$$

by condition 2.7d. Substituting this into 2.7a, the stationarity condition becomes

$$Hx^* + [A]_{\mathcal{A}^*}^T [\lambda]_{\mathcal{A}^*} = -f. \tag{2.12}$$

Further, the primal feasibility condition is reduced to

$$[A]_{\mathcal{A}^*} x^* = [b]_{\mathcal{A}^*}. \tag{2.13}$$

Together, this leads to the reduced KKT system

$$\begin{pmatrix} H & [A]_{\mathcal{A}^*}^T \\ [A]_{\mathcal{A}^*}^T & 0 \end{pmatrix} \begin{pmatrix} x^* \\ [\lambda]_{\mathcal{A}^*} \end{pmatrix} = \begin{pmatrix} -f \\ [b]_{\mathcal{A}^*} \end{pmatrix}, \tag{2.14}$$

which is equivalent to the KKT system for 2.10. □

By Lemma 2.2.3, knowing the active set reduces the QP with inequality constraints to a system of linear equations. By this result, the fundamental idea of the active-set method is to identify the active set $\mathcal{A}^*$. To achieve this, the working set $\mathcal{W}$ serves as approximation of the active set and is updated iteratively. At each iteration, one constraint is either added to or removed from $\mathcal{W}$ until the active set of an optimal solution x^* is found and the linear system can be solved in a single iteration. To speed up convergence, the algorithm can be warm-started by beginning with a non-empty working set, which allows for faster identification of the optimal active set. During the iterative process, the complementary slackness condition 2.7d is always guaranteed, while the other conditions are checked subsequently [31].

In the literature, multiple active-set solvers are presented ([11–13]), which can be categorized into primal and dual methods, based on whether they operate on the primal or dual problem. While solvers within each category are mathematically identical in terms of the sequence of points generated, they differ in their numerical implementation [35]. Since warm-starting a dual active-set solver is straightforward, this thesis focuses on dual methods [14, 36, 37].

Primal active-set methods act on the primal QP 2.1. Through all iterations, primal feasibility 2.7b and the complementary slackness condition 2.7d are ensured, while dual feasibility 2.7c and the stationarity condition 2.7a are only

satisfied in the last iteration. Since the primal feasibility is maintained in all iterations, early stopping with an approximate solution is possible. However, warm-starting a primal active-set solver can be challenging due to finding a feasible starting iterate [31].

Example 2.2.4. Example 2.1.1 is used to illustrate the primal active-set method. The iterative steps of the algorithm are shown in Figure 2.3 and are given by

$$x_0 = \begin{pmatrix} 0 \\ -3 \end{pmatrix}, \quad x_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

with x_2 reaching the constrained optimal solution.

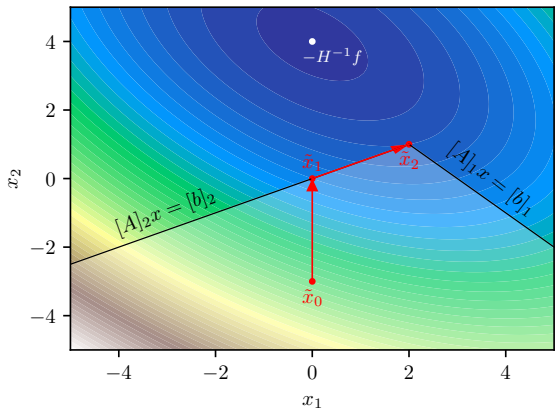

Figure 2.3: Visualization of the steps taken by the primal active-set algorithm when solving problem 2.2.

As warm-starting plays a crucial role in accelerating the active-set methods, the dual active-set solver proposed by [37] was selected for this thesis. The main idea of the dual active-set solver is to apply the primal active-set solver stated in [31] while operating on the dual problem defined in 2.6. If H is positive definite ($H \succ 0$), the dual problem can be reformulated as

$$\begin{aligned} \underset{\lambda \in \mathbb{R}^m}{\text{minimize}} \quad & \frac{1}{2}\lambda^T A H^{-1} A^T \lambda + (b + AH^{-1}f)^T \lambda \\ \text{subject to} \quad & \lambda \geq 0, \end{aligned} \tag{2.15}$$

by solving 2.7a for x, i.e. $x = -H^{-1}(f + A^T\lambda)$, and substituting it into the

problem definition 2.6. Similar to [11], this can be rewritten as

$$\min_{\lambda \geq 0} \ \frac{1}{2}\lambda^T M M^T \lambda + d^T \lambda, \tag{2.16}$$

with

$$M \coloneqq AR^{-1}, \quad v \coloneqq R^{-T} f, \quad d \coloneqq b + Mv, \tag{2.17}$$

where R is an upper triangular Cholesky factor of H, such that $H = R^T R$. It is worth noting that the set of active constraints in the primal problem corresponds to the inactive constraints in the dual problem.

Example 2.2.5. The dual problem corresponding to Example 2.1.1 is given by

$$\underset{\lambda \geq 0}{\text{minimize}} \quad \frac{1}{3}\lambda_1^2 + \frac{7}{3}\lambda_2^2 + \frac{1}{3}\lambda_1\lambda_2 - \lambda_1 - 8\lambda_2. \tag{2.18}$$

This problem can be expressed in the standard dual QP form 2.16 with

$$MM^T = \begin{pmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{14}{3} \end{pmatrix} \quad \text{and} \quad d = \begin{pmatrix} -1 \\ -8 \end{pmatrix}. \tag{2.19}$$

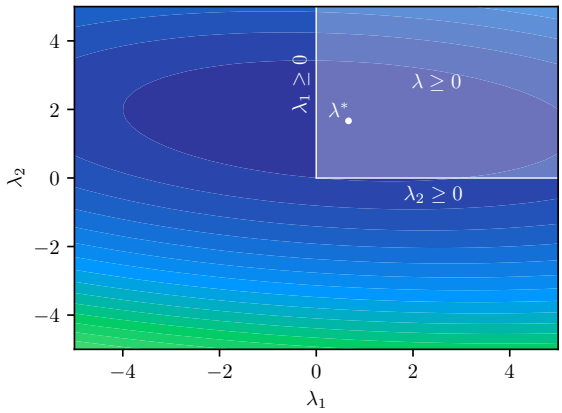

Figure 2.4: Visualization of the dual problem 2.18.

Figure 2.4 shows the level sets of the dual objective function as ellipsoids. Additionally, the feasible region given by $\lambda \geq 0$ is highlighted. In this example, the constrained and unconstrained solutions are intersecting and are denoted by

$$\lambda^* = \begin{pmatrix} \frac{2}{3} \\ \frac{5}{3} \end{pmatrix}.$$

To apply the principles of the primal algorithm to the dual algorithm, the dual active-set solver guarantees dual feasibility 2.7c in each step, which simplifies the constraints to $\lambda \geq 0$. In each iteration, the complementary slackness condition 2.7d is preserved by the working set $\mathcal{W}$, which contains the active constraints, while the other components of λ are set to zero. Consequently, in the dual active-set method, the stationarity condition 2.7a and primal feasibility 2.7b are not satisfied at every step.

Example 2.2.6. Example 2.1.1 is used to illustrate the dual active-set method as described in detail in Algorithm 1. The steps of the algorithm are illustrated in Figure 2.5, and have the corresponding values

$$\lambda_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \lambda_1 = \begin{pmatrix} 0 \\ \frac{12}{7} \end{pmatrix}, \quad \lambda_2 = \begin{pmatrix} \frac{2}{3} \\ \frac{5}{3} \end{pmatrix}$$

with λ_2 being the optimal solution.

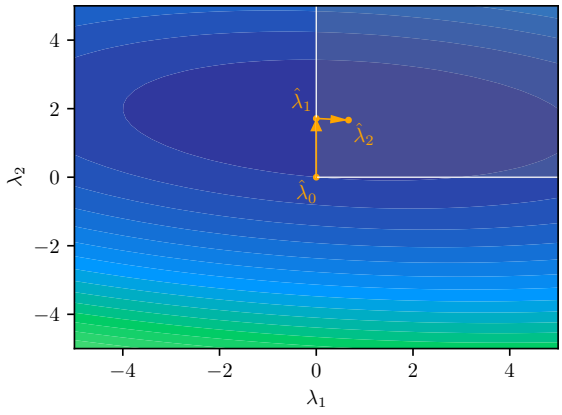

Figure 2.5: Visualization of the steps taken by the dual active-set algorithm 1 when solving problem 2.18 illustrated on the dual objective function.

To compare the steps taken by the primal and dual active-set algorithms, both paths are depicted in Figure 2.6. In the primal active-set algorithm, the process begins from a randomly selected point within the feasible region and iteratively moves outward toward the unconstrained optimal solution, stopping at the boundary where the constrained optimal solution is found. In contrast, the dual active-set solver starts from the unconstrained optimal solution and moves away from it until the feasible region is reached, also converging to the constrained optimal solution. It is evident that primal feasibility is violated in all steps of the dual algorithm except the final one, preventing early stopping.

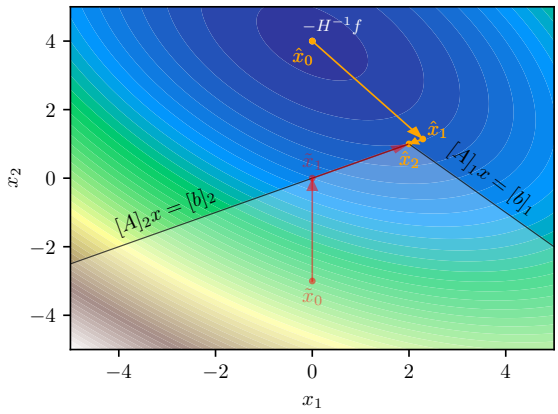

Figure 2.6: Visualization of the steps taken by the dual active-set algorithm 1 when solving problem 2.18 illustrated on the primal objective function.

The dual active-set solver [37] is presented in Algorithm 1 and explained step by step in the following. The algorithm takes as input the matrices M, d, v, R^{-1}, which define the dual representation of a QP, along with an initial working set $\mathcal{W}_0$ and an initial dual guess λ_0. The output consists of the optimal primal and dual solutions, as well as the corresponding working set at the optimal point.

To improve the readability of the algorithm, some notation is introduced in advance. The expression $M_{\mathcal{W}}$ and $d_{\mathcal{W}}$ denote the matrix M and vector d reduced to the rows indexed by the working set $\mathcal{W}$. The complement of the working set, denoted by $\overline{\mathcal{W}}$, corresponds to the constraints for which the associated components of the dual variable λ are fixed to zero. Thus, $M_{\overline{\mathcal{W}}}$ and $d_{\overline{\mathcal{W}}}$ refer to the matrix and vector reduced to the rows indexed by $\overline{\mathcal{W}}$.

In each iteration, the idea is to solve the equality constrained dual subproblem given by the current working set $\mathcal{W}$

$$\begin{aligned} \underset{\lambda \in \mathbb{R}^m}{\text{minimize}} \quad & \frac{1}{2}\lambda^T M M^T \lambda + d^T \lambda \\ \text{subject to} \quad & [\lambda]_i = 0, \quad \forall\, i \in \overline{\mathcal{W}}. \end{aligned} \tag{2.20}$$

The minimizer is an approximation of the optimal value, derived by considering only the components included in the working set, denoted by λ^*. By the multiplication with λ in both terms of the objective function, the rows corresponding to the indices not in the working set are effectively nullified. This allows for a reduction of M, d, and λ to only the non-zero components, while the components of λ corresponding to the indices outside the working set are fixed to zero.

Algorithm 1: Dual active-set method for solving 2.16 as stated in [37]

Input: $M, d, v, R^{-1}, \mathcal{W}_0, \lambda_0$
Output: $x^*, \lambda^*, \mathcal{A}^*$

1 **while** *true* **do**
2 **if** $M_{\mathcal{W}} M_{\mathcal{W}}^T$ is nonsingular **then**
3 $\lambda_{\mathcal{W}}^* \leftarrow$ solution to $M_{\mathcal{W}} M_{\mathcal{W}}^T \lambda_{\mathcal{W}}^* = -d_{\mathcal{W}}$
4 **if** $\lambda^* \geq 0$ **then** // λ^* dual feasible
5 $\mu_{\overline{\mathcal{W}}} \leftarrow M_{\overline{\mathcal{W}}} M_{\mathcal{W}}^T \lambda_{\mathcal{W}}^* + d_{\overline{\mathcal{W}}}$
6 $\lambda \leftarrow \lambda^*$
7 **if** $\mu \geq 0$ **then** // x^* primal feasible
8 optimum found **go to** step 19 // optimal solution found
9 **else** // x^* not primal feasible
10 $j \leftarrow \operatorname{argmin}_{i \in \overline{\mathcal{W}}} [\mu]_i$
11 $\mathcal{W} \leftarrow \mathcal{W} \cup \{j\}$
12 **else** // λ^* not dual feasible
13 $p \leftarrow \lambda^* - \lambda$
14 $\mathcal{B} \leftarrow \{i \in \mathcal{W} \mid [\lambda^*]_i < 0\}$
15 $[\lambda, \mathcal{W}] \leftarrow$ FIXCOMPONENT$(\lambda, \mathcal{W}, \mathcal{B}, p)$
16 **else** $M_{\mathcal{W}} M_{\mathcal{W}}^T$ singular
17 $p_{\mathcal{W}} \leftarrow$ solution to $M_{\mathcal{W}} M_{\mathcal{W}}^T p_{\mathcal{W}} = 0, \quad p^T d < 0$
18 $\mathcal{B} \leftarrow \{i \in \mathcal{W} \mid [p]_i < 0\}$
19 $[\lambda, \mathcal{W}] \leftarrow$ FIXCOMPONENT$(\lambda, \mathcal{W}, \mathcal{B}, p)$
20 **return** $x^* \leftarrow -R^{-1}(M_{\mathcal{W}}^T \lambda_{\mathcal{W}}^* + v)$, λ^*, $\mathcal{W}$

21 **procedure** FIXCOMPONENT$(\lambda, \mathcal{W}, \mathcal{B}, p)$
22 $j \leftarrow \operatorname{argmin}_{i \in \mathcal{B}} -\frac{[\lambda]_i}{[p]_i}$
23 $\mathcal{W} \leftarrow \mathcal{W} \backslash \{j\}$
24 $\lambda \leftarrow \lambda - \left(\frac{[\lambda]_j}{[p]_j}\right) p$

This simplification reduces the problem to

$$\min_{\lambda_{\mathcal{W}} \in \mathbb{R}^{|\mathcal{W}|}} \frac{1}{2} \lambda_{\mathcal{W}}^T M_{\mathcal{W}} M_{\mathcal{W}}^T \lambda_{\mathcal{W}} + d_{\mathcal{W}}^T \lambda_{\mathcal{W}}, \quad \lambda_{\overline{\mathcal{W}}} = 0. \tag{2.21}$$

To solve this efficiently, a matrix factorization of the form

$$M_{\mathcal{W}} M_{\mathcal{W}}^T = L D L^T, \tag{2.22}$$

where L is a lower unit triangular matrix and D is a diagonal matrix, can be used [37].

In the following, the individual steps in Algorithm 1 are explained in detail.

(Step 2-3) If $M_{\mathcal{W}} M_{\mathcal{W}}^T$ is nonsingular, the subproblem has a unique solution. The solution of 2.21 is therefore assigned to the rows indicated by the working set of λ^*, and the other rows are fixed to zero.

(Step 4-6) In this step, the algorithm checks if the dual feasibility holds for the found solution. If the dual feasibility holds, we set λ to the value of λ^* and calculate the primal feasibility.

(Step 7-8) If the system is also primal feasible, all KKT conditions are satisfied and the optimal solution is found. The algorithm terminates by calculating the optimal x^*.

(Step 9-11) If the preliminary solution λ^* is primal infeasible, at least one constraint is still violated. The first violated constraint corresponds to the most negative component of μ, which is then added to the working set $\mathcal{W}$.

(Step 12-15) If the potential solution λ^* is not dual feasible, at least one constraint in $\mathcal{W}$ is not active at the optimum. A line search in the direction of the calculated λ^* is performed. To guarantee dual feasibility, the constraint violated first will be removed from the working set $\mathcal{W}$ using FixComponent.

(Step 16-19) If the matrix $M_{\mathcal{W}} M_{\mathcal{W}}^T$ is singular, no bounded solution exists. Therefore, the objective function has no lower limit such that it can become arbitrarily small by moving in the direction p, which satisfies

$$p_{\overline{\mathcal{W}}} = 0, \qquad M_{\mathcal{W}} M_{\mathcal{W}}^T p_{\mathcal{W}} = 0, \qquad p^T d < 0. \tag{2.23}$$

The optimization is performed by the line search in direction p. If the QP is feasible, at least one of the negative components of λ will become zero. The first of these components to become zero is getting removed from the working set $\mathcal{W}$ fixing the corresponding component of λ to zero. This process is captured in the procedure FixComponent, which also calculates the necessary update for the variable λ.

In the case of primal infeasibility in the stated QP, this will be detected in Step 18 by $\mathcal{B} = \emptyset$, as shown in [37]. The procedure is repeated with an updated working set until the algorithm converges after a finite number of steps, as discussed in Section 3 of [14].

If no additional information is provided, the initial working set is defined to be the empty set $\mathcal{W}_0 = \emptyset$, this process is known as **cold-starting**. In contrast, a system can also be **warm-started** by providing a non-empty set as initial working set $\mathcal{W}_0 \neq \emptyset$ [7], which can significantly reduce the number of iterations needed to reach the optimum [38]. However, identifying the active set at the optimum for warm-starting is non-trivial, as the optimal solution x^* must be known.

Warm-starting a primal active-set solver can be challenging due to the difficulty of identifying a feasible working set at each iteration. In contrast, warm-starting the dual active-set method is simpler, since a dual feasible working set is generally easier to identify, as indicated by condition 2.7c [31]. Consequently, the experiments presented in Section 4 are based on the dual active-set method.

2.3 Graph neural networks

Graph neural networks (GNNs) is a class of neural networks that operates on graphs. To apply this method to solve a QP, the relating problem 2.1 will be represented as a graph.

Definition 2.3.1. (a) A **graph** is given by the ordered pair $G = (V, E)$ with V representing the set of **vertices** and $E \subset \{(s,t) \mid s,t \in V\}$ representing the set of **edges**. The pairs in E are ordered and we denote e_{st} as the directed edge connecting from s to t.

(b) The graph is called **undirected**, if $\forall\ (s,t) \in E : (t,s) \in E$, otherwise we call it **directed**.

(c) A **feature vector** is a mapping that assigns attributes to vertices and edges in a graph. The feature vector for a vertex $s \in V$ is represented by $x_s \in \mathbb{R}^{d_v}$, where d_v is the dimension of the vertex features. The feature vector for an edge $e_{st} \in E$ is denoted as $z_{st} \in \mathbb{R}^{d_e}$, where d_e is the dimension of the edge features.

(d) The **neighborhood** $\mathcal{N}_s$ of a vertex s is given by all vertices directly connected to it. It is denoted by $\mathcal{N}_s = \{t \mid (t,s) \in E \vee (s,t) \in E\}$.

Graph neural networks (GNNs) were first introduced by [39] and [40]. GNNs are parameterized models that can be applied to various learning frameworks, including supervised, semi-supervised, unsupervised, and reinforcement learning, to learn representations of edges, nodes, or entire graphs. These networks are capable of addressing both regression and classification tasks, with applications spanning diverse domains such as text classification [41], modeling physical systems [42, 43], and biochemical challenges like protein interface prediction [44, 45]. For further applications and detailed discussions, refer to [46–49].

The following definition is based on the message-passing GNN in [50]. A **graph neural network** is a deep learning architecture, in which a neural network operates on graphs. The model consists of multiple **GNN layers**, where vertex and edge features are updated in each layer. The update is performed by the permutation-invariant aggregation function $\bigoplus$, which acts on the neighborhood of each vertex, and the learnable functions ϕ and ψ. Most commonly, the graph structure remains unchanged throughout the process.

The update of layer l begins with computing the edges features for the subsequent layer $l+1$ using the parametrized message function $\phi_{\theta_z^{(l)}}$

$$z_{ts}^{(l+1)} \coloneqq \phi_{\theta_z^{(l)}}\left(z_{ts}^{(l)}, x_t^{(l)}, x_s^{(l)}\right), \tag{2.24}$$

where the output is referred to as **message**. In the next step, all incoming messages to a vertex s are aggregated by applying the aggregation function $\bigoplus$ over the neighborhood $\mathcal{N}_s$

$$m_s^{(l+1)} \coloneqq \bigoplus_{t \in \mathcal{N}_s} z_{ts}^{(l+1)}. \tag{2.25}$$

Common choices for $\bigoplus$ include summation, maximization, and averaging. The outcome $m_s^{(l+1)}$ serves as input for the vertex feature update for layer $l+1$ by the function $\psi_{\theta_x^{(l)}}$

$$x_s^{(l+1)} \coloneqq \psi_{\theta_x^{(l)}} \left(x_s^{(l)}, m_s^{(l+1)} \right). \tag{2.26}$$

The update functions ϕ and ψ are parameterized and learnable [48, 51].

The advantage of learning on graphs rather than on matrices is that graph objects are permutation equivariant [52], such that the order of constraints does not affect the model. Additionally, GNNs can operate on graphs of varying sizes, making them easily adaptable to different QPs. This fexibility allows the model to be applied to a wide range of problems [30].

Chapter 3

Methods

This chapter presents the complete pipeline for solving quadratic programs using a GNN-based approach. It begins with data generation (Section 3.1) and graph representation (Section 3.2), continues with the mapping learned by the GNN (Section 3.3), and concludes with the dual active-set solver (Section 3.4) and the final model architecture (Section 3.5).

All methods are implemented in Python. The Graph Neural Network is implemented using the PyTorch Geometric library[1], which builds on the PyTorch framework[2] for deep learning.

3.1 Problem generation

Since many applications require solving a sequence of similar QPs where only specific parameters change between iterations, the synthetic data used in the experiments is designed to capture this behavior.

The series of QPs is parametrized by $\theta \in \mathbb{R}^p$, which affects only the linear term in the objective function f and the right-hand side of the constraints b such that each QP is given by

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2} x^T H x + f(\theta)^T x \\ \text{subject to} \quad & Ax \leq b(\theta). \end{aligned} \tag{3.1}$$

As a result, the overall problem structure remains unchanged, even though different instances are generated.

The synthetic data used for the experiments consists of 5000 QPs. To ensure that the matrix H is symmetric and positive semi-definite, it is generated using a

[1] https://pytorch-geometric.readthedocs.io/en/latest/
[2] https://docs.pytorch.org/docs/stable/index.html

matrix $M \in \mathbb{R}^{n \times n}$ with $H = MM^T$. The right-hand side of the constraints and the linear term in the objective function are defined by affine transformations $f : \mathbb{R}^p \to \mathbb{R}^n$ and $b : \mathbb{R}^p \to \mathbb{R}^m$ given by

$$f(\theta) = \hat{f} + F\theta, \qquad b(\theta) = \hat{b} - AT\theta, \tag{3.2}$$

where $\hat{f} \in \mathbb{R}^n$, $F = \mathbb{R}^{n \times p}$, $\hat{b} \in \mathbb{R}^m$ and the transformation matrix $T = \mathbb{R}^{n \times p}$ ensure primal feasibility of $x = T\theta$.

All variables are sampled from a standard normal distribution $\mathcal{N}(0, 1)$, except for $\hat{b}$, which is drawn from a uniform distribution over $[0, 1)$ to ensure that the origin is feasible. This is necessary to guarantee that the DAQP solver used in the implementation finds a feasible solution.

In the initial experiment, all problem instances share a common quadratic matrix H and constraint matrix A, such that the variations between instances are completely determined by the parameter θ. However, to assess the generalization capabilities of the model to a broader class of quadratic programs, additional experiments are conducted on a dataset where H and A are resampled for each problem instance. A detailed comparison between the fixed and varying matrix settings is provided in Section 4.1.2.

To ensure data independence and meaningful evaluation, the problem instances in the training, validation, and test sets are generated using distinct random seeds. Specifically, the training data is generated with the random seed 123, the validation data with 124, and the test data with 125. The datasets are split in a ratio of 80% for training, 10% for validation, and 10% for testing.

3.2 Graph representation of QPs

To train a Graph Neural Network on the generated problems, the QPs need to be represented as graphs, where variables and constraints are modeled as different node types and their relationships as edges. The following graph representation of a QP is based on [30].

Definition 3.2.1. The **graph representation** of a QP is defined by $G = (W \cup C, E_W \cup E_C)$, where

- the set $W = \{1, \ldots, n\}$ represents the **variable nodes**. The i-th vertex represents the i-th variable of the QP and the assigned feature vector $x_s \in \mathbb{R}^1$ captures the corresponding component of f as well as the node type.
- the set $C = \{n + 1, \ldots, m\}$ represents the **constraint nodes**. The j-th vertex represents the j-th constraint of the QP and the assigned feature vector $x_s \in \mathbb{R}^2$ captures the corresponding component of b as well as the direction of the inequality sign and the node type.

- the edges contained in E_W and their feature vectors are given by the adjacency matrix H, which defines the quadratic term in the QP.
- the edges contained in E_C and their feature vectors are given by the adjacency matrix A and show the relations between the variables and constraints.

Example 3.2.2. The graph representation is illustrated on Example 2.1.1, which is given by

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2}\begin{pmatrix} x_1 & x_2 \end{pmatrix} \underbrace{\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}}_{=H} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \underbrace{\begin{pmatrix} -4 & -8 \end{pmatrix}}_{=f} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ \text{subject to} \quad & \underbrace{\begin{pmatrix} -x_1 + 2x_2 \end{pmatrix}}_{=A} \leq \underbrace{\begin{pmatrix} 0 \end{pmatrix}}_{=b}. \end{aligned} \tag{3.3}$$

The graph representation shown in Figure 3.1 is constructed according to the definition in 3.2.1, where different node types are distinguished by both color and the final component of their feature vector. Colored nodes represent the variables nodes with a node feature value of 0, while gray nodes represent constraint nodes with a node feature value of 1. The two variable nodes contain the components of vector f as features, whereas the two constraint nodes contain the corresponding values of b and the inequality type as feature vector. The matrices H and A are encoded as edge features connecting the nodes.

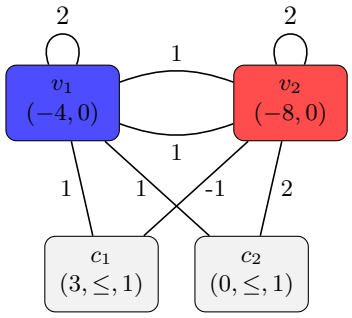

Figure 3.1: Graph representation of the QP presented in Example 2.1.1.

To convert the generated QPs into graphs suitable for GNN processing, the graph representation defined in 3.2.1 is implemented in Python. The resulting graphs are stored using the Data object from PyTorch Geometric. Consequently, the datasets used for training, validation, and testing do not contain the QPs directly, but rather their corresponding graph representations.

3.3 Mapping learned by the GNN

The key idea of using machine learning on the generated problems is to learn a mapping p from a problem instance represented by a graph G to the resulting active set $\mathcal{A}^*$

$$\mathrm{p} : \mathcal{G} \to \mathcal{P}(\mathbb{N}_m), \quad \mathrm{p}(G) = \mathcal{A}^*, \tag{3.4}$$

where $\mathcal{G}$ denotes the set of all graphs and $\mathcal{P}$ denotes the power set. The mapping corresponds to a node classification task on the graph G, determining whether each constraint vertex is active or not. Finding this mapping is not trivial, since the optimal solution x^* must be known to determine the active set $\mathcal{A}^*$ [22]. The predicted active set of a QP serves as the initial working set $\mathcal{W}_0$, warm-starting the solver, such that the quality of the mapping is given by the accuracy of the GNN. If the prediction matches the true active set at the optimal solution, the solver converges in a single iteration. Otherwise, additional iterations are required to adjust the working set until it aligns with the active set at the optimal solution. An inaccurate initial prediction may still reduce the overall number of iterations, as Algorithm 1 adapts the working set by at most one constraint per iteration. Consequently, this approach is expected to reduce the computational complexity by reducing the number of iterations and therefore time compared to solving each iteration fully with Algorithm 1 [38].

3.4 DAQP solver

We employ a supervised learning approach, which requires labeled data to associate each generated problem instance with a ground-truth outcome y for comparison. To obtain these labels, each problem is solved using the DAQP solver [37], a computational efficient implementation of the dual active-set method.

The solver follows Algorithm 1 and employs efficient numerical updates based on the following factorization

$$M_{\mathcal{W}} M_{\mathcal{W}}^T = L D L^T, \tag{3.5}$$

where $D \in \mathbb{R}^{|\mathcal{W}| \times |\mathcal{W}|}$ represents a diagonal matrix and $L \in \mathbb{R}^{|\mathcal{W}| \times |\mathcal{W}|}$ is a lower unit triangular matrix. This factorization provides several advantages. Since the working set is updated by at most one constraint per iteration, the matrices L and D can be iteratively updated using rank-one updates, avoiding the need for full recomputation in each iteration, which significantly reduces computational complexity.

With this matrix factorization, the check performed in Step 2 can be computed very efficiently. The singularity of $M_{\mathcal{W}} M_{\mathcal{W}}^T$ can be directly determined by the diagonal elements of matrix D. If all diagonal elements are non-zero ($D_{ii} \neq 0, \ \forall i \in \mathbb{N}_{|\mathcal{W}|}$), $M_{\mathcal{W}} M_{\mathcal{W}}^T$ is non-singular.

Furthermore, Step 3 can be effectively solved by leveraging the matrix factorization. Instead of computing the inverse of $M_{\mathcal{W}} M_{\mathcal{W}}^T$, which is computationally

expensive, the equation

$$M_{\mathcal{W}} M_{\mathcal{W}}^T \lambda_{\mathcal{W}}^* = -d_{\mathcal{W}} \Leftrightarrow \; LDL^T \lambda_{\mathcal{W}}^* = -d_{\mathcal{W}} \tag{3.6}$$

is solved using forward and backward substitution. The process is as follows

$$y \leftarrow \text{Solve } Ly = -d_{\mathcal{W}} \qquad \text{// forward substitution} \tag{3.7a}$$

$$z \leftarrow \text{Scale } y \text{ with } D, \quad [z]_i = \frac{[y]_i}{[D]_{ii}} \tag{3.7b}$$

$$\lambda_{\mathcal{W}}^* \leftarrow \text{Solve } L^T \lambda_{\mathcal{W}}^* = z. \qquad \text{// backward substitution} \tag{3.7c}$$

Equation 3.7a and 3.7b do not need to be fully recomputed in each iterations but can reuse results from previous iterations. The extent to which previous results can be reused depends on how $\mathcal{W}$ has changed in the current iteration.

3.5 Model architecture

The GNN model used in the experiments follows the architecture outlined in Section 2.3. During the model development, extensive tuning was performed to ensure robust predictive performance. This process included evaluating various graph convolution layers, such as GCNConv, as well as other layer architectures from PyTorch Geometric, optimized to effectively exploit the bipartite structure of the input graphs induced by the distinction between variable and constraint nodes[3].

Among the tested configurations, **Local Extrema Convolution (LEConv)** layers consistently outperformed alternative architectures across all metrics, enabling the model to capture local and global extrema within a graph. This is achieved by incorporating self-loops and learning functions that relate to local extrema as described in [53, 54].

Accordingly, the edge feature update function introduced in 2.24 is defined as

$$z_{ts}^{(l+1)} \coloneqq z_{ts}^{(l)} \cdot \left(x_s^{(l)} W_2 - x_t^{(l)} W_3 \right), \tag{3.8}$$

where W_i are learnable weight matrices. The LEConv architecture employs the sum as the aggregation function $\bigoplus$ in 2.25. Combining the edge feature update and the aggregation function yields

$$m_s^{(l+1)} \coloneqq \sum_{t \in \mathcal{N}_s} z_{ts}^{(l)} \left(x_s^{(l)} W_2 - x_t^{(l)} W_3 \right), \tag{3.9}$$

[3] https://pytorch-geometric.readthedocs.io/en/2.5.1/cheatsheet/gnn_cheatsheet.html#

which represents the aggregated message for node s. The vertex update function $\psi(\cdot)$, as introduced in 2.26, is then given by

$$\begin{aligned} x_s^{(l+1)} &\coloneqq \sigma\left(x_s^{(l)} W_1 + m_s^{(l+1)}\right) \\ &= \sigma\left(x_s^{(l)} W_1 + \sum_{t \in \mathcal{N}_s} z_{ts}^{(l)} \left(x_s^{(l)} W_2 - x_t^{(l)} W_3\right)\right), \end{aligned} \tag{3.10}$$

where $\sigma(\,\cdot\,)$ denotes the activation function.

During early-stage experimentation, a range of architectural parameters were systematically explored, which includes layer width, network depth, learning rate, and classification threshold. The final configuration was selected based on its strong performance on smaller problem instances, which enabled for fast testing.

The final model consists of three layers, an input layer, a hidden layer and an output layer. All layers share the same architectural structure, presented above. The primary difference lies in their respective width, which is adapted according to the specific role of each layer. The hidden layer has a width of 128. Further, LeakyRELU was chosen as the activation function with a negative slope of 0.1, defined by

$$\sigma(x) \coloneqq \max(0, x) + 0.1 \min(0, x). \tag{3.11}$$

To interpret the output as probabilities indicating whether a node is active or inactive, the sigmoid activation function is applied to the output layer, mapping the results to the interval $[0, 1]$.

Chapter 4

Experiments

We evaluate the GNN-based approach for solving QPs, as outlined in Section 3, through a series of experiments designed to address the research questions. Three distinct sets of experiments are conducted, each corresponding to one of the research questions.

All experiments are executed on a local machine running Windows 11 Pro, equipped with 16 GB RAM and a 12th Gen Intel(R) Core(TM) i5-1235U CPU. The implementation is available on Github[1]. All methods are implemented in Python, with the Graph Neural Network developed using the PyTorch Geometric framework.

4.1 Active-set prediction using GNN

In this section the following research question is examined:

> How accurately can a GNN predict the active constraints of a QP?

To investigate this question, a experimental pipeline was constructed based on the methods outlined in Section 3. The data is first generated as described in Section 3.1. The resulting problem instances are converted into graph representations as explained in Section 3.2, and stored using the DataLoader class from PyTorch Geometric. The resulting graphs serve as input to the model presented in detail in Section 3.5.

The model is trained using the AdamW optimizer with default learning rate settings. To address class imbalance, the loss function is defined as a weighted Binary Cross-Entropy loss, where the class weights are derived from the empirical class distribution.

[1] https://github.com/ellaschmidtobreick/acc_daqp

To prevent overfitting, we apply early stopping[2] based on the validation loss. Specifically, training is terminated if the validation loss does not improve by at least 0.001 over five consecutive epochs. The model parameters corresponding to the best validation performance are retained.

The evaluation is conducted using standard classification metrics, including accuracy, prediction, recall and the F1-score. Since the task is framed as a node-level classification problem, these metrics are computed over the entire set of nodes. For instance accuracy reflects the proportion of correctly classified nodes across the entire dataset.

However, these metrics do not capture the model's performance at the level of individual graphs. To provide a more fine-grained, graph-level assessment, custom evaluation metrics are introduced.

The initial approach considered was the **Graph Accuracy**, which measures the proportion of graphs for which all nodes are predicted correctly, given by

$$\frac{\#\text{ fully correctly predicted graphs}}{\#\text{ total graphs}}. \tag{4.1}$$

In practice, this metric is only informative for small problem instances, e.g. problems with two variables and five constraints. As the problem size increases, the likelihood of at least one node being misclassified also increases, often reducing the metric to zero. Since this limits its interpretability for larger problem sizes, the metric is omitted from further analysis.

To mitigate this limitation while preserving a graph-level perspective, the metric **Average of Correctly predicted Nodes per Graph (ACNG)** is introduced. This metric quantifies the average proportion of correctly classified nodes per graph and is normalized by the total number of nodes in each graph to allow for comparability across varying graph sizes

$$\text{Average}\left(\frac{\#\text{ correctly predicted nodes per graph}}{\#\text{ total nodes per graph}}\right). \tag{4.2}$$

While the resulting values are often very similar to standard node-level accuracy, ACNG provides a complementary graph-level perspective and is therefore included in the following results.

To provide a comprehensive evaluation, the reported metrics include values for training, validation and testing. All metrics are reported as percentages and rounded to two decimal places. When referring to the naïve model, it is meant that all nodes are classified as inactive corresponding to cold-starting the solver. As a baseline for comparison, a naïve model that classifies all nodes as inactive, corresponding to cold-starting the solver, is given.

[2] https://www.geeksforgeeks.org/how-to-handle-overfitting-in-pytorch-models-using-early-stopping/

4.1.1 Baseline model

In the first part of the experiment, the model was trained on problem instances consisting of 10 variables and 40 constraints. The dataset was generated according to the procedure described in Section 3.1, using fixed matrices H and A across all instances within each phase. The model is trained for 18 epochs and terminates based on the early stopping criterion.

The evaluation results are presented in Table 4.1. The model demonstrates strong performance on the test data, achieving scores above 90% across all standard metrics in comparison to the naïve model only achieving an accuracy of 82.96%. The model also performs well on the custom metric, with only misclassifying 2% of the nodes per graph, which corresponds to approximately 1.5 nodes per instance. For comparison, the naïve baseline model, evaluated on the same data, achieves only 83% accuracy and misclassifies 17% of the nodes per graph and therefore performs significantly worse overall.

H fixed, A fixed	Train	Val	Test
Accuracy (%)	97.78	98.04	97.73
Precision (%)	94.48	96.29	92.32
Recall (%)	92.36	91.97	94.55
F1-score (%)	93.41	94.08	93.42
ACNG (%)	97.99	97.98	97.79

Table 4.1: Metrics of the tuned GNN model trained on the 10 variables and 40 constraints with fixed matrices H and A.

Given the model's strong performance under fixed conditions, the next step is to investigate how its performance generalizes when the matrices H and A vary across problem instances.

4.1.2 Varying edge weights

Three additional training scenarios are considered to capture different levels of variability while still keeping the problem size and therefore the graph structure as in the previous experiment:

1. The matrix A is sampled independently for each problem instance.
2. The matrix H is sampled independently for each problem instance.
3. Both matrices H and A are sampled independently for each instance.

In the original setup, where the matrices H and A are fixed across all problem instances, variability arises solely from changes in the node features. In

contrast, the extended scenarios introduce variability both in the node features and in the edge features, which are derived from the underlying problem structure.

For comparison, the test metrics of all four models are shown in Table 4.2. It is evident that the models trained on graphs with varying edge weights perform significantly worse than the model trained with fixed matrices H and A, with precision, recall and F1-score dropping substantially, occasionally falling below 50%. The ACNG scores of both models, when matrix A is flexible, are comparable to that of the naïve model, indicating similarly weak performance. Detailed training and validation results for the models trained on varying matrices can be found in Appendix A.

10 var., 40 const.	H, A fixed	A flexible	H flexible	H, A flexible
Accuracy (%)	97.73	83.62	88.07	83.01
Precision (%)	92.32	54.39	69.02	52.71
Recall (%)	94.55	45.35	59.12	40.41
F1-score (%)	93.42	49.46	63.69	45.75
ACNG (%)	97.79	83.71	88.02	82.83

Table 4.2: Test metrics of tuned GNN models trained on problem instances with 10 variables and 40 constraints.

4.1.3 Scaling of problem size

To investigate whether the high precision observed for the model trained on fixed edge weights generalizes to larger problem instances, the model from the previous section, trained on graphs with 10 variable nodes and 40 constraint nodes, is tested on problem instances with 25 variables and 100 constraints. This setup preserves the ratio of variable to constraint nodes from previous experiments while increasing the overall problem size. When evaluating the model trained on the smaller dataset against the larger problem instances, test accuracy decreases to 71.98%, with precision, recall, and F1-score each falling below 36%. A detailed table of these results is provided in Appendix A, which clearly shows that the model trained on a fixed graph size does not generalize well to larger graph instances.

To address this limitation, an additional model is trained on problem instances with 25 variables and 100 constraints. The corresponding results remain strong despite the increased problem complexity. As presented in Table 4.3, nearly all standard metrics exceed 90%, with the exception of precision, which

reaches 85.55% on the test set. This performance is well above the naïve model's accuracy of 81.98%, demonstrating the effectiveness of the trained GNN. The ACNG score is marginally lower compared to the model trained on smaller instances, but the model still only misclassifies approximately 4 of 125 constraints per graph.

H fixed, A fixed	Train	Val	Test
Accuracy (%)	96.44	95.94	96.56
Precision (%)	85.84	83.11	85.55
Recall (%)	96.16	97.42	97.33
F1-score (%)	90.71	89.70	91.06
ACNG (%)	96.44	95.94	96.48

Table 4.3: Metrics of the tuned GNN model trained on the 25 variables and 100 constraints with fixed matrices H and A.

4.1.4 Varying problem sizes

To build a more robust model, the training data was extended to include graphs of different sizes. In this experiment, one half of the input data consists of graphs with 10 variable nodes and 40 constraint nodes, while the other half consists of graphs with 25 variable nodes and 100 constraint nodes. To keep the total number of training samples consistent with previous experiments, 2500 problem instances are generated for each graph size.

H fixed, A fixed	Train	Val	Test	Test	Test
Variable nodes	[10,25]	[10,25]	[10,25]	[10]	[25]
Constraint nodes	[40,100]	[40,100]	[40,100]	[40]	[100]
Accuracy (%)	96.05	96.21	96.77	96.92	96.79
Precision (%)	87.74	89.12	89.34	91.59	88.91
Recall (%)	90.48	89.58	92.85	90.21	93.93
F1-score (%)	89.09	89.35	91.06	90.89	91.35
ACNG (%)	-	96.08	97.79	96.92	96.79

Table 4.4: Metrics of the tuned GNN model trained on the problems with 10 variables and 40 constraints and 25 variables and 100 constraints with fixed matrices H and A.

The trained model is evaluated on three different test sets, one containing graphs with mixed graph sizes, and the other two containing only problems of the respective sizes. The results are presented in Table 4.4.

The model performs consistently well across all test datasets, producing comparable results. On the mixed-size dataset, performance metrics generally fall between those observed of the smaller and larger problem instance datasets.

The model achieves higher accuracy and precision when tested on the dataset with smaller problem instances, while it performs better on recall and F1-score when tested on the dataset containing larger problem instances. This trend aligns with the observations discussed in Section 4.1.3, where increasing the problem size resulted in lower precision but a higher recall. Notably, the F1-score on the dataset with larger problem instances is relatively high, as the drop in precision is less pronounced compared to the previous experiments. Overall, the performance differences are only marginal, indicating good generalization.

The ACNG score on the training dataset could not be computed, as the shuffling performed by the training data loader prevents the identification of the correspondence between nodes and their original graph instances, including graph size.

Interestingly, the ACNG score on the mixed dataset does not fall between the values from the smaller and larger datasets, but instead exceeds both, highlighting the model's potential to generalize effectively when exposed to a broader range of graph sizes.

4.2 Impact of problem structure on predictive performance

Following the demonstration in the previous section that accurate active set predictions can be achieved using a GNN, especially under fixed edge weights, this section addresses the second research question:

- How does incorporating the structure of the optimization problem affect predictive performance?

The aim is to compare the performance metrics of a standard **Multi-layer Perceptron (MLP)** with those of a Graph Neural Network (GNN), in order to evaluate whether incorporating the problem structure through a graph-based model yields improved predictive performance.

Since the MLP requires a single vector as input, all components defining a QP are concatenated into a vector. The Hessian matrix $H \in \mathbb{R}^{n \times n}$ and the constraint matrix $A \in \mathbb{R}^{m \times n}$ are flattened and concatenated with the vectors

$f \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, resulting in an input vector of dimension $n \cdot n + m \cdot n + n + m$.

The MLP outputs a binary classification to predict which constraints are active. To ensure comparability with the graph-based model, the output dimension is set to $n + m$. While predictions are made for both variables and constraints, the primary interest lies in identifying which of the m constraints are predicted to be active.

The MLP architecture was designed to closely follow the GNN architecture to ensure comparability. It consists of three fully connected layers, with Leaky ReLU activation functions applied between layers. A sigmoid activation function is used at the output layer to map predictions to the interval $[0, 1]$. The same classification threshold as in the GNN model is employed to maintain consistency in evaluation between the two architectures.

4.2.1 Baseline model

First, the baseline MLP model is compared against the baseline GNN model. The results of the baseline MLP model are given in Table 4.5.

H fixed, A fixed	Train	Val	Test
Accuracy (%)	98.69	98.56	98.69
Precision (%)	93.73	92.84	93.40
Recall (%)	98.92	99.13	99.34
F1-score (%)	96.25	95.88	96.28
ACNG (%)	98.69	98.56	98.72

Table 4.5: Metrics of the MLP model trained on the 10 variables and 40 constraints with fixed matrices H and A.

The metrics obtained when evaluating the MLP model are consistently high, exceeding 92% across all evaluated metrics. Compared to the baseline GNN model, given in Table 4.1, the MLP achieves approximately 1% higher accuracy, precision, and ACNG, while recall and F1-score show improvements exceeding 1%. These results indicate that the active set of QPs with fixed Hessian matrix H and constraint matrix A can be accurately predicted using a standard MLP. Notably, this approach also offers a significant advantage in terms of computational efficiency during training compared to the GNN-based model.

4.2.2 Varying edge weights

Consistent with the previous section, performance metrics for MLP models under varying matrices H and A are evaluated to determine whether the MLP

consistently outperforms the GNN in active-set prediction. The corresponding results are presented in Table 4.6.

10 var., 40 const.	H, A fixed	A flexible	H flexible	H, A flexible
Accuracy (%)	98.69	75.71	92.10	76.10
Precision (%)	93.40	37.96	72.87	38.38
Recall (%)	99.34	59.09	88.18	57.47
F1-score (%)	96.28	46.22	79.80	46.02
ACNG (%)	98.72	75.78	92.07	76.20

Table 4.6: Test metrics of MLP models trained on problem instances with 10 variables and 40 constraints.

The results indicate that the MLP model achieves high performance when both matrices H and A are fixed. Its effectiveness decreases when either matrix varies across problem instances.

Resampling the constraint matrix A leads to a substantial reduction in all metric values for the MLP, with precision and F1-score falling below 50%, indicating a reduced ability to accurately identify active constraints under varying problem structures. In this setting, the GNN model clearly outperforms the MLP, as shown by the comparison of Tables 4.2 and 4.6 (columns 2 and 4), achieving higher scores across all metrics except recall. This performance gap underscores the GNN's advantage in leveraging structural information from neighboring nodes, particularly relevant since A defines the constraints to be classified as active or inactive.

In the case where the matrix A is fixed and the Hessian matrix H varies, the MLP outperforms the GNN. Since the structure of the constraints remains unchanged, the influence of neighboring node features is reduced, and the GNN's structural advantage is less impactful.

Furthermore, it is notable, that the MLP achieves higher recall than the GNN across all datasets, with the largest improvement of 29% observed for models trained on data with a varying Hessian matrix H.

4.2.3 Scaling of problem size

To remain consistent with the previous chapter, an MLP model, trained on scaled problem sizes is evaluated and compared to the performance of the GNN model presented in Section 4.1.2. To enable this comparison, the MLP's input and output size are adjusted accordingly.

When scaling the problem size while keeping the matrices H and A fixed,

H fixed, A fixed	Train	Val	Test
Accuracy (%)	98.45	98.48	98.61
Precision (%)	93.28	93.25	93.93
Recall (%)	98.53	98.74	98.68
F1-score (%)	95.83	95.92	96.25
ACNG (%)	98.45	98.48	98.61

Table 4.7: Metrics of the MLP model trained on the 25 variables and 100 constraints with fixed matrices H and A.

the MLP maintains a strong performance, as shown in Table 4.7. Compared to the GNN results in Table 4.3, the MLP consistently outperforms the GNN across all metrics. The largest improvement is observed in precision, which had been comparatively low for the GNN in the scaled setting. The MLP achieves particularly high scores in terms of accuracy, recall and ACNG.

As the problem size increases, the substantially reduced training time of the MLP becomes even more advantageous. For instance, with 25 variables and 100 constraints, the MLP completes training in approximately 7 minutes, while the GNN requires over 40 minutes. Consequently, when computational resources or time are limited, the MLP demonstrates a clear advantage over the GNN in terms of training efficiency.

4.2.4 Varying problem sizes

Traditional MLP models require a fixed input size, as they operate on a single concatenated vector representation. This limits their flexibility with respect to varying problem sizes. Similarly, the output dimension of the MLP, corresponding to the total number of variables and constraints $n + m$, must remain fixed, which is incompatible with with the natural variability of problem dimensions.

In contrast, GNNs inherently support inputs and outputs of varying sizes, as problem instances are modeled as graphs with dynamic numbers of nodes and edges. Although the GNN performs slightly worse than the MLP in the baseline setting and on scaled problem sizes, its key advantage lies in its flexibility. The GNN can not only capture varying constraint structures across instances but also handle inputs of different sizes, an essential benefit that highlights the strength of GNNs over MLPs in scenarios with dynamic problem dimensions.

4.3 Effect of predictive model on solver efficiency

Following the demonstration in the previous section that incorporating the problem structure into the model can have a positive impact, especially for problem instances with varying matrix A, this section addresses the final research question:

- How does the predictive model affect optimization performance?

To assess the impact of the GNN's prediction, the solver performance is compared between cold-starting and warm-starting scenarios. As the baseline, the solver is cold-started, solving the problem without any prior information of the active set. In the warm-start scenario, the solver is initialized with the active set predicted by the GNN. The performance is evaluated based on the two metrics: runtime and number of iterations required for convergence.

When warm-starting the solver, the predicted active set is treated as a set of equality constraints. The DAQP solver thus takes two sets of constraints as input: a set of inequalities (classified as inactive) and a set of equality constraints (classified as active). Since the prediction may not be fully accurate, it can occur that the resulting problem becomes infeasible if a truly inactive constraint is incorrectly predicted as active. In such cases, the solver flags the problem as overdetermined and does not proceed with solving it.

As the solver does not have a built-in mechanism for solving overdetermined problems, this step was handled manually. Specifically, if the solver returns an overdetermined status, constraints are iteratively removed from the predicted active set until the problem becomes feasible. Since DAQP adds constraints one by one and stops as soon as a newly added constraint makes the problem infeasible, constraints are removed from the end of the working set, corresponding to the last added constraints that caused infeasibility. This manual removal happens before the performance evaluation begins since it could be integrated directly into the solver.

To ensure that the final solution corresponds to the initial problem and that no inactive constraint was incorrectly classified as active, potentially altering the solution, the problem is solved a second time using the adjusted active set. If the resulting constraints now match the true active set, this final solve requires only a single iteration, as discussed in Chapter 2.2. Both the runtime and the iteration count of this final, ideally single-iteration solve are included in the overall performance metrics.

In the initial solve, the runtime includes both set-up and solving time as reported by the solver. The set-up time is not counted again, as this feasibility check could, in principle, be integrated directly into the solver pipeline.

4.3.1 Baseline model

To assess whether leveraging the problem's graph structure through the GNN-predicted active set reduces runtime and the iteration count, the trained models are evaluated using these metrics. Figure 4.1 illustrates the impact of the model trained on graphs with 10 variable and 40 constraint nodes with fixed edge weights, demonstrating a clear reduction of approximately 75% in iteration count and 70% in runtime. For better visualization, extreme runtime outliers have been excluded from the figure to maintain a representative scale. Detailed results, including mean metrics for both cold- and warm-started solvers as well as the reduction achieved through active-set prediction, are presented in Table 4.8.

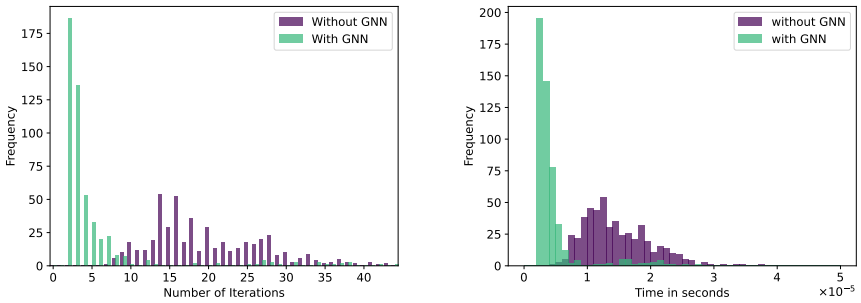

Figure 4.1: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 10 variables and 40 constraints with fixed matrices H and A.

4.3.2 Varying edge weights

All models trained on problem instances with 10 variables and 40 constraints are evaluated and compared in this section. The results presented in Table 4.8 summarize the mean performance metrics calculated across the various test datasets.

It can be observed that the models performing better during training and testing, as given in Section 4.1.2, tend to yield greater improvements when warm-starting the solver with the model's predicted active set. Among these, the model trained on fixed edge weights clearly outperforms the other models.

If only the matrix A is varied, a slight reduction in the iteration count is observed, however, the overall solving time increases, making warm-starting less effective than cold-starting in this scenario. The model on problem instances with varying matrices H and A also performs poorly, but showing a marginally greater runtime reduction compared to the case where only A varies. However,

it achieves a smaller decrease in the iteration count. These results highlight that both metrics, runtime and iteration count, are important and should be evaluated jointly to comprehensively assess solver performance. The figures of the remaining models are given in Appendix B.

10 var., 40 const.	H, A fixed	A flexible	H flexible	H, A flexible
Time cold-started (μs)	23.78	17.28	23.45	23.92
Time warm-started (μs)	7.07	18.59	16.40	21.50
Time reduction (μs)	**16.71**	**-1.31**	**7.05**	**2.42**
Iteration cold-started	19.91	21.27	20.23	19.88
Iteration warm-started	5.04	19.77	16.00	19.49
Iteration reduction	**14.87**	**1.50**	**4.22**	**0.38**

Table 4.8: Performance impact of incorporating the problem structure into the solver using a GNN model trained on problem instances with 10 variables and 40 constraints.

It should be emphasized that the iteration count serves as a direct and hardware-independent metric, whereas the runtime may vary depending on the computational environment.

4.3.3 Scaling of problem size

To provide a comprehensive overview, Table 4.9 presents the performance impact of the model trained on larger problem instances with 25 variables and 100 constraints.

25 var., 100 const.	H, A fixed
Time cold-started (μs)	152.31
Time warm-started (μs)	108.44
Time reduction (μs)	**43.87**
Iteration cold-started	58.04
Iteration warm-started	35.29
Iteration reduction	**22.75**

Table 4.9: Performance impact of incorporating the problem structure into the solver using a GNN model trained on problem instances with 25 variables and 100 constraints.

In this setup, the edge weights defined by the matrices H and A are sampled once and kept fixed across all problem instances.

As illustrated in Figure 4.2, both the number of iterations and the runtime are significantly reduced. The iteration count decreases by 39%, while the runtime is reduced by 29%. Although these percentages are lower than those observed for smaller problem instances, the overall potential for time and iteration savings increases, as cold-starting larger problems requires substantially more iterations and runtime. The figures of the remaining models are given in Appendix B.

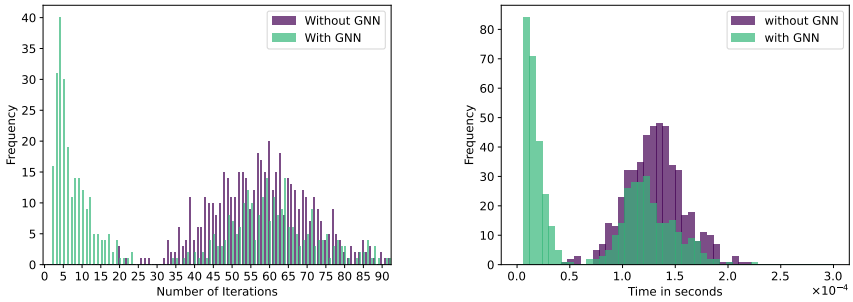

Figure 4.2: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 25 variables and 100 constraints with fixed matrices H and A.

Although the runtime is significantly reduced, the prediction time must be taken into account when warm-starting the solver. For the models trained on problem instances with 10 variables and 40 constraints, the average prediction time is approximately 4.65 ms, further details can be found in Appendix B. For the model trained on the larger problem instances, the average prediction time increases slightly to 6.89ms, but remains consistently in the order of milliseconds.

4.3.4 Varying problem sizes

The model trained on problem instances of varying sizes was evaluated on three separate test datasets. For each dataset, both solver runtime and iteration count were measured. The results are summarized in Table 4.10.

The results show a clear improvement in both iteration count and runtime, comparable to the performance of the previously trained models. The improvements achieved by the model trained on graphs of mixed sizes lie between those observed for the models trained exclusively on smaller or larger instances. This

outcome is expected, as the training data was composed of an equal proportion of both graph sizes.

Variable nodes	[10,25]	[10]	[25]
Constraint nodes	[40,100]	[40]	[100]
Time cold-started (μs)	103.65	22.91	257.83
Time warm-started (μs)	40.55	6.76	81.78
Time reduction (μs)	**63.1**	**16.15**	**176.05**
Iteration cold-started	38.67	19.91	58.04
Iteration warm-started	14.02	5.11	22.88
Iteration reduction	**24.65**	**14.80**	**35.16**

Table 4.10: Performance impact of incorporating the problem structure into the solver using a GNN model trained on problem instances with varying size.

The performance on the smaller problem instances is comparable to that presented in Section 4.3.2, where training was limited to graphs of matching size. For a complete overview, the corresponding figures are provided in Appendix B.

It is worth mentioning that, the average cold-starting runtime for the test dataset containing larger problem instances is notably higher than the corresponding value reported in Section 4.3.3, where the model was only trained on graphs of equal size. However, warm-starting the solver results in lower solving times compared to those in Section 4.3.3, suggesting a more effective use of the GNN-predicted active set. The positive effect is illustrated in Figure 4.3. Additionally, the iteration count is reduced by 60%, exceeding the 39% reduction observed in Section 4.3.3, despite having the comparable cold-start baselines.

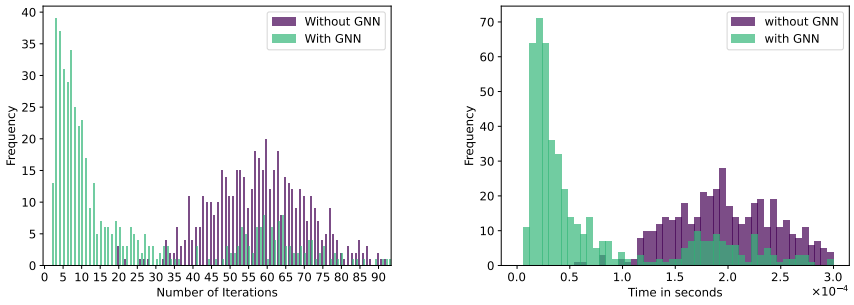

Figure 4.3: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 25 variables and 100 constraints using the mixed-size trained model.

Chapter 5

Discussion

As shown in the results presented in Section 4.1, the model's ability to learn the underlying pattern strongly depends on the structure of the training data. If the matrices H and A remain fixed across problem instances, the model is able to effectively capture the structure induced by the parameter θ.

However, once the edge weights defined by H and A vary across problem instances, the model's ability to learn this structure deteriorates. A possible explanation is that the matrices H and A are independent of the parameter θ, making it harder for the model to identify consistent patterns across instances. In particular, LEConv layers are optimized for detecting local extrema, which may not generalize well across graphs with different structures.

It is evident that varying the matrix A across problem instances has a more pronounced negative impact on model performance than varying the matrix H, as demonstrated by the outcomes of both sets of experiments. This difference likely stems from the way the graph is constructed. The matrix A defines the relationships between variable and constraint nodes, which are central to the classification task, as the goal is to identify active constraints. In contrast, the matrix H determines the edge weights between variable nodes, which are typically not active and thus less critical for the classification. As a result, changes in matrix A directly affect the graph structure relevant to the prediction task, while changes in matrix H have a comparatively smaller effect.

Furthermore, the experiments demonstrate, that accurate predictions can be maintained even when scaling the input size, as the model trained on larger graphs continues to achieve strong test performance. Notably, precision decreases while recall improves as the problem size increases. This trend is particularly important, given the recall is the more critical metric in this context. Misclassifying an active constraint as inactive is more problematic than the reverse. To determine whether this pattern persists for even larger problem sizes and to identify any clear trends, further investigation involving additional graph sizes is necessary.

The final set of experiments, which involved varying problem sizes within the training data, demonstrates that the resulting model can still effectively capture the underlying patterns. The observed improvement aligns with the expected performance when combining the results of models trained on single graph sizes. Notably, the mixed-size model outperforms the single-size model on larger problem instances, suggesting potential advantages for scaling to more complex problems. This observation motivates further investigation, particularly regarding the impact of training on a broader range of graph sizes simultaneously.

To demonstrate the importance of incorporating graph structure into the neural network architecture, the performance of the GNN was compared to that of a traditional MLP.

In the simpler cases, where the problem structure remains fixed, the MLP slightly outperforms the GNN. This could be attributed to the design of the GNN, which employs LEConv layers that are optimized to capture local extrema among neighboring nodes. When edge weights are constant across problem instances, these local variations are less pronounced, potentially limiting the GNN's ability to learn meaningful structural patterns.

As the constraint matrix A begins to vary across problem instances, the GNN demonstrates a clear performance advantage over the MLP. In this setting, the relationships between variable and constraint nodes, defined by matrix A, are essential to the classification task. The GNN is able to leverage this structural information and adapt to the changes, while the MLP, relying solely on concatenated feature vectors, lacks this capability.

A further strength of GNNs lies in their flexibility. Since they process graph-structured input, they can naturally accommodate problem instances of different sizes. This enables a single GNN model to generalize across a wide range of optimization problems. In contrast, MLPs require fixed-size input and output layers, restricting them to problems of a specific dimension unless retrained or redesigned.

Finally, it is important to highlight the practical advantage of MLPs in terms of computational efficiency. MLPs train significantly faster than GNNs, making them a valuable alternative when computational resources are constrained, though this comes at the cost of reduced accuracy in active-set prediction.

The overall results indicate that the active set can be predicted with high accuracy using a GNN, regardless of the problem size. In particular, using the well-performing model trained on data with fixed matrices H and A, incorporating the structure of the optimization problem as a graph, positively impacts the performance. When warm-starting the solver with the predicted active set, the number of iterations and the solving time significantly decreases.

It is worth noting that reported runtimes reported in Section 4.3 include both solving and set-up phases, which may scale differently, such that a more detailed breakdown could provide additional insights.

Although these results are promising, the primary bottleneck remains. As discussed in Section 4.3, the prediction time of the GNN is the main limiting factor in the proposed approach. While the GNN prediction operates in the order of 10^{-3} seconds, the DAQP solver itself runs in the order of 10^{-5} seconds for smaller problem instances. To fully benefit from the reduced number of solver operations, problem instances would need to be substantially larger than those considered in this study. For example, problem instances with 25 variables and 100 constraints already have solving times in the order of 10^{-4} seconds. Future work could investigate larger problem instances to identify the threshold at which the time saved by warm-starting the solver surpasses the overhead introduced by the GNN prediction.

Chapter 6

Conclusion

This thesis explored the use of graph neural networks to accelerate the solving of quadratic programs, a widely used class of optimization problems. The focus was on active-set methods, specifically the DAQP solver, which iteratively determine a subset of inequality constraints, known as the active set, that can be treated as equalities to simplify the problem. If the active set is known beforehand, the solution can be obtained in a single iteration. The central idea was to use a GNN to predict the active set prior to solving, thereby reducing the number of iterations required and improving the overall efficiency.

The results demonstrated that GNNs can accurately predict active sets and improve solver performance, especially when the structure of the problem instances, given by the matrices H and A, remains consistent. However, the performance decreased significantly when the problem instances varied, particularly due to changes in matrix A, which defines edge weights between variable and constraint nodes. This highlights the sensitivity of the approach to structural variations and emphasizes the importance of consistency for effective warm-starting using GNNs.

Moreover, the study showed that strong performance can still be achieved when scaling the problem size up or using mixed-sized training data. Although training the GNN on more complex input graphs required more time, the model was still able to learn meaningful structure using only three layers. Importantly, the potential for iteration and runtime savings increases with problem size, which supports the scalability of the approach in principle.

In addition, a comparison with a traditional MLP provided further insight into the benefits and limitations of incorporating graph structure into the model architecture. While the MLP performed slightly better in scenarios with fixed problem structure, likely due to its simpler design and lower computational resources, the GNN demonstrated clear advantages when the constraint matrix A varied across instances. In these cases, the GNN effectively leveraged structural

information critical to predicting the active set. Moreover, unlike the MLP, which is restricted to fixed-size inputs and outputs, the GNN architecture offers the flexibility to handle problem instances of varying sizes, further underscoring its suitability for real-world applications.

A key limitation identified was the prediction time of the GNN, which emerged as a bottleneck in practical applications. While GNN prediction time lies in the order of 10^{-3} seconds, the DAQP solver operates one to two orders of magnitude faster, depending on the problem size.

Future work should therefore explore the scalability of the method in more detail, particularly to determine the problem size beyond which the prediction overhead becomes negligible. Additionally, training GNNs on mixed-size problem instances should be further explored, as this may enhance generalization and broaden the practical applicability of the approach.

In summary, the findings indicate that the structure of QPs can be effectively captured using graphs, and that GNN-based prediction of the active set can accelerate active-set solvers. However, the benefit depends on the problem size and structural consistency. For small-scale problems, the cost of prediction may outweigh the savings in iteration time.

Declaration on the Use of Generative AI

The Generative AI service ChatGPT was used to support debugging during implementation, particularly for understanding and interpreting error messages. Additionally, it assisted in finding synonyms and alternative formulations to enhance the writing style. Generative AI was also used to resolve LaTeX-related issues, including problems with formatting the bibliography.

Appendix A

Additional results on active-set prediction

The following presents the training and validation results for the models trained on datasets with 10 variables and 40 constraints, where the matrix H or A is varied. The tables are provided to make all relevant data readily accessible.

H fixed, A flexible	Train	Val	Test
Accuracy (%)	83.21	83.65	83.62
Precision (%)	53.35	56.85	54.39
Recall (%)	33.54	25.79	45.35
F1-score (%)	41.19	35.48	49.46
ACNG (%)	83.02	83.12	83.71

Table A.1: Metrics of the tuned GNN model trained on the 10 variables and 40 constraints with a fixed matrix H and a flexible matrix A.

Further, the full results of the model trained on problem instances with 10 variables and 40 constraints, but tested on problem instances with 25 variables and 100 constraints is presented in Table A.4.

H flexible, A fixed	Train	Val	Test
Accuracy (%)	87.92	87.78	88.07
Precision (%)	69.50	67.67	69.02
Recall (%)	57.17	59.23	59.12
F1-score (%)	62.73	63.17	63.69
ACNG (%)	87.64	86.76	88.02

Table A.2: Metrics of the tuned GNN model trained on the 10 variables and 40 constraints with a flexible matrix H and a fixed matrix A.

H flexible, A flexible	Train	Val	Test
Accuracy (%)	83.02	83.2	83.01
Precision (%)	52.98	56.43	52.71
Recall (%)	32.39	21.55	40.41
F1-score (%)	40.20	31.19	45.75
ACNG (%)	82.98	81.58	82.83

Table A.3: Metrics of the tuned GNN model trained on the 10 variables and 40 constraints with a flexible matrices H and A.

H fixed, A fixed	Train	Val	Test
Variable nodes	[10]	[10]	[25]
Constraint nodes	[40]	[40]	[100]
Accuracy (%)	97.78	98.04	71.98
Precision (%)	94.48	96.29	28.21
Recall (%)	92.36	91.97	35.89
F1-score (%)	93.41	94.08	31.59
ACNG (%)	97.99	97.98	71.96

Table A.4: Metrics of the tuned GNN model trained on the 10 variables and 40 constraints with fixed matrices H and A but tested on graphs with 25 variable nodes and 100 constraint nodes.

Appendix B

Additional results on solver improvement

The following figures are included to provide a complete overview of the data. Each pair of figures corresponds to one of the models trained with a varying matrix H or A.

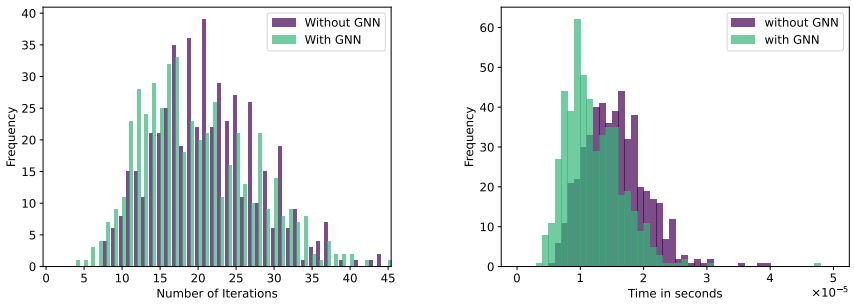

Figure B.1: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 10 variables and 40 constraints with a fixed matrix H and a flexible matrix A.

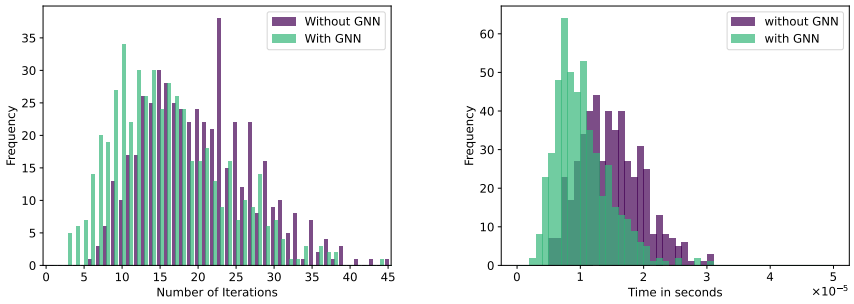

Figure B.2: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 10 variables and 40 constraints with a independent matrix H and a fixed matrix A.

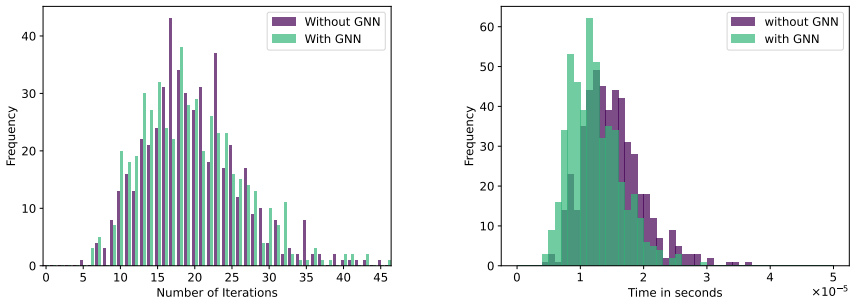

Figure B.3: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 10 variables and 40 constraints with independent matrices H and A.

Table B.1 presents the prediction times of the models trained on graphs with 10 variable nodes and 40 constraint nodes.

10 var., 40 const.	H, A fixed	A flexible	H flexible	H, A flexible
Prediction time (ms)	4.66	4.54	4.46	4.94

Table B.1: Prediction time of the models trained on problem instances with 10 variables and 40 constraints.

The following figures show the improvement in iteration count and runtime for the model trained on mixed-sized graphs and tested on different datasets. The figures are included to provide a complete overview of the data.

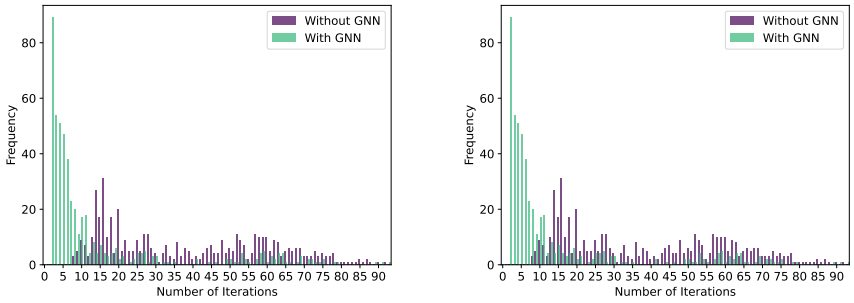

Figure B.4: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances of mixed sizes using the mixed-size trained model.

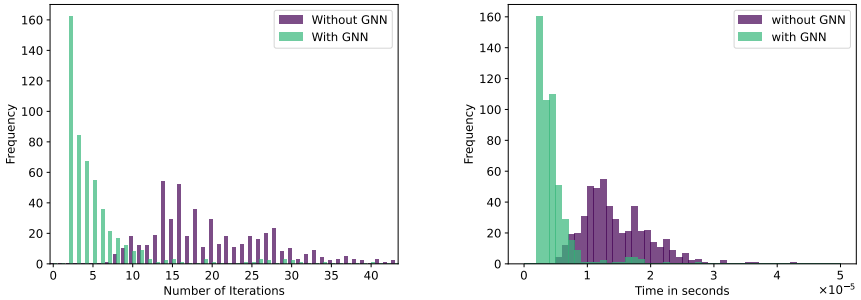

Figure B.5: Comparison of iterations (left) and time (right) of cold-starting and warm-starting the DAQP solver on problem instances with 10 variables and 40 constraints using the mixed-size trained model.

Bibliography

[1] S. Kuindersma et al. "An efficiently solvable quadratic program for stabilizing dynamic locomotion". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. ISSN: 1050-4729. 2014, pp. 2589–2594.

[2] R. A. Bartlett et al. "Quadratic programming algorithms for large-scale model predictive control". In: *Journal of Process Control* vol. 12, no. 7 (2002), pp. 775–795.

[3] J. Gondzio and A. Grothey. "Parallel interior-point solver for structured quadratic programs: Application to financial planning problems". en. In: *Ann Oper Res* vol. 152, no. 1 (2007). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Kluwer Academic Publishers-Plenum Publishers, pp. 319–339.

[4] G. Mitra et al. "Quadratic programming for portfolio planning: Insights into algorithmic and computational issues". en. In: *J Asset Manag* vol. 8, no. 3 (2007). Company: Palgrave Distributor: Palgrave Institution: Palgrave Label: Palgrave Number: 3 Publisher: Palgrave Macmillan UK, pp. 200–214.

[5] R. Sambharya et al. "End-to-End Learning to Warm-Start for Real-Time Quadratic Optimization". en. In: *Proceedings of The 5th Annual Learning for Dynamics and Control Conference*. ISSN: 2640-3498. PMLR, 2023, pp. 220–234.

[6] J. Ichnowski et al. "Accelerating Quadratic Optimization with Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 21043–21055.

[7] M. Klaučo et al. "Machine learning-based warm starting of active set methods in embedded model predictive control". In: *Engineering Applications of Artificial Intelligence* vol. 77 (2019), pp. 1–8.

[8] S. W. Chen et al. "Large scale model predictive control with neural networks and primal active sets". In: *Automatica* vol. 135 (2022), pp. 109947.

[9] C. V. Rao et al. "Application of Interior-Point Methods to Model Predictive Control". en. In: *Journal of Optimization Theory and Applications* vol. 99, no. 3 (1998). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 3 Publisher: Kluwer Academic Publishers-Plenum Publishers, pp. 723–757.

[10] Y. Wang and S. Boyd. "Fast Model Predictive Control Using Online Optimization". In: *IEEE Transactions on Control Systems Technology* vol. 18, no. 2 (2010). Conference Name: IEEE Transactions on Control Systems Technology, pp. 267–278.

[11] A. Bemporad. "A Quadratic Programming Algorithm Based on Nonnegative Least Squares With Applications to Embedded Model Predictive Control". In: *IEEE Transactions on Automatic Control* vol. 61, no. 4 (2016). Conference Name: IEEE Transactions on Automatic Control, pp. 1111–1116.

[12] C. Schmid and L. T. Biegler. "Quadratic programming methods for reduced hessian SQP". In: *Computers & Chemical Engineering*. An International Journal of Computer Applications in Chemical Engineering vol. 18, no. 9 (1994), pp. 817–832.

[13] H. J. Ferreau et al. "qpOASES: a parametric active-set algorithm for quadratic programming". en. In: *Math. Prog. Comp.* vol. 6, no. 4 (2014). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 4 Publisher: Springer Berlin Heidelberg, pp. 327–363.

[14] D. Goldfarb and A. Idnani. "A numerically stable dual method for solving strictly convex quadratic programs". en. In: *Mathematical Programming* vol. 27, no. 1 (1983). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer-Verlag, pp. 1–33.

[15] J. B. Rosen. "The Gradient Projection Method for Nonlinear Programming. Part I. Linear Constraints". In: *Journal of the Society for Industrial and Applied Mathematics* vol. 8, no. 1 (1960). Publisher: Society for Industrial and Applied Mathematics, pp. 181–217.

[16] J. B. Rosen. "The Gradient Projection Method for Nonlinear Programming. Part II. Nonlinear Constraints". In: *Journal of the Society for Industrial and Applied Mathematics* vol. 9, no. 4 (1961). Publisher: Society for Industrial and Applied Mathematics, pp. 514–532.

[17] J. Douglas and H. H. Rachford. "On the Numerical Solution of Heat Conduction Problems in Two and Three Space Variables". In: *Transactions of the American Mathematical Society* vol. 82, no. 2 (1956). Publisher: American Mathematical Society, pp. 421–439.

[18] P. L. Lions and B. Mercier. "Splitting Algorithms for the Sum of Two Nonlinear Operators". In: *SIAM Journal on Numerical Analysis* vol. 16, no. 6 (1979). Publisher: Society for Industrial and Applied Mathematics, pp. 964–979.

[19] D. Gabay and B. Mercier. "A dual algorithm for the solution of nonlinear variational problems via finite element approximation". In: *Computers & Mathematics with Applications* vol. 2, no. 1 (1976), pp. 17–40.

[20] B. Stellato et al. "OSQP: an operator splitting solver for quadratic programs". en. In: *Math. Prog. Comp.* vol. 12, no. 4 (2020). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 4 Publisher: Springer Berlin Heidelberg, pp. 637–672.

[21] J. Nocedal and S. J. Wright, eds. *Numerical Optimization*. en. Springer Series in Operations Research and Financial Engineering. New York: Springer-Verlag, 1999.

[22] V. Zinage et al. "TransformerMPC: Accelerating Model Predictive Control via Transformers". In: arXiv:2409.09266. 2024.

[23] Y. Bengio et al. *Learning a synaptic learning rule*. Citeseer, 1990.

[24] S. Thrun and L. Pratt. "Learning to Learn: Introduction and Overview". en. In: *Learning to Learn*. Springer, Boston, MA, 1998, pp. 3–17.

[25] S. Hochreiter et al. "Learning to Learn Using Gradient Descent". en. In: *Artificial Neural Networks — ICANN 2001*. Springer, Berlin, Heidelberg, 2001, pp. 87–94.

[26] M. Andrychowicz et al. "Learning to learn by gradient descent by gradient descent". In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016.

[27] O. Wichrowska et al. "Learned Optimizers that Scale and Generalize". en. In: *Proceedings of the 34th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, 2017, pp. 3751–3760.

[28] X. Zhang et al. "Safe and Near-Optimal Policy Learning for Model Predictive Control using Primal-Dual Neural Networks". In: *2019 American Control Conference (ACC)*. ISSN: 2378-5861. 2019, pp. 354–359.

[29] Q. Cappart et al. "Combinatorial Optimization and Reasoning with Graph Neural Networks". In: *Journal of Machine Learning Research* vol. 24, no. 130 (2023), pp. 1–61.

[30] Z. Chen et al. *Expressive Power of Graph Neural Networks for (Mixed-Integer) Quadratic Programs*. arXiv:2406.05938. 2024.

[31] D. Arnström. "Real-Time Certified MPC : Reliable Active-Set QP Solvers". eng. In: (2023). Publisher: Linköping University Electronic Press.

[32] H. Jung et al. *Learning context-aware adaptive solvers to accelerate quadratic programming*. arXiv:2211.12443. 2022.

[33] S. Boyd and L. Vandenberghe. *Convex Optimization*. en. ISBN: 9780511804441 Publisher: Cambridge University Press. Cambridge University Press, 2004.

[34] W. S. Dorn. "Duality in quadratic programming". en. In: *Quart. Appl. Math.* vol. 18, no. 2 (1960), pp. 155–162.

[35] M. J. Best. "Equivalence of some quadratic programming algorithms". en. In: *Mathematical Programming* vol. 30, no. 1 (1984). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer-Verlag, pp. 71–87.

[36] R. A. Bartlett and L. T. Biegler. "QPSchur: A dual, active-set, Schur-complement method for large-scale and structured convex quadratic programming". en. In: *Optim Eng* vol. 7, no. 1 (2006). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Kluwer Academic Publishers, pp. 5–32.

[37] D. Arnström et al. "A Dual Active-Set Solver for Embedded Quadratic Programming Using Recursive LDL$^{\mathrm{T}}$ Updates". In: *IEEE Transactions on Automatic Control* vol. 67, no. 8 (2022). Conference Name: IEEE Transactions on Automatic Control, pp. 4362–4369.

[38] P. Otta et al. "Measured-state driven warm-start strategy for linear MPC". In: *2015 European Control Conference (ECC)*. 2015, pp. 3132–3136.

[39] F. Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* vol. 20, no. 1 (2009). Conference Name: IEEE Transactions on Neural Networks, pp. 61–80.

[40] M. Gori et al. "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. ISSN: 2161-4407. 2005, 729–734 vol. 2.

[41] M. Malekzadeh et al. "Review of Graph Neural Network in Text Classification". In: *2021 IEEE 12th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. 2021, pp. 0084–0091.

[42] P. Battaglia et al. "Interaction Networks for Learning about Objects, Relations and Physics". In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016.

[43] T. Kipf et al. "Neural Relational Inference for Interacting Systems". en. In: *Proceedings of the 35th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, 2018, pp. 2688–2697.

[44] A. Fout et al. "Protein Interface Prediction using Graph Convolutional Networks". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[45] X.-M. Zhang et al. "Graph Neural Networks and Their Current Applications in Bioinformatics". English. In: *Front. Genet.* vol. 12 (2021). Publisher: Frontiers.

[46] J. Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI Open* vol. 1 (2020), pp. 57–81.

[47] J. Sjölund and M. Bånkestad. *Graph-based Neural Acceleration for Nonnegative Matrix Factorization*. arXiv:2202.00264. 2022.

[48] P. Häusner et al. *Neural incomplete factorization: learning preconditioners for the conjugate gradient method*. arXiv:2305.16368. 2024.

[49] P. Häusner et al. "Learning incomplete factorization preconditioners for GMRES". en. In: *Proceedings of the 6th Northern Lights Deep Learning Conference (NLDL)*. ISSN: 2640-3498. PMLR, 2025, pp. 85–99.

[50] P. W. Battaglia et al. *Relational inductive biases, deep learning, and graph networks*. arXiv:1806.01261. 2018.

[51] M. M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. arXiv:2104.13478. 2021.

[52] C. van de Panne and A. Whinston. "The Simplex and the Dual Method for Quadratic Programming". In: *Journal of the Operational Research Society* vol. 15, no. 4 (1964). Publisher: Taylor & Francis _eprint: https://doi.org/10.1057/jors.1964.60, pp. 355–388.

[53] L.-I. Tsui et al. "NG-DTA: Drug-target affinity prediction with n-gram molecular graphs". In: *2023 45th Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*. ISSN: 2694-0604. 2023, pp. 1–4.

[54] E. Ranjan et al. *ASAP: Adaptive Structure Aware Pooling for Learning Hierarchical Graph Representations*. arXiv:1911.07979. 2020.