Abstract

Achieving fusion of deep learning with combinatorial algorithms promises transformative changes to AI. Creating an impact in a real-world setting requires AI techniques to span a pipeline from data, to predictive models, to decisions. Aligning these components together requires careful consideration, as having these components trained separately does not account for the end goal of the model. This work surveys general frameworks for End-to-End optimization learning; we focus on the integration of optimization methods within machine learning architectures. We discuss challenges and limitations associated with these methods and propose a novel approach to overcome the bottlenecks that arise.

End-to-End Decision Focused Learning using Learned Solvers

by

Yehya Marwan Farhat

B.S., American University of Beirut 2020

Thesis
Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Syracuse University
May 2023

Acknowledgment

I would like to take this opportunity to express gratitude to all those who have contributed to the completion of this thesis and my journey throughout the degree. I would like to express my sincere appreciation to my advisor, Dr. Ferdinando Fioretto, for creating an exceptional research environment and fostering a collaborative atmosphere. Under their guidance, I have had the privilege to work with a team of talented individuals. In particular, I would like to extend a special thank you to James Kotary. His expertise, insights, and collaborative spirit have significantly contributed to the success of this project.
I would like to express my deepest gratitude to my uncle, Hassan Farhat, for his unwavering support and for the countless opportunities throughout my life. To my Dearest grandmother, mother, and siblings, your sacrifices, selflessness, and love have shaped me into the person I am today. All four of you are my guiding compass and your influence will forever shape my journey.
To my lifelong friend, Amir Hamad, thank you for the countless conversations we've shared and the constant encouragement.
To the rest of my beloved family and friends, thank you for your constant love and support.

In loving memory of Marwan Farhat

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constrained optimization (CO), a critical component of optimization theory, has significantly advanced various industrial and societal sectors. Its applications span an array of fields, including supply chain management, energy scheduling, transportation, and finance, among others. As one of the most rapidly expanding subfields in mathematics, mathematical optimization is distinguished by its versatility and widespread applicability in solving complex problems across numerous domains.

The modern history of CO goes back to the Second World War. Operations research started out as an initiative to use mathematics and computer science to assist military planners with decisions. This was also part of a bigger initiative during this time, where scientific research in general took the forefront of the war.During this period, history witnessed the development of *quantum mechanics*, a mathematical framework to explain atomic structures and their properties. This then led to the successful development of the atomic bomb. Along with these significant scientific advances came the overshadowed development of the *simplex algorithm*. The simplex algorithm was developed by an American mathematician, George Dantzig, while working at the RAND Corporation. Dantzig was tasked by the U.S. Air Force to help optimize the use of its resources. Dantzig realized that this problem can be formulated as a linear programming problem, which involves maximizing or minimizing a linear function subject to a set of linear constraints. The

development of the simplex algorithm revolutionized the field of CO, making it possible to solve large-scale linear programming problems with hundreds or even thousands of variables and constraints. This led to the widespread adoption of CO techniques in a variety of fields. Following the development of the simplex algorithm was the development of further CO techniques concerned with nonlinear problems, such as the gradient descent algorithm.

Given the boom of CO and computers, the usage of solvers became the next logical step to tackle CO problems. Solvers usually employ various algorithms to explore the feasible region of the problem and different solvers are designed for different types of optimization problems. The availability of algorithms to solve CO problems heavily depends on their form. Some problems can be efficiently and reliably solved, while others have been proven to have no efficient solutions. Constrained optimization problems are particularly important in many fields.

Some CO problems are characterized by their discrete state spaces, and their optimal solutions often involve combinatorial structures, such as selecting subset permutations. These permutations can represent paths through a network or other discrete structures that represent a set of optimal decisions. Such problems are notoriously difficult and often fall into the category of NP-hard problems.

Despite their complexity, many CO problems are routinely solved using a wide spectrum of techniques developed by the AI and operational research communities. These approaches typically exploit the problem structure to accurately solve the problem within a reasonable time frame. However, the complexity of CO problems remains a significant barrier when trying to adopt them in contexts that require repeated decision-making or in time-sensitive settings where real-time decisions need to be made, such as multi-year planning studies or expensive simulations.

To address the challenges posed by complex CO problems, machine learning (ML) methods have emerged as a promising solution, particularly given the existing relationship between these two fields. Since optimization is at the core of ML, most machine

learning algorithms use an optimization model to learn the parameters in the objective function from available data. In today's era of vast data, numerical optimization methods significantly affect the effectiveness and efficiency of machine learning models. As machine learning has gained prominence in recent years, it has pushed the boundaries of optimization theory since many machine learning techniques rely on optimization algorithms. For instance, the backpropagation algorithm utilized to train neural networks is designed to minimize a function, which involves minimizing the difference between the predicted output and the actual value (i.e., the target). The intersection of ML and CO has gained traction and become increasingly popular over recent years.

Current research areas in the intersection of CO and ML are categorized into two main directions: ML-augmented CO and End-to-End CO learning (E2E CO). The former refers to the use of ML techniques to enhance the performance of traditional optimization methods, while the latter refers to entire systems that are trained end-to-end, meaning that the optimization and learning components are integrated into a single system that is trained jointly.

The focus of this study is on E2E-constrained optimization learning. We will explore some of the popular techniques in the field and propose a novel approach to attempt to tackle some of the main challenges. A fundamental problem in computer science and optimization is the *shortest path* problem. The problem involves finding the shortest path between two points in a graph, where the edges of the graph have a weight or cost associated with them. The shortest path problem has many important applications due to the nature of what graphs are able to model. Many real-world scenarios are modeled as graphs. Formally, a graph G is an ordered pair defined as $G = (V, E)$, where V is a set of vertices or nodes denoted as $\{v_1, v_2, \ldots, v_n\}$, and E is a set of edges denoted as $\{e_1, e_2, \ldots, e_n\}$. Usually, each edge e_i has an associated weight or cost $w(e_i)$. Important applications of the shortest path problem include routing, logistics, resource allocation, circuit design, and navigation. In general, the shortest path problem is important because it provides a foundation for many optimization and decision-making problems in various fields. There are well-known techniques for solving the shortest path problem,

including the famous Dijkstra's, Bellman-Ford, and Floyd-Warshall algorithms. However, some variations of the shortest path problem, such as the 3D shortest-path problem in a polyhedral environment, are NP-hard. They suffer the same fate as other combinatorial NP-hard problems—they are not efficiently solvable. Hence, it is a natural candidate for this study.

The aim of this study is to address the following questions:

1. Is it possible to solve the shortest path problem using some of the conventional E2E Learning methods?

2. Can our proposed approach overcome the bottlenecks induced by traditional methods?

3. How well do all these solutions compare with one another?

Chapter 2

literature review

2.1 Constrained Optimization

Constrained optimization (CO) problems pose the task of optimizing an *objective function* $f : \mathbb{R}^n \to \mathbb{R}$ with respect to some variables $\boldsymbol{z} \in \mathbb{R}^n$ subject to a set of *constraints* $\boldsymbol{C}$ on those variables. The objective function is either a cost function that we need to minimize or a utility function that we need to maximize:

$$\mathcal{O} = \underset{z}{argmin/argmax}\ f(z) \quad subject\ to\ z \in C$$

An assignment of $\boldsymbol{z}$ that satisfies $\boldsymbol{C}$ is called a *feasible solution*; additionally for a minimization problem, if $f(z) \leq f(w)$ for all feasible w, it is called an *optimal solution* or $f(z) \geq f(w)$ for a maximization problem. A well-understood class of optimization problems are *convex* problems, those in which the constraint set $\boldsymbol{C}$ is a convex set and the objective f is a convex function. Convex problems are a favorable class of optimization problems due to them being efficiently solvable with strong theoretical guarantees on the existence and uniqueness of solutions.

A common constraint set that arises in many problems is of the following form: $\boldsymbol{C} = \{z : \mathbf{A}z \leq \mathbf{b}\}$, where $\mathbf{A} \in \mathcal{R}^{m \times n}$ and $\mathbf{b} \in \mathcal{R}^m$. in this case the set $\boldsymbol{C}$ is convex.

2.1.1 Convex Optimization Problems

Convex optimization problems are problems where all of the constraints are convex functions, and the objective function is a convex function if minimizing, or a concave function if maximizing. The key takeaway from having convexity properties is that there can only be one optimal solution, which is globally optimal. Many classes of convex optimization problems admit *polynomial-time* algorithms. On the other hand, nonconvex problems do not have these nice properties; they may have multiple feasible regions and multiple locally optimal points within each region. These problems, in general, are NP-hard and require more sophisticated techniques to solve efficiently.

2.1.1.1 Convex Sets

A set C is convex if the line segment between any two points in C lies entirely within C. More formally, $\forall x_1, x_2 \in C, and\, \forall \theta \in [0, 1]$ the below condition is satisfied [15]

$$\theta x_1 + (1 - \theta)x_2 \in C$$

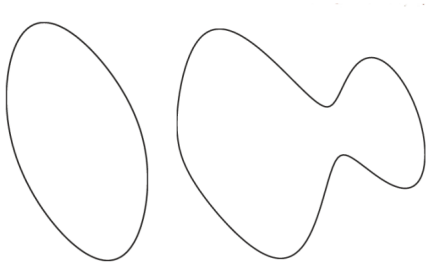


Figure 2.1: Example of a convex set (left) and a non-convex set (right)

2.1.1.2 Convex Functions

A function is called convex if the function is defined over a convex domain such that, for any two disjoint points on the graph, the line segment connecting both these points lies above the graph, as in figure 2.2 [15].

Formally, a function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if $dom(f)$ is a convex set and if $\forall x, y \in dom(f)$ *and*, $\forall \theta \in [0, 1]$ the below condition is satisfied

$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta)f(y)$$

Figure 2.2: Convex Function

The convexity of a function is also closely related to the concept of a convex set; a function f is convex if and only if its *epigraph* (the set of all points above the function graph) is a convex set.

The epigraph of a function $f : \mathbb{R}^n \to \mathbb{R}$ can be defined as follows; the set of points $epi(f) = \{(x, t)|x \in dom(f), t \geq f(x)\}$

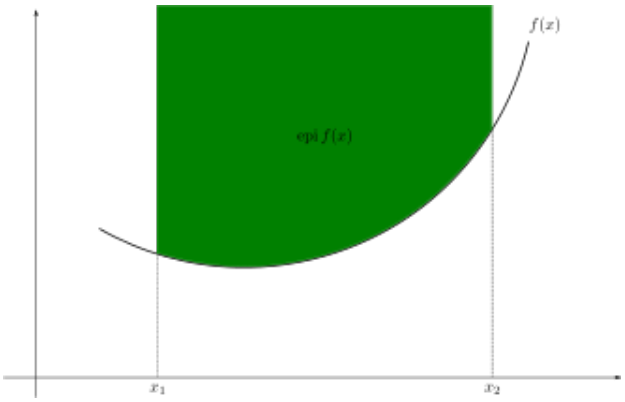


Figure 2.3: Epigraph of a function f

2.1.2 Linear Programs

If the objective function f is an affine function, the problem is called a *linear program* (LP).

Maximize (or Minimize) $f(x) = c^T x$

Subject to:

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$

$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$

$\vdots$

$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$

$x_1, x_2, \ldots, x_n \geq 0$

We can view these problems geometrically. An LP is to minimize/maximize a linear function over a polyhedron. in other terms this means "going as far as possible in the direction -c or c", where c is the vector that defines the (linear) objective function.

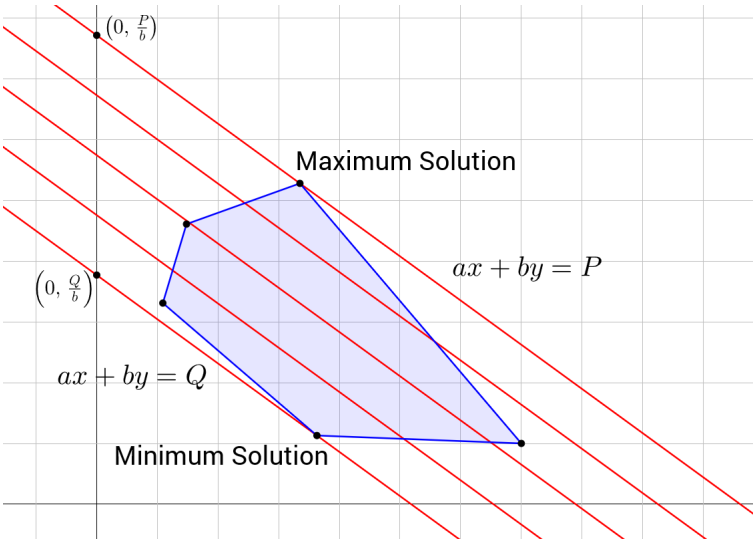


Figure 2.4: LP optimization example [3]

2.1.3 Quadratic programs

If the optimization includes a quadratic objective function rather than a linear one, the resulting problem is called a *quadratic program* (QP).

Maximize (or Minimize) $f(x) = x^T Q x + c^T x$

Subject to:

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$

$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$

$\vdots$

$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$

$x_1, x_2, \ldots, x_n \geq 0$

The geometrical visualization of a QP is to minimize a convex quadratic function over a polyhedron.

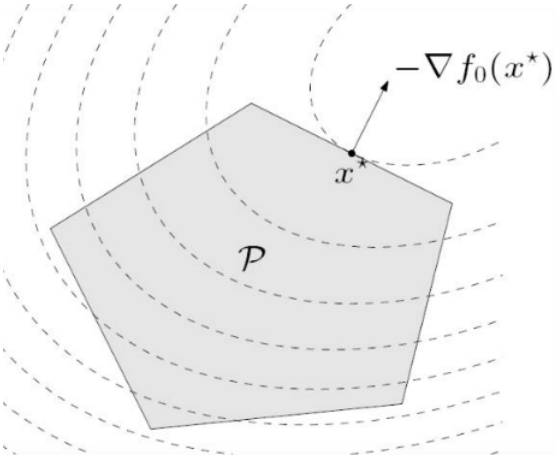


Figure 2.5: QP optimization example [5]

2.1.4 Mixed integer Programs

Some subset of problems require their variables to take integer values. Such problems are called *mixed integer programs* (MIP). While LP and QP with convex objectives belong to the class of convex problems, the introduction of integral constraints on x ($x \in \mathbb{N}^n$) results in a nonconvex set, the feasible set in a MIP consists of distinct, disjoint points in $x \in \mathbb{R}^n$. This produces a much more difficult problem to solve, generally NP-hard.

$f(x) = c^T x$

Subject to:

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$

$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$

$\vdots$

$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$

$x_1, x_2, \ldots, x_n \geq 0$

$x_1, x_2, \ldots, x_n \in \mathbb{N}^n$

The geometrical visualization of a MIP depends on the type of problem, but we can have a linear program with integer constraints (such a problem is actually called a mixed integer linear program, which will be explored further in the next section). Such optimization problems will look exactly like an LP but the set is a set of disjoint integral points.

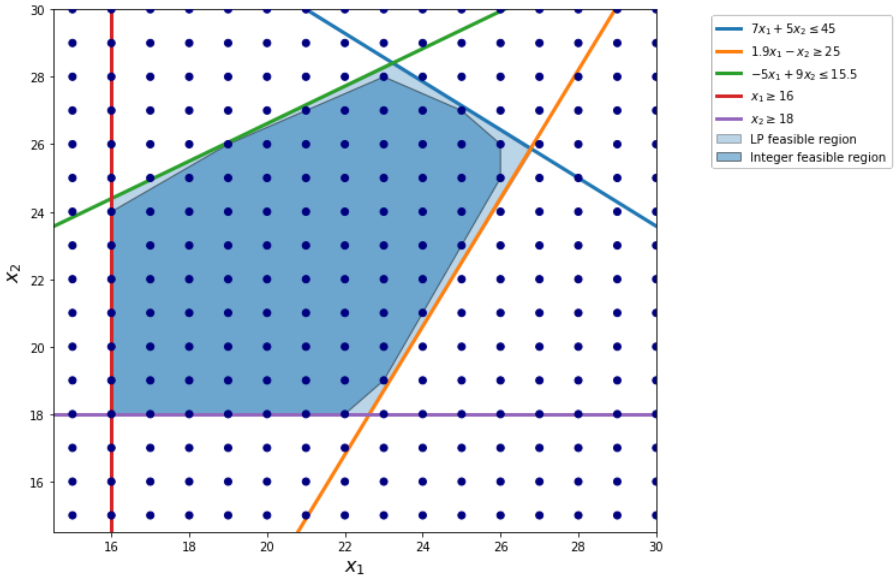


Figure 2.6: MIP optimization example [37]

2.1.5 Mixed Integer linear programs

Mixed integer linear programs (MILP) are a subset of MIPs. MILPs are of particular interest to us in this work, they are linear programs with integer constraints on x. Both maxflow and shortest path are modeled as MILP. the general formulation of a MILP can be found in the MIP section along with its geometrical visualization, please refer to figure 2.6

In general optimization problems are a huge class of mathematical problems intersecting all fields of science.

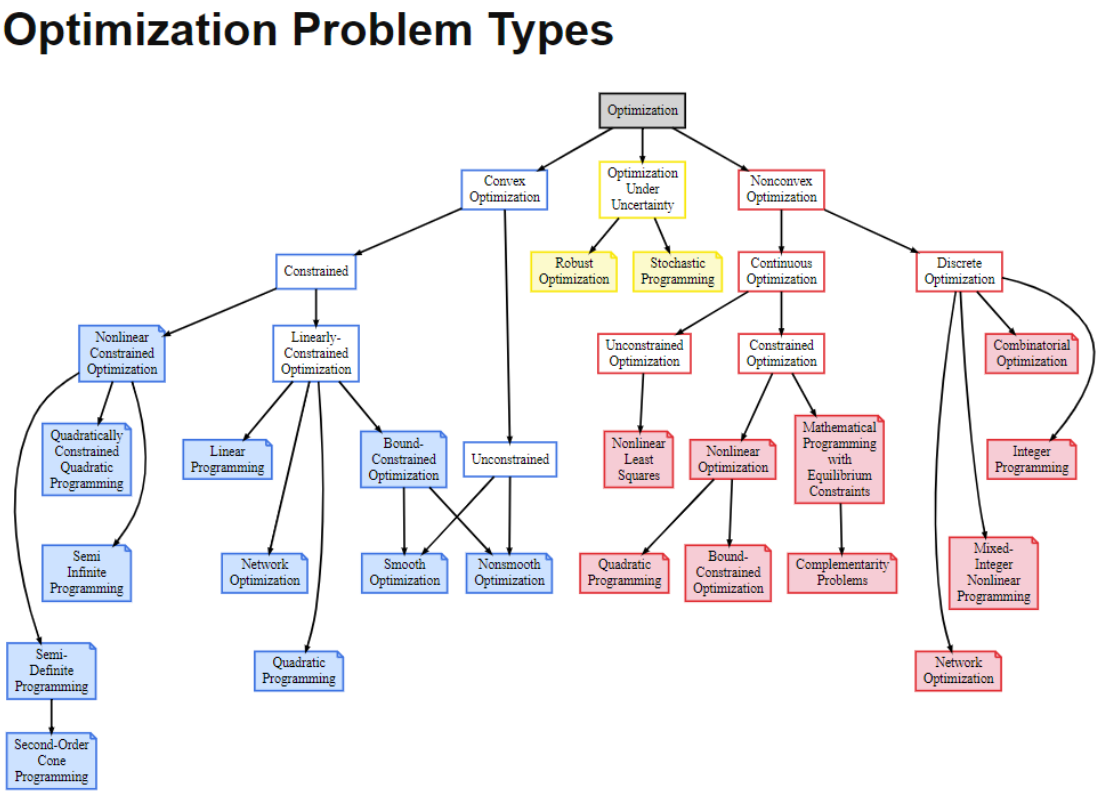


Figure 2.7: optimization classes

[25]

2.1.6 Optimization solving methods

A well-developed theory exists for solving convex optimization problems. Convex problems are known to be efficiently solvable and fall in the class of **P**. Some notable methods for solving these problems are *Augmented Lagrangian* methods [26], *simplex* methods [6], and *interior point* methods [29].

2.1.6.1 Branch and Bound

MILPs require a different approach, as they are generally NP-hard problems, and the above methods do not guarantee solutions. One notable method for MILPs is the *branch and bound* framework. Branch and bound combines optimization and searching. Searching is represented by a search tree in which an LP *relaxation* of the MILP is formed at

each node of the search tree and solved using any LP solving method (e.g., simplex). The branch and bound algorithm starts by solving a relaxed version of the MILP problem to obtain a lower bound on the optimal solution. The algorithm selects a value that is not an integer in the solution and branches according to that variable. Branching creates two sub-problems, one where the selected variable is fixed to its upper bound and the other fixed to its lower bound. That is, one node will have the following constraint $x_i \geq a_i$, while the other will have $x_i < a_i$, where x_i is a variable with fractional value a_i. These problems are recursively solved until all regions of the search space have been explored or a solution is found.

The following is the skeleton of a generic branch and bound algorithm for minimizing an objective function *f*. In order to develop an operational algorithm using the approach below, we need to specify a bounding function called 'bound' to calculate the lower limits of f for the various nodes of our search tree, along with a branching rule. Both of these specifications are usually problem-specific.

Several different queue data structures can be used here. The FIFO queue-based implementation will yield a *breadth-first search* algorithm, while a LIFO-queue (stack) will yield a *depth-first algorithm*. A *best-first* branch and bound algorithm can also be obtained by using a priority queue. [34]

Algorithm 1: Generic Branch and Bound

1. Using a heuristic, find a solution x_h to the optimization problem. Store its value, $B = f(x_h)$. (If no heuristic is available, set B to infinity.) B will denote the best solution found so far and will be used as an upper bound on candidate solutions;
2. Initialize a queue to hold a partial solution with none of the variables of the problem assigned;
3. **while** *queue is not empty* **do**
   - Take a node N off the queue;
   - **if** *N represents a single candidate solution x and* $f(x) < B$ **then**
     - x is the best solution so far. Record it and set B $\leftarrow$ f(x)
   - **else**
     - branch on N to produce new nodes N_i;
     - **for** *each* N_i **do**
       - **if** *bound*($N_i > B$) **then**
         - do nothing; since the lower bound is greater than the upper bound of the problem and will never lead to an optimal solution
       - **else**
         - store N_i on the queue
       - **end**
     - **end**
   - **end**

end

2.1.7 Relaxation of constrained optimization problems

A relaxation is a modeling strategy in mathematical optimization where a difficult problem is approximated by a slightly easier one. This "easier" problem usually involves removing or modifying certain constraints from the original problem. This approximation is referred to as a relaxation, and solving it provides valuable insights into the original problem.

For example, consider a situation where we are dealing with a mixed-integer linear programming (MILP) problem. In this case, we can employ a linear programming relaxation to eliminate the integrality constraint, thereby allowing non-integer rational solutions. This transformation converts the problem into a linear program, which is generally easier to solve compared to the original MILP. Similarly, other types of relaxations can be applied to complex problems, such as *Lagrangian relaxation* and *Convex Relaxation* these relaxations can also be used to either supplement the branch and bound algorithm or complement it by using them to obtain bounds when using the branch and bound

algorithm. There are a lot of relaxation techniques and the choice of which technique to use depends on the specific problem at hand and the optimization tools at our disposal.

In general two main properties have to hold to provide a valid relaxation

1. The original problems feasible domain is a subset of the relaxed problems feasible domain.

2. The relaxed problem should provide a bound on the original problem.

2.1.7.1 Lagrangian Relaxation

Lagrangian relaxation involves penalizing violations of inequality constraints using *Lagrangian multipliers* to penalize violations of these constraints by integrating the constraint into the objective function. This means that instead of strictly enforcing the constraints, the problem is modified to include additional costs for violating them. This relaxed problem is often easier to solve than the original problem.

Suppose we are given the following problem

$max\ c^T x$

Subject to:

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$

$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$

$\vdots$

$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$

$x \in \mathbb{R}^n\ A \in \mathbb{R}^{mxn}$

We can split the constraints into two parts, the first half of the constraints will look like the following

$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$

$\vdots$

$a_{\frac{m1}{2}}x_1 + a_{\frac{m2}{2}}x_2 + \cdots + a_{\frac{mn}{2}}x_n \leq b_{\frac{m}{2}}$

Label the first half of the constraints as $A_1 x \in b_1$

while the second half of the constraints will be labeled as $A_2 x \in b_2$ and will look like the

following

$a_{\frac{m1}{2}+1}x_1 + a_{\frac{m2}{2}+1}x_2 + \cdots + a_{\frac{mn}{2}+1}x_n \leq b_{\frac{m}{2}+1}$

$\vdots$

$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m$

After splitting the constraints the new equivalent problem will look as follows

$max\ c^T x$

Subject to:

$A_1 x \leq b_1$

$A_2 x \leq b_2$

We may then introduce one of the constraints into the objective to obtain the following problem

$max\ c^T x + \lambda^T (b_2 - A_2 x)$

Subject to:

$A_1 x \leq b_1$

$\lambda = (\lambda_1, ..., \lambda_{m2})$

The above relaxed problem solution is a bound on the original problem, for any positive semi-definite matrix $\tilde{\lambda}$ the optimal result of the lagrangian relaxation problem will be no smaller than the optimal result of the original problem. Let x^* be the optimal solution to the original problem, and let $\tilde{x}$ be the optimal solution to the relaxation. We can then say that

$$c^T x^* \leq c^T x^* + \tilde{\lambda}^T (b_2 - A_2 x^*) \leq c^T \tilde{x} + \tilde{\lambda}^T (b_2 - A_2 \tilde{\lambda})$$

The first inequality holds because x^* is feasible in the original problem and the second inequality holds because $\tilde{x}$ is the optimal solution to the Lagrangian relaxation [35].

2.1.8 Duality

Duality theory provides a powerful tool for analyzing and solving optimization problems, particularly when the primal problem is challenging to solve directly. It is important to

note that the dual problem may not always be easier to solve than the primal; in some cases, it may even be harder. However, the dual problem can provide valuable information about the primal problem, such as bounds on the optimal value, identification of near-optimal solutions, and proofs of optimality. These insights can be helpful in solving the primal problem or understanding its properties.

Duality offers a deeper understanding of the optimization problem by considering it from two different perspectives. It allows us to establish connections between the primal and dual problems, exploit the relationships between their solutions, and derive meaningful conclusions. While strong duality, where the primal and dual have the same optimal solutions, is desirable, it is not always guaranteed. Weak duality, on the other hand, always holds and provides lower and upper bounds on the optimal values of the primal and dual problems, respectively.

Overall, duality theory is a valuable tool in optimization, providing insights and techniques to tackle challenging problems and gain a better understanding of their properties.

2.1.8.1 Lagrangian Duality

Setting up the dual problem is done as follows, and is referred to as lagrangian duality. Consider the following optimization problem

$$\mathcal{O} = \underset{y}{argmin} f(y) \;\; subject\, to: \;\; g_i(y) \leq 0 \; (\forall_i \in [m])$$

When relaxing the problem using Lagrangian relaxation one can either relax some or all the problem constraints into the objective function, each constraint will have a Lagrangian multiplier to penalize any constraint violation.

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m} \lambda_i g_i(y)$$

The λ terms describe the lagrangian multipliers, $\lambda = (\lambda_1, ..., \lambda_m)$ denotes the vector of all the multipliers. In this formulation, the constraints $g(y)$ may take up negative

numbers. This formulation can be augmented to enforce positivity on the constraints as follows.

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m} \lambda_i max(0, g_i(y))$$

After applying the augmented lagrangian the optimization problem becomes.

$$LR_\lambda = \underset{y}{argmin} f_\lambda(y)$$

The above function will satisfy $f(LR_\lambda) \leq f(\mathcal{O})$, this follows from the fact that the relaxation of an optimization problem is a lower bound on the original problem. The Lagrangian dual can then be formulated to obtain the best Lagrangian multipliers and give us a tighter lower bound.

$$LD = \underset{\lambda \geq 0}{argmax}\ f(LR_\lambda)$$

For various problems, the lagrangian dual provides a good approximation of $\mathcal{O}$

2.2 Deep Learning

Supervised deep learning can be viewed as the task of approximating a non-linear mapping from a sample of targeted data. In other words, the task can be viewed as approximating $f(x)$ given a set of data samples x and y. Deep Neural Networks (DNNs) are deep learning architectures composed of a sequence of layers, each typically taking as inputs the result of the previous layer. DNNs are conventionally feed-forward neural networks, where the layers are fully connected and the function connecting the layer is given by $o = \pi(\mathbf{W}x + \mathbf{b})$. Where $x \in \mathbb{R}^n$ is the input vector, $o \in \mathbb{R}^m$ is the output vector, $\mathbf{W} \in \mathbb{R}^{mxn}$ a matrix of weights, and $\mathbf{b} \in \mathbb{R}^m$ a bias vector. The function $\pi(.)$ is often non-linear (e.g., ReLu, Sigmoid) [18].

The supervised learning task can be enumerated as follows, consider the dataset $X = \{x_i, y_i\}_{i=1}^n$ consisting of N data points. $x_i \in \mathcal{X}$ being the input vector and $y_i \in \mathcal{Y}$ is the respective target vector. The goal of the learning task is to learn a model mapping

$\mathcal{M}_\theta : \mathcal{X} \to \mathcal{Y}$, where θ is a vector of real-valued parameters. The quality of this learned mapping is measured in terms of a non-negative, and differentiable, loss function. $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$. The non-negativity and differentiability of the loss function are of great importance in this work and will be explored further in the coming sections. We can view the entire learning task as minimizing the empirical risk function (ERM):

$$\min_\theta J(\mathcal{M}_\theta, X) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\mathcal{M}_\theta(x_i), y_i)$$

.

All methods and techniques used in this work are DNNs whose training conforms to the objective above. Other notable classes of deep learning methods that have been used to solve constrained optimization problems are sequence models (e.g., RNNs, LSTMs), Graph Neural Networks (GNNs), and Reinforcement Learning (RL).

2.3 ML and CO

Current research areas in the synthesis of constrained optimization and machine learning can be categorized into two main branches: *ML-Augmented-CO* which focuses on using ML to aid the performance of CO solvers, and *End-to-End CO learning* (E2E-COL) which focuses on integrating combinatorial solver or optimization methods into deep learning architectures. E2E-COL learning can be further categorized into two branches: (1) Approaches that develop ML architectures to predict fast, approximate solutions to predefined CO problems and (2) Approaches that exploit CO solvers as neural network layers for the purpose of structured logical inference, referred to here as the *Predict-and-Optimize* paradigm. The predict and optimize paradigm is what our project is based on and will be explored further in the coming few sections. Readers can refer to figure 2.8 for a nice visualization of the different research branches in ML and CO [18].

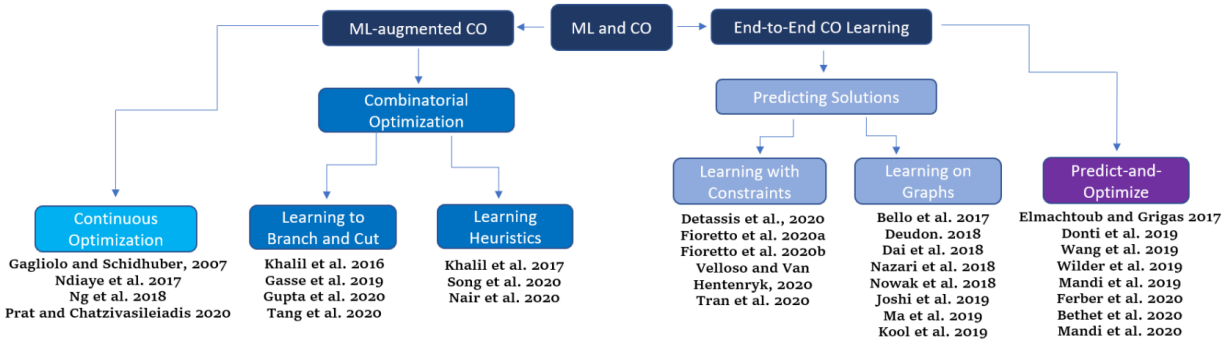


Figure 2.8: Branches of Machine Learning and Constrained Optimization

2.3.1 ML-augmented CO

ML-augmented CO involves the augmentation of CO solvers, this area draws from both supervised and reinforcement learning to develop highly optimized and efficient approaches to various aspects of CO solving. In the context of combinatorial optimization where branch and bound is the general framework used to solve these problems as it guarantees optimality. The methods can be classified as (1) Methods that learn to guide the search decision in branch and bound, and (2) Methods that guide the application or usage of the *primal heuristics* within the branch and bound solvers. Some notable techniques in (1) include neural networks that emulate expensive branching rules for MIPs [20] [16] [14], and learning to cut when using cutting plane methods [30]. On the other hand, some notable techniques in (2) include predictions of the most effective nodes at which to apply the primal heuristic [10], and optimal choice of variable partitions in *large neighborhood search* (LNS) paradigm, which iteratively chooses a subset of variables to optimize while leaving the remaining variables fixed [28]. Such ML techniques have also been applied to continuous CO problems. Some notable mentions include learning active constraints to reduce problem size before feeding into CO solvers [27] [23], and learning rules to ignore optimization variables leading to faster solutions [22]. For a thorough and deep dive into ML-augmented CO, the reader is referred to the following survey [2].

2.3.2 End-to-End COL: Predicting CO Solutions

As with the traditional approaches to most machine learning paradigms, ML architectures are focused on developing ML architectures to predict fast, approximate solutions to predefined problems by learning from a given data set. These E2E COL architectures are no different, we care about approximating fast solutions to CO problems without the use of CO solvers at the time of inference. These approaches contrast with what we discussed in the above section, instead of using ML to augment and direct the solver we take a more traditional ML training approach and encapsulate the solver itself as a black box i.e a trained neural network, this is done by observing a set of solved instances of the CO problem at hand. The literature on Predicting CO solutions can be categorized as 2 methods, the first being *learning with constraints*, which involves incorporating constraints into end-to-end learning for predicting optimal solutions, and *learning solutions on graphs*, which involves producing output as combinatorial structures from variable sized inputs.

2.3.2.1 Learning with Constraints

Consider the following dataset, let $X = \{x_i, y_i\}_{i=1}^{n}$, where x_i describe the inputs for a problem instance, and y_i describes a complete solution to the problem P with its respective input x_i. Each sample might specify a different instance of the problem, namely with a varying objective function, coefficients, and constraints.

A very early approach to the use of predicting CO problem solutions was done by using Hopfield networks with modified energy functions to solve the *traveling salesman problem* (TSP) [17]. The problem by Hopfield and Tank was solved by first considering an nxn matrix M whose i_{th} row describes the i_{th} city's location while each column represents the ordering of the tour. While considering the solution of this TSP by a Hopfield network, every node in the network corresponds to one element in the matrix. So for example for a 5-city problem instance, we would have a total of 25 neurons and

a $5x5$ matrix. Energy functions are then used to emulate the objective of the TSP and use Lagrangian multipliers to penalize constraint violations, satisfying constraints, and outputting optimized solutions require the energy function to be at a minimum. This technique eventually fell out of favor due to its weakness in significantly depending on the initial state of the network. Parallel to Hopfield networks was the use of *deformable template models* such as the development of Elastic Net, and Self Organizing Map. However, addressing the limitations of these models has been a central focus in subsequent research [4] [11] [31]. Despite their appealing properties, these neural networks have not yielded satisfactory results compared to algorithmic methods when carefully benchmarked. This eventually made way for superior frameworks that exploit *Lagrangian duality* to guide the prediction and satisfy the problem constraints. Other end-to-to learning approaches have demonstrated success by injecting information about constraints from targeted feasible solutions, one notable method presented in [7] uses an iterative process of using external solvers to adjust targeted solutions to better align with model predictions, while still maintaining feasibility, and reducing constraint violations in subsequent iterations, this method essentially teaches the model how to satisfy constraints by using state-of-the-art solvers. A decomposition schema is used, alternating master steps, which is in charge of enforcing the constraints, and learner step, which is where the normal supervised ML model training takes place.

2.3.2.1.1 Lagrangian dual framework

The *Lagrangian dual framework* (LDF) [13] is based on integrating the learning task or the loss function with the augmented Lagrangian to obtain an approximation of the problem, after which we can compute the Lagrangian dual to compute the optimal Lagrangian multipliers and obtain an even tighter approximation.

Given a problem defined as

$$\mathcal{O}(d) = \underset{y}{argmin} f(y, d) \; subject \; to \; g_j(y, d) \leq 0 \; (\forall_j \in [m])$$

with the following set of samples (dataset) $D = \{(d_i, y_i)\}_{i=1}^{n}$, parametric model $\mathcal{M}[w]$ with weights w and a loss function $\mathcal{L}$ then the learning task can be stated as follows

$$w^* = \underset{w}{argmin} \sum_{i=1}^{n} \mathcal{L}(M[w](d_i), y_i)$$

$$subject\ to\ \ g_j(M[w](d_i), d_i) \leq 0\ (\forall_j \in [m], i \in [n])$$ to obtain the approximation $\hat{\mathcal{O}} = M[w^*]\ \ of\ \ \mathcal{O}$

attempting to solve this learning task can be quite difficult, not only do we have to find a set of weights w to minimize our loss function, but the weights need to be chosen such that constraints are satisfied for all the samples. A naive approach to solving this problem will result in predictors that significantly violate the constraints, leading to a useless model because it cannot be used in practice.

To deal with the above challenge the LDF is used, the LDF will exploit the Lagrangian dual method to solve the problem while satisfying its constraints. The new learning task will look as follows, given a set of Lagrangian multipliers $\lambda = (\lambda_1, ..., \lambda_m)$ we will have the following loss function

$$\mathcal{L}_\lambda(\hat{y}_i, y_i, d_i) = \mathcal{L}(\hat{y}_l, y_l) + \sum_{j=1}^{m} \lambda_j v(g_j(\hat{y}_i, d_i))$$

where the function $v(.)$ encapsulates how you want to deal with negative violations (i.e, using max(0, g(.))), and where $\hat{y}_i = M[w](d_i)$ is the prediction of the model. Solving the task then becomes

$$w^*(\lambda) = \underset{w}{argmin} \sum_{i=1}^{n} \mathcal{L}_\lambda(\hat{y}_i, y_i, d_i)$$

this will produce an approximation $\hat{\mathcal{O}}_\lambda = M[w^*(\lambda)]$ of $\mathcal{O}$. The Lagrangian dual then computes the optimal multipliers, that is

$$\lambda^* = \underset{\lambda}{argmax}\ \underset{w}{min} \sum_{i=1}^{n} \mathcal{L}_\lambda(\hat{y}_i, y_i, d_i)$$

to obtain the strongest Lagrangian relaxation of $\mathcal{O}$, i.e, $\hat{\mathcal{O}}^* = M[w^*(\lambda^*)]$ Learning $\hat{\mathcal{O}}^*$ depends on an iterative schema, this schema is summarized in the below algorithm. Given the input dataset D, the optimizer step α, and a Lagrangian step size s_k. The Lagrangian multipliers are initialized in line 1 and the training is performed for a fixed number of epochs. At each epoch, k optimizes the model weights

Algorithm 2: LDF for constrained optimization problems

input : $D = (d_i, y_i)_{i=1}^n$;
$\alpha, s = (s_o, s_1, ...)$;
$\lambda_j^0 \leftarrow 0 \, \forall_j \in [m]$;
for *epoch* k **do**
  forall $(y_i, d_i) \in D$ **do**
    $\hat{y}_i \leftarrow M[w, \lambda^k](d_i)$;
    $w \leftarrow w - \alpha \nabla_w \mathcal{L}_{\lambda^k}(\hat{y}_i, y_i, d_i)$;
  end
  $\lambda_j^{k+1} \leftarrow \lambda_j^k + s_k \sum_{i=1}^n v_j(g_j(\hat{y}_i, d_i)) \, \forall_j \in [m]$;
end

2.3.2.2 Learning Solution on Graphs

In contrast to the above section where solutions are being learned on unstructured CO problems, simply by having traces of a solved CO problem a variety of methods learn to solve CO problems represented on graphs. Deep learning architectures such as transformers [32] and GNNs [21] have proven to be effective tools in addressing these tasks

The authors of this paper [33] proposed a pointer network that utilizes an encoder-decoder architecture and an attention mechanism to produce permutations over inputs of variable size. They identified certain constrained optimization (CO) problems as a suitable application for their architecture and tested it on the Traveling Salesman (TSP), Delaunay Triangulation, and Convex Hull problem variants. For each problem, the solution to a given instance can be expressed as a single permutation. The authors developed a pointer network model to predict near-optimal solutions by learning from previously solved instances in a supervised manner. The pointer network takes as input the 2D coordinates of each city that must be visited for the 2D Euclidean TSP, and produces

a predicted permutation that represents a tour over all cities. The target label is a permutation that represents the pre-computed minimum-length tour over its corresponding input coordinates. However, this approach can only be applied to problems where the solutions take the form of a single permutation and all permutations are feasible, meaning that the problem does not feature constraints on which tours are allowed in the TSP. Another approach was proposed in [24], The authors developed a supervised learning approach to solve the Quadratic Assignment Problem (QAP) using graph neural networks. The model was trained on individual instances of the problem and their corresponding solutions, and it produced approximate permutation matrices as output. The method was tested on the Traveling Salesman Problem (TSP) and two graph matching problems (which are instances of the more general QAP). A beam search algorithm was used to convert the permutation matrices into feasible tours. Although the model achieved promising accuracy results on small instances, its ability to generalize to larger instances was not demonstrated. Furthermore, the method itself outputs infeasible graphs, having to use beam search to satisfy constraints is another weakness. Following this approach was the use of Graph Convolutional Networks to solve the 2D Euclidean TSP [19]. This approach uses the same techniques in [24] but instead of using graph neural networks the more robust Graph Convolutional Networks are used. This approach produces superior results but also suffers from the same generalization problem.

This work is mainly concerned with supervised learning, but it is important to note that a lot of research has been done in the area of reinforcement learning for attempting to solve CO problems. Optimization problems have a native objective function, which makes it possible in theory to substitute the loss function with the CO objective function. This objective function will serve as a natural reward function in reinforcement learning. Interested readers are referred to section 6.2 of the following survey [18] for an extensive overview of end-to-end RL and for a general overview of all the methods discussed in this work.

2.3.3 End-to-End COL: Predict-and-Optimize

An emerging area at the intersection of machine learning (ML) and computational optimization (CO) involves merging prediction models (ML) and decision models (CO). In this approach, decision models are described by optimization problems that are only partially defined, and their missing parameters are predicted from data. These composite models use constrained optimization as a neural network layer and are trained end-to-end based on their decision-making performance. This approach differs from previous efforts, which focused on solving pre-defined optimization problems more efficiently. Instead, the aim is to combine predictive and prescriptive techniques to create ML systems that learn to make decisions based on real-world data. Take the following constrained optimization problem, in which the objective function f_y and the feasible region C_y depend on a parameter vector y:

$$\mathcal{O}(y) = \underset{z}{argmin}\ f_y(z)\ \ subject\ to\ \ z \in C_y$$

the learning task here is to use supervised learning to predict the unspecified parameter of the CO problem from the empirical data, call this prediction $\hat{y}$, such that the optimal solution $\mathcal{O}(\hat{y})$ best matches the targeted solution $\mathcal{O}(y)$. The empirical data in this setting belongs to some abstractly defined data set $\mathcal{X}$. This predict-and-optimize framework aims to improve on the conventional *two-stage* approach to solving such a problem. The two-stage approach would first involve training an ML model using some sort of conventional loss function such as MSE on the targeted labels y to predict $\hat{y}$ before solving the associated CO problem. This approach has obvious advantages, in which the model learning phase is well justified and independent of any secondary task besides best fitting y to $\hat{y}$. In theory, this approach is infallible if we are able to get a perfect model to make precise predictions. However, when put in practice the two-stage approach is prone to poor performance in the common setting where the true distribution of the data is unknown and cannot be fully represented by the model. Prediction errors of y although low in a good model do not take into account the accuracy of the resulting solution $\mathcal{O}(y)$, resulting in the error propagating into the solution and causing suboptimal models that

are not good in practice for their intended use cases.

The Predict-and-Optimize framework solves this problem by minimizing the decision utility of the prediction $\mathcal{L}(\mathcal{O}(\hat{y}), \mathcal{O}(y))$, with respect to objective decision values $f_{\hat{y}}(\mathcal{O}(\hat{y}))$. This notion of training loss is referred to as regret:

$$regret(\hat{y}, y) = f_{\hat{y}}(\mathcal{O}(\hat{y})) - f_y(\mathcal{O}(y))$$

using this notion of a loss function the training procedure targets optimal solutions $\mathcal{O}(y)$. One important assumption that goes into this notion is that our CO problem has a unique solution, this, of course, implies that $\mathcal{O}(y)$ is directly determined by y and that no other solution will output an optimal solution.
Training these end-to-end architectures using the regret function requires access to the decisions $\mathcal{O}(y)$ and $\mathcal{O}(\hat{y})$, thus this will require the introduction of external CO solvers into the training loop of our ML model. It is very important to note that combinatorial problems are problems with discrete state spaces. This means that the *argmin* of a discrete problem is a piecewise constant function, making it very hard to obtain useful gradients for backpropagation. This problem is a central challenge in a lot of the research in this area and our work directly tries to address this challenge as we will see in the methodology section. To deal with this we need to find ways of forming useful approximations to $\frac{\partial \mathcal{L}}{\partial y}$, this term can be approximated directly but a growing body of work models $\frac{\partial \mathcal{L}}{\partial y}$ by breaking it down into $\frac{\partial \mathcal{O}(y)}{\partial y}$ and $\frac{\partial \mathcal{L}}{\partial \mathcal{O}}$, the difficulty will lie in the first term by having to take derivatives through the *argmin* function.

One of the earliest pieces of work on this topic was the introduction of *differentiable optimization layers* [1], in the paper titled OptNet the authors introduce a network architecture that integrates optimization problems in the form of quadratic programs as individual layers in the larger end-to-end training of a deep neural network. The layers utilize a quadratic program solver that offers exact gradients for backpropagation. This made way to the work that was introduced in [9] which proposed a *predict-and-optimize*

model, which incorporated QPs (quadratic programs) with stochastic constraints into the loop. This would help generate precise solutions for inventory and power generator scheduling problems based on real-world data. Following this work was the introduction of an alternative framework to predict-and-optimize which focused on linear programming problems [36]. The technique allows for exact differentiation of a smoothed approximation of the problem. Although LPs are a special case of QPs the techniques used for gradient calculation in [1] break down. The new proposed technique addresses this issue by forming an approximation of the LP objective function, the new objective function is formed by adding a small quadratic regularization term to turn the objective from this

$$\mathcal{O}(y) = \underset{z}{argmin}\, y^T z \;\; subject\, to \;\; Az \leq b$$

into this

$$\mathcal{O}(y) = \underset{z}{argmin}\, y^T z + \epsilon \|z\| \;\; subject\, to \;\; Az \leq b$$

this technique is essentially what allows us to train our machine learning model on combinatorial decision making problems. This relaxation turns the combinatorial problem into a continuous one, allowing us to analytically differentiate the optimal solution of the continuous problem as a function of the model predictions. The approximation of the desired LP has unique solutions that vary smoothly as a function of their parameters, allowing for the desired accurate backpropagation. This work demonstrated success on problems where the cost vector was predicted from a feature set and was shown to outperform two-stage models.

[12] extended the work of [36] to integrate MILP within the end-to-end training loop. The goal is to tackle more complex NP-Hard combinatorial problems with parameters predicted from data. This is achieved by first reducing the MILP with integer constraints to a linear programming (LP) problem using cutting planes. In an ideal scenario, the LP formulation should produce the same optimal solution as its original mixed-integer form. Exact gradients can then be computed for its smoothed QP. Although the LP approximation to MILP improves with more solving time, practical concerns arise when

the MILP problem cannot be solved to completion. One of the main challenges is that each instance of the NP-Hard problem needs to be solved in every forward pass of the training loop, which can lead to significant runtime obstacles. Additionally, a drawback of this approach is that cutting-plane methods are generally considered less efficient than standard methods such as branch and bound. In light of these drawbacks, superior results were obtained on portfolio optimization and diverse bipartite matching problems when compared to LP-relaxation models as in [36]

Chapter 3

Methodology

we ended our literature review in Chapter 2 by discussing the predict-and-optimize paradigm and discussing some of the key research points in that area but I would like to start this section by further fledgling out the topic by first formally defining the problem we are trying to solve and by then further motivating our work and discussing where it fits in into the current body of research work.

3.1 Problem Description

We consider a general combinatorial optimization problem of the form

$$\min_{x \in \mathcal{X}} f(x, y)$$

where $\mathcal{X}$ is a discrete set enumerating the feasible decisions, and $\mathcal{X} \in \{0, 1\}^n$, that is the decision variable x is a binary vector. The objective f depends on a parameter $y \in Y$. The problem could be easily solved using various methods if the value of y were completely known. However, in this work, we will focus on the scenario where y is not known and needs to be estimated from data. This is a common scenario in bipartite matching, where x denotes whether a pair of nodes are matched and y represents the reward for matching each pair. Frequently, these rewards are obtained from past data. The decision maker observes a feature vector $\theta \in \Theta$ which correlates with y, This creates

a learning challenge that needs to be addressed before any optimization can take place. We handle this issue by modeling y and θ as being drawn from a joint distribution P, following the same approach as in conventional supervised learning. The algorithm will observe training instances $(\theta_1, y_1)...(\theta_n, y_n)$ drawn i.i.d from P. At test time the algorithm is given a feature vector θ that corresponds to some unobserved y, and the algorithm will use θ to predict some parameter value $\hat{y}$. After which, we will solve the optimization problem

$$\min_{x \in \mathcal{X}} f(x, \hat{y})$$

to obtain a decision $\mathcal{O}(\hat{y})$. The utility of this decision is the objective value that $\mathcal{O}(\hat{y})$ obtains with respect to the true but unknown parameter y (i.e, the target), $f(\mathcal{O}(\hat{y}), y)$ let $m : \Theta \to \mathcal{Y}$ denote the model mapping from the observed feature to the unknown parameters. The end goal of the data-decision pipeline is to minimize

$$\mathcal{L}(\mathcal{O}(m(\theta)), y) \tag{3.1}$$

3.2 Motivation

As we have seen with the proposed methods in our literature review, solving this using the two-stage approach is not a good way of approaching this problem, instead, the main research focus for tackling this problem is using the predict-and-optimize paradigm. Almost all the proposed methods in the literature approach this problem by first forming an approximation of the combinatorial optimization problem to allow for useful gradients for backpropagation [36], the end-to-end pipeline is then constructed with an integrated CO solver at the end to attain decisions. The predictive model is trained on the decision quality of its predictions, the forward pass will consist of us first passing a feature vector θ to obtain a predicted parameter $\hat{y}$ which is then passed to the solver to obtain $\mathcal{O}(\hat{y})$. The backward pass entails calculating the regret and then backpropagating back to the model, and depending on the methods, gradients can either be estimated or solved analytically on some approximations of the original problem.

In this work, we aim to address some weaknesses present in the aforementioned methods. One of the major challenges is the training time bottleneck caused by the integrated solver. Additionally, the inference time required by these methods is a more serious issue. Combinatorial problems are typically NP-hard and even if we apply relaxation techniques, solving a convex problem will still require a considerable amount of time, especially as we scale up the problem size. During training, the solver needs to calculate $2 \times Batchsize$ problems for each forward pass to compute the regret. For each sample in the batch, we require the prediction decision $\mathcal{O}(\hat{y})$ and the target decision $\mathcal{O}(y)$. However, it is possible to speed up the training time by computing $\mathcal{O}(y)$ beforehand. Inference time is another significant issue to consider since this type of model may not perform well in time-sensitive settings due to the need to call the solver to obtain the decision for each prediction.

3.3 Proposed solution

To tackle the training and inference time bottleneck caused by the CO solver, we propose a novel approach that replaces the CO solver with a trained blackbox solver, which we refer to as a *proxy*. This method involves training a separate model, the proxy, to learn the mapping between the combinatorial input and output. Our goal in using the proxy instead of the CO solver is to eliminate the need for the solver during both training and inference, reducing the computational burden and significantly enhancing the scalability and efficiency of combinatorial optimization models. This approach involves a *pretraining* phase in which the proxy is trained on a diverse set of problems to improve its generalization capabilities. Additionally, the proxy can be fine-tuned to specific problem instances, further enhancing its performance. The use of a proxy has the potential to improve the performance of combinatorial optimization models while reducing the computational complexity associated with using a CO solver.

Once the pretraining phase is completed, we will integrate the proxy into an end-to-

end pipeline to train our predictive model. The pipeline will begin with a neural network that predicts the missing parameter of our combinatorial optimization problem, denoted as $\hat{y}$. Next, $\hat{y}$ is passed to the proxy, which produces a decision. Training is then carried out by minimizing the following loss function

$$\mathcal{L}(\hat{y}, y) = y^T \mathcal{O}(\hat{y}) - y^T \mathcal{O}(y) \tag{3.2}$$

where y is the ground-truth solution to the combinatorial optimization problem, and $\mathcal{O}(\hat{y})$, $\mathcal{O}(y)$ represent the decisions made by the proxy for the predicted and ground-truth solutions, respectively. This loss function measures the utility of the prediction against the optimal one. In theory, minimizing this function should result in a predictive model that produces optimal decisions, and assuming the existence of unique optimal solutions $\hat{y}$ should converge to y. By using this pipeline, we can train our model in an end-to-end manner to target the quality of the decision.

3.4 Shortest Path Problem

Our proposed solution is a versatile framework that is applicable to both Linear Programming (LP) and Quadratic Programming (QP). The framework builds on previous work presented in [36], and in this study, we evaluate its performance on the well-known shortest path problem. The shortest path problem is an important combinatorial optimization problem that has applications in many fields. The problem is formulated as an LP with integer constraints (A MILP) and is known to be NP-hard. The LP formulation of the shortest path is given as follows:

$$min \sum_{u,v \in A} c_{uv} x_{uv}$$

$$s.t \sum_{v \in V^+(s)} x_{sv} - \sum_{v \in V^-(s)} x_{vs} = -1$$

$$\sum_{v \in V^+(u)} x_{uv} - \sum_{v \in V^-(u)} x_{vu} = 0 \quad \forall\, u \in V \backslash \{s,t\}$$

$$\sum_{v \in V^+(t)} x_{tv} - \sum_{v \in V^-(t)} x_{vt} = 1$$

$$x_{uv} \in \{0,1\}^n \quad \forall\, (u,v) \in A$$

The graph's incidence matrix is denoted by A, and the set of vertices is denoted by V. The cost or weight of the edges is represented by the vector c, the vector x denotes the permutation of the selected edges. The constraints can be seen as $Ax = b$ and are referred to as flow constraints, the coefficients a_{ij} of A are 1 if the arc j has its head a node i, -1 if arc j has its tail at node i, and 0 otherwise. The final constraint is an integrality constraint on x, 1 represents choosing an edge in the path, and 0 otherwise. And the objective function is essentially minimizing the path by deciding which edges to select.

To gain more intuition about this shortest path formulation let us go through a small example of how this problem gets solved. Take a trivial undirected graph with 3 nodes and 3 edges, let the vector $x = \{e1, e2, e3\}$ denote the respective edge selection permutation, and let the vector $c = \{1, 2, 2\}$ denote the respective weight of each edge. the problem is to essentially find a binary vector x called the decision, such that the objective function is minimized while satisfying the problem constraints. The graph of the problem will look like the following:

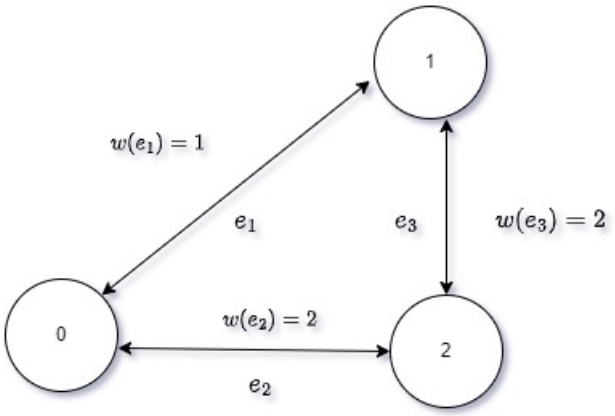


Figure 3.1: trivial graph example

Let's assume we want to find the shortest path from node 0 to node 2. Then the optimal decision, in this case, would be $x = \{0, 1, 0\}$, with an optimal path cost of 2. We will use the following notation $c^T x$ for the objective function, expanding it out would give us

$$1 \times (0) + 2 \times (1) + 2 \times (0) = 2$$

3.5 Pretraining Phase

The proposed solution has a lot of working parts, the first and arguably the most important part of this pipeline is constructing the proxy. Previous methods utilized CO solvers to obtain decisions. These CO solvers are very robust and produce optimal or near-optimal solutions to the problem at hand, Therefore, to ensure the success of our proposed solution, it's essential to develop a sturdy proxy that can effectively replace the solver. The learning task is to learn a mapping $\mathcal{M} : \mathcal{Y} \to \mathcal{X}$ from a vector that represents the edge weights to a vector that represents the edge decision of the shortest path. We will need to construct a dataset $\mathcal{D} = \{y_i, x_i\}_{i=1}^{N}$ where the feature vector $y_i \in \mathcal{Y}$ is the edge weights and $x_i \in \mathcal{X}$ is the decision. Constructing this dataset will require careful consideration before proceeding as it is a non-trivial task. There are various ways of obtaining the dataset, and the shortest path can essentially be solved using a Mixed Integer Linear Programming (MILP) approach through techniques like branch and bound or constraint programming algorithms. Alternatively, one could use graph representations

of the problem and solve it using Dijkstra's or Bellman-Ford algorithms. However all these approaches encounter the same challenge, the MILP formulation of shortest path is a piecewise constant function, this means that the gradients will not exist at some points due to discontinuity, or if they do exist they are zero. In theory, if we are able to learn a good mapping then the proxy will learn a piecewise constant function which will result in useless gradients when backpropagating. Of course, in practice we do not expect to be able to estimate the function exactly, which might result in a function that allows for useful gradients but such a function may not be well-defined and could lead to unexpected behavior.

One way to address this issue is to apply the techniques outlined in the literature review. Specifically, we can approximate the objective function by introducing a quadratic term to it, as suggested in [36]. This quadratic term will result in an approximation of the desired LP and allows for precise backpropagation. The resulting objective function will look as follows

$$\mathcal{O}(y) = \underset{x}{argmin}\, y^T x + \epsilon \|x\|$$

We have now defined our new objective function and we will need to construct our dataset. To do so we will need an LP solver to obtain the decisions, the solver used in this work is CVXPY [8]. CVXPY is a Python-embedded modeling language designed for convex optimization problems. It has an inbuilt feature called cvxpy layers, which allows for easy integration into a neural network.

CVXPY is primarily designed for convex optimization, our approximated shortest path problem is a mixed-integer program, which poses a challenge for the direct use of CVXPY. In order to utilize CVXPY, we will need to take an additional preliminary step to convert our problem from a mixed-integer linear program into a linear program. To obtain a convex problem, we will relax the integer constraints of our problem. By relaxing the integer constraints on x and turning the constraint to $0 \leq x \leq 1$, we create a convex set

on the constraints. This will transform our optimization problem from its original form

$$\begin{aligned} \mathcal{O}(y) = \underset{x}{argmin}\ y^T x + \epsilon\|x\| \\ s.t\ \ Ax = b \\ x \in \{0,1\}^n \end{aligned} \tag{3.3}$$

into this following relaxed convex problem

$$\begin{aligned} \mathcal{O}(y) = \underset{x}{argmin}\ y^T x + \epsilon\|x\| \\ s.t\ \ Ax = b \\ x \in [0,1]^n \end{aligned} \tag{3.4}$$

By utilizing equation 3.4, we can acquire the appropriate dataset to train the proxy. It is worth mentioning that several distinct quadratic terms are available to obtain slightly varied smoothed objectives, but the basic concept remains the same.

3.6 Defining The Shortest Path Problem for The Decision-Focused Learning Task

We have established the shortest path linear program and the associated learning task. The next step is to adjust the shortest path problem to align with the learning task. To accomplish this, we will need to partially define the shortest path problem, to model it according to the learning task as in equation 3.1. The absent parameter in our problem statement are the edge weights of the graph.

The first step will be to define the dataset for the predictive model. Let the dataset $\mathcal{D} = \{\theta_i, y_i\}$ where θ is an abstract feature vector and y is the edge weight vector. The feature vector θ can be formulated in a number of ways, as mentioned a common way of formulating θ is by using the historical data, this definition essentially turns $\mathcal{D}$ is a time series dataset.

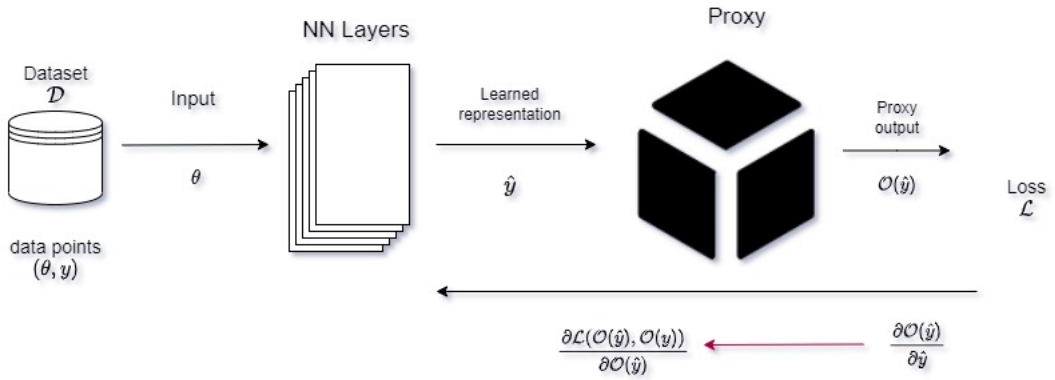


Figure 3.2: E2E framework architecture

The goal of the predictive model will be to learn to produce accurate edge weight predictions from the feature set, such that predictions produce optimal decisions.

3.6.1 Utility Measurement of The Predictive Model

We have discussed why measuring the utility of predictions is a superior method for models that demand a decision based on their predictions. This notion of measurement is known as regret 3.2, to better understand this function and its application in the architecture, let's examine an example of a forward pass. Assume we have the same graph as in figure 3.1. Then the target vector $y = \{1, 2, 2\}$, and let the predicted vector $\hat{y} = \{1, 2.5, 1.3\}$. Looking at this prediction we can see that y and $\hat{y}$ are not that far off and using a standard measurement such as MSE will give us $L_2(y, \hat{y}) = 0.74$

Let us now compute the regret, to do that we will need the decision on each vector. The target vector is $\mathcal{O}(y) = \{0, 1, 0\}$ and the predicted vector is $\mathcal{O}(\hat{y}) = \{1, 0, 1\}$.

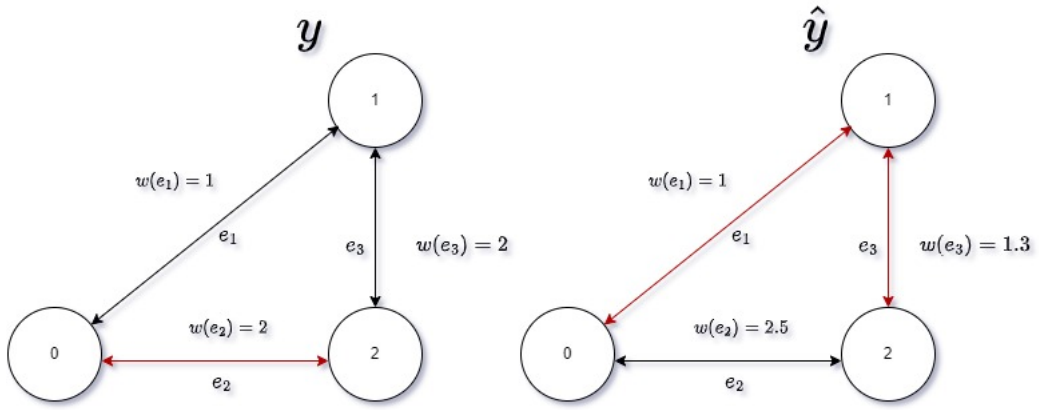


Figure 3.3: graph with true edge weights y (left) graph with predicted edge weights $\hat{y}$ (right)

Substituting the value back into the regret function 3.2 will give us

$$\mathcal{L}(\hat{y}, y) = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 1$$

Regret is not concerned with the prediction itself, it measures suboptimality by measuring the utility of the prediction against the optimal utility.

3.7 Benchmarks

To evaluate our proposed approach, we must first establish a few baselines. Our initial benchmark will involve comparing our method directly with the approach described in [36], which our work seeks to build upon by replacing the CO solver with a proxy. We of course do not expect to beat the solver pipeline with respect to accuracy as the solver is always optimal, and expecting to train a perfect proxy is not realistic, nonetheless we would like to get close while significantly improving the training and inference times of the pipeline.

Another benchmark that we aim to establish is against the two-stage approach, which involves training a predictive model using a standard loss function, such as MSE, and assessing its performance based on the utility of its predictions. These two benchmarks

will establish a sort of bound on the performance of our model. The two stage approach will serve as a lower bound as we do not anticipate it to achieve high decision accuracy, while the solver pipeline will act as an upper bound.

Expected performance in terms of utility of the predictions

$$Two - Stage \leq Proxy\ Pipeline \leq Solver\ pipeline$$

Chapter 4

Results

The training data for the proxy and the pipeline is generated randomly using a uniform distribution, Specifically, we generate a random incidence matrix with randomly assigned edge weights and use 0.75 and 0.25 training test split, respectively.

4.1 Proxy Results

The robustness of the proxy is measured using the following criteria

1. **L2 error**

2. **L1 error**

3. **Average Violations** violations in this sense refer to constraint satisfaction. our solution will need to satisfy two constraints

   - flow violations $Ax = b$

   - decision violation $1 \geq x \geq 0$

The training was done on a number of models, we utilize two types of training functions $L1$ and $L2$, as they produced the most favorable results. The reported results are obtained after training and are obtained via the test dataset. Specifically, we report the $L1$ and $L2$ errors along with the average constraint violations. The experiments were conducted on various problem sizes denoted by A. In all the problems, the smoothing function used

was $\epsilon||x||_2$. The architecture column refers to the respective loss function used during training along with the two different training techniques. The first approach involves incorporating a penalty term for constraint violations in the loss function, denoted by $+Pen$, while the other approach utilizes the Augmented Lagrangian method by relaxing flow constraints into the objective, and is referred to as $AugmentedLag$.

A	Architecture	$L2$	$L1$	$Violations$
5×8	$L2$	0.0045	0.0457	0.3089
	$L2_{+Pen}$	0.0048	0.0474	**0.2979**
	$L2_{AugmentedLag}$	0.0049	0.0545	1.7280
	$L1$	**0.0022**	**0.0278**	0.3077
20×30	$L2$	0.0043	0.0301	1.0603
	$L2_{+Pen}$	0.0044	0.0307	1.0557
	$L2_{AugmentedLag}$	0.0182	0.0924	3.2992
	$L1$	**0.0022**	**0.0184**	**0.9367**
80×130	$L2$	0.0001	0.0021	1.5598
	$L2_{+Pen}$	0.00009	0.0020	1.5594
	$L2_{AugmentedLag}$	0.0041	0.0475	8.3101
	$L1$	**0.00007**	**0.0015**	**1.4091**

Table 4.1: Proxy results

4.2 Pipeline Results

The pipeline will require a feature set for edge weight predictions. As discussed in Chapter 3 the typical approach is to use historical data, however, in this work, we generate the feature set as follows. Let y represent the target data (i.e the true edge weights), and let A be a random matrix where each element a_{ij} is a value between 0 and 1, obtained from a uniform distribution. The vector y is multiplied by the inverse of A to obtain the feature vector θ

$$\theta = A^{-1}y$$

thus the predictive models learning task is to learn a function $f(\theta) = y$. The robustness of the pipelines are measured using the below criteria

1. **L2 weight error** The $L2$ weight error on the weight prediction of the predictive model

2. **L2 decision error** The $L2$ error of the decision vector obtained from the predictive model (this is measured using the proxy in the proxy pipeline and the solver in the solver pipeline and 2stage)

3. **Average Violations**

4. **Inference time** Time required to solve all problem instances in the test dataset (Seconds)

We will integrate the best proxies into the decision-focused pipeline represented by $proxy-decision$ in the architecture columns, and benchmark it against the solver pipeline ($Solver - Decision$) and two-stage approach (2stage).

A	Architecture	$L2_{weights}$	$L2_{decision}$	$Violations$	$Inf\ time$
5×8	$Solver - Decision$	0.031	0.190	0.000	0.331
	$L2proxy_{+Pen} - Decision$	0.079	0.061	1.275	**0.003**
	$L1proxy - Decision$	0.037	0.107	0.772	**0.003**
	$2Stage$	**0.009**	**0.106**	0.000	0.331
20×30	$Solver - Decision$	0.060	**0.040**	0.000	0.415
	$L2proxy_{+Pen} - Decision$	0.009	0.010	1.663	**0.003**
	$L1proxy - Decision$	0.009	0.005	1.340	**0.003**
	$2Stage$	**0.007**	0.052	0.000	0.390
80×130	$Solver - Decision$	0.014	**0.002**	0.000	0.531
	$L2proxy_{+Pen} - Decision$	0.011	0.003	1.915	**0.006**
	$L1proxy - Decision$	0.016	0.001	1.728	**0.006**
	$2Stage$	**0.003**	0.003	0.000	0.475

Table 4.2: Pipeline results

Chapter 5

Discussion

It is crucial for the proposed method to utilize a robust proxy instead of the solver, as the accuracy of the decisions depends on it. The various attempted proxy training methods are listed in Table 4.1. Convergence was achieved for all measured properties, with training graphs provided in the appendix. Both L1 and L2 errors were comparatively low, but employing L1 as the loss function yielded slightly superior results. Despite our efforts, we encountered difficulty in satisfying constraints; all attempted methods failed to accomplish this effectively with the Augmented Lagrangian performing the worst. Satisfying equality constraints is usually challenging and our case is no different. The constraints grow with problem size and scaling up made them harder to satisfy. We experimented with different network architectures and hyperparameters but that did not yield any substantial improvement with constraint satisfaction, Consequently, we turned to some methods from the literature in an attempt to improve our results the first method we tried was *Reparametrization*. Satisfying constraints would be easy if we had a method to round the predictions from the proxy, rounding to exact integers would be difficult but if we could round the prediction to near integer values then we could better satisfy the constraints. Since rounding functions are usually discontinuous, we had to devise some approximated differentiable rounding functions. However, each proxy was unique, and at every stage of training, the predictions were slightly different. As a result, having a one-size-fits-all rounding function was not feasible, and integrating such a function into

the training loop resulted in weird behavior. In general, the proxy performed worse when using these rounding techniques than when not using them at all. Another approach we attempted was to use *Projected gradient descent*, which involved projecting the solutions back into the feasible region. However, this method did not produce satisfactory results either. The training was very noisy, and it never converged.

In general such a proxy will not perform well due to its loose satisfaction of constraint as shown in 4.2, the proxy pipeline converges with respect to weight prediction but performs very poorly with respect to constraint satisfaction making it unreliable. Although our method was able to overcome the runtime issue and achieves a speedup of 110x, the pipeline itself does not effectively solve the problem. The solutions provided by the pipeline are infeasible and far from optimal when compared with the benchmarks, rendering them unsuitable for use.

Chapter 6

Conclusions and Future Work

Our method has shown great potential in terms of runtime, as it provided a massive speedup when compared to popular E2E approaches and the typical two-stage approach. Unfortunately, the quality of the pipeline does not hold up to the benchmarks with respect to solution quality. Constraint satisfaction is crucial in CO problems, and the pipeline's inability to satisfy these constraints is a fundamental issue with the proposed technique. The main cause of the poor solution quality is the proxy; substituting the solver with a proxy assumes that we have a robust proxy that can obtain decision vectors of high quality. To make this approach work, a more sophisticated training procedure for the proxy is required such that it can obtain solutions with minimal constraint violations.

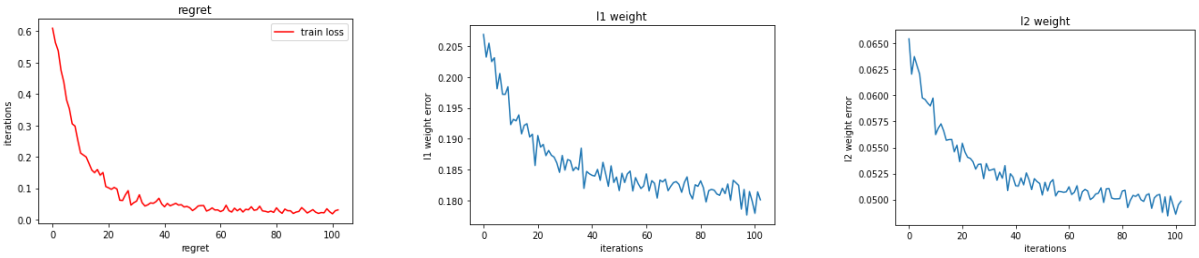
One possible future approach is to train a proxy using the Lagrangian Dual Framework (LDF) [13]. This method has demonstrated success across multiple domains in predicting CO solutions while successfully satisfying constraints. One could also leverage superior network architectures for such problems and implement a pointer network, as done in [33], or try to apply the supervised learning approach using GNNS [24] to our problem. There is plenty of room for improvement, and calling on more sophisticated techniques will likely help us produce a better working proxy. However, no matter how robust the proxy is, we cannot expect it to perform as well as a solver. This is a problem because, no matter how small the error is, the predictive model will eventually try to exploit it to minimize the regret. This results in a pipeline that produces infeasible solutions for

problems where feasibility and constraint satisfaction are of utmost importance. This plug-and-play method for the proxy will likely not work on its own, we will either have to augment the regret function to force the predictive model to avoid exploiting the small errors induced by the proxy or move away from the proxy and explore techniques that may involve integrating a hybrid model into the pipeline such as [20] to get the best of both worlds.

Appendix A

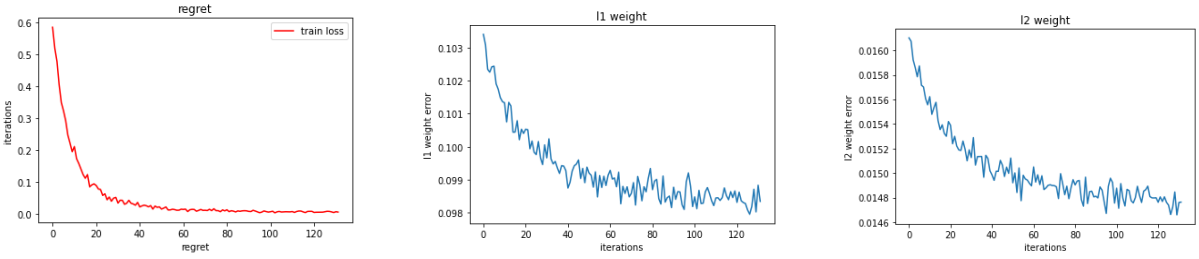
A.1 Experimental Data

A.1.1 Solver Decision-Focused Pipeline



(a) regret (b) L1 weight error (c) L2 weight error

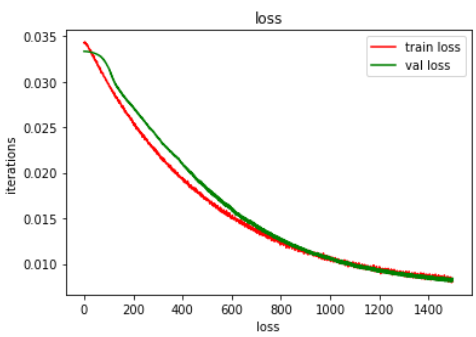
Figure A.1: Decision focused learning solver pipeline graph size 20×30, 7 layer architecture, SGD optimizer, Lr 0.00001



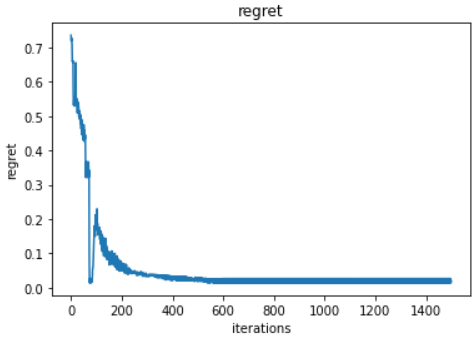
(a) regret (b) L1 weight error (c) L2 weight error

Figure A.2: Decision focused learning solver pipeline graph size 80×130, 7 layer architecture, SGD optimizer, Lr 0.00001

A.1.2 Two-Stage

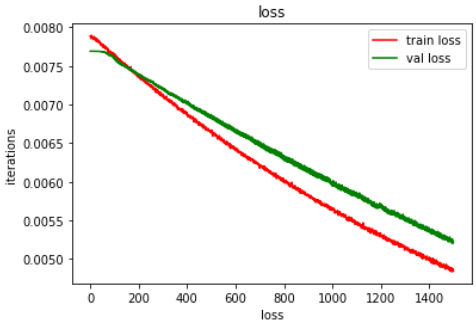


(a) loss

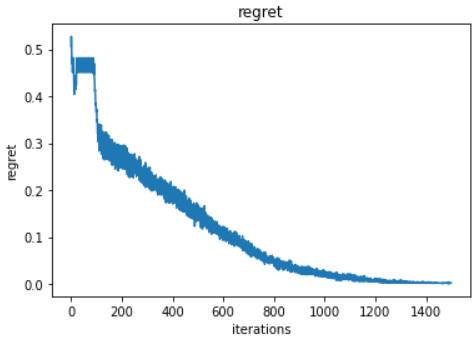


(b) regret

Figure A.3: graph size 20×30, 7 layer architecture, SGD optimizer, Lr 0.001



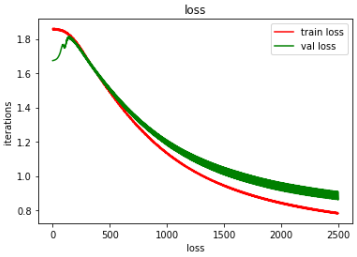
(a) loss



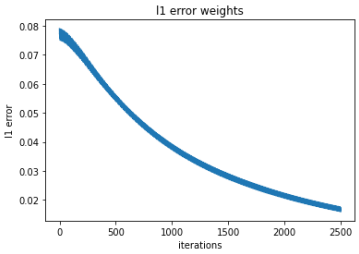
(b) regret

Figure A.4: 2stage 80×130, 7 layer architecture, SGD optimizer, Lr 0.001

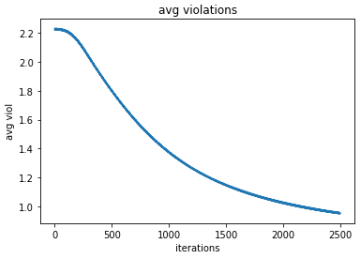
A.1.3 Proxy PreTraining



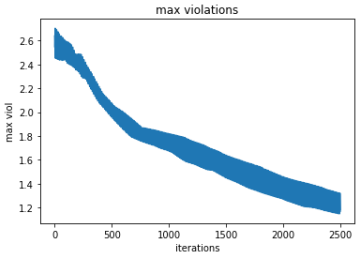
(a) Loss



(b) L1 weight error

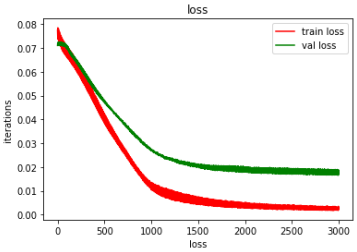


(c) average violations per batch

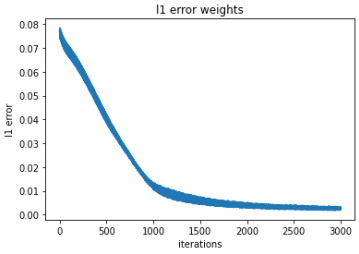


(d) max violations per batch

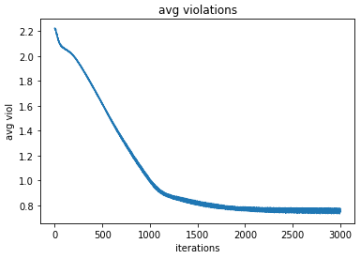
Figure A.5: Proxy Training using MSE + violation Penalty, graph size 20×30, 7 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.0001



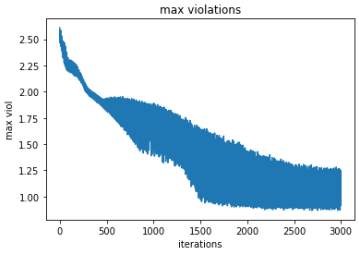
(a) Loss



(b) L1 weight error

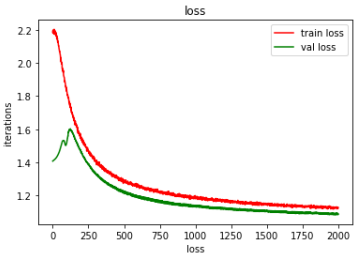


(c) average violations per batch

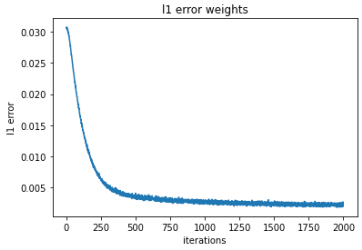


(d) max violations per batch

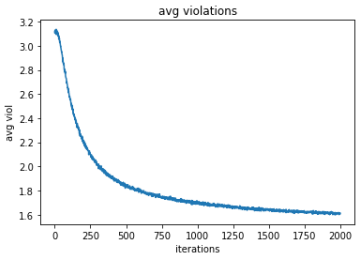
Figure A.6: Proxy Training using L1 loss, graph size 20×30, 7 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.0001



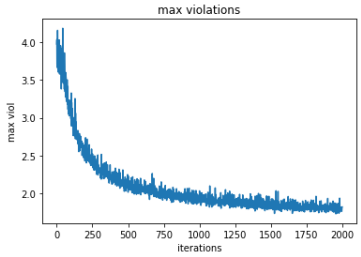
(a) Loss



(b) L1 weight error

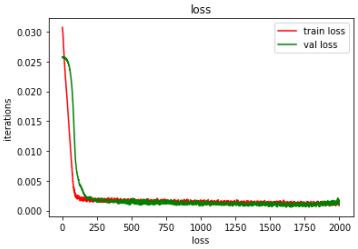


(c) average violations per batch

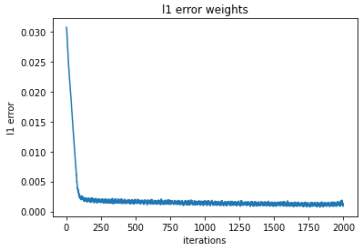


(d) max violations per batch

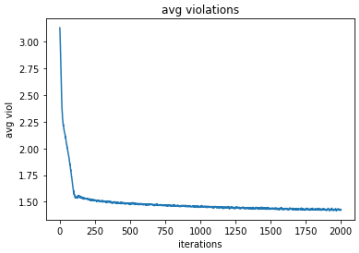
Figure A.7: Proxy Training using MSE + penalty term, graph size 80 × 130, 8 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.001



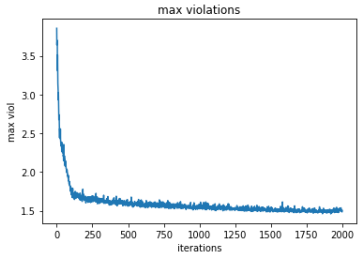
(a) Loss



(b) L1 weight error



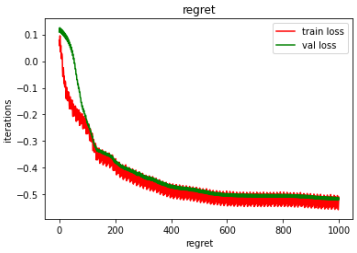
(c) average violations per batch



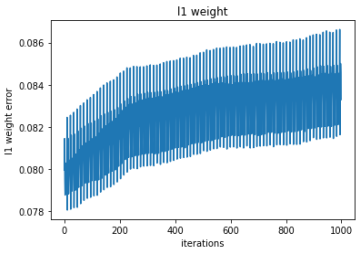
(d) max violations per batch

Figure A.8: Proxy Training using L1 loss, graph size 80 × 130, 8 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.001

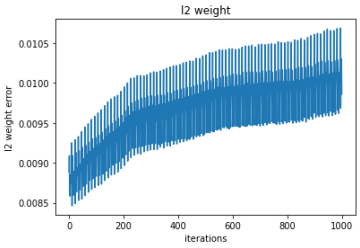
A.1.4 Proxy Decision-Focused Pipeline



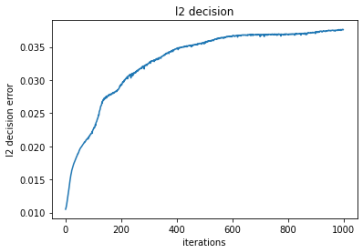
(a) regret



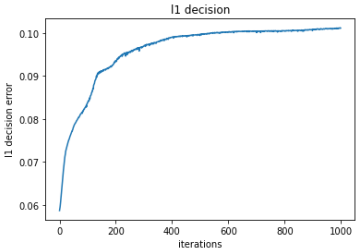
(b) L1 weight error



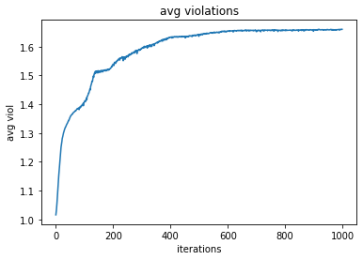
(c) l2 weight error



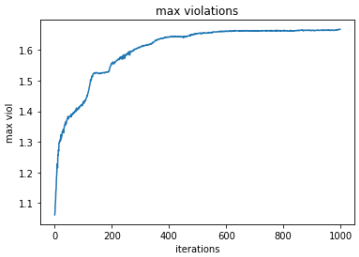
(d) l2 decision



(e) l1 decision

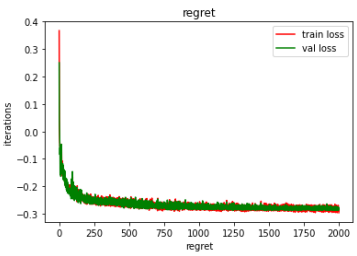


(f) avg viol

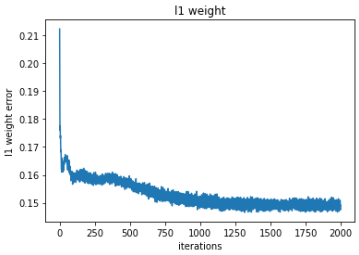


(g) max viol

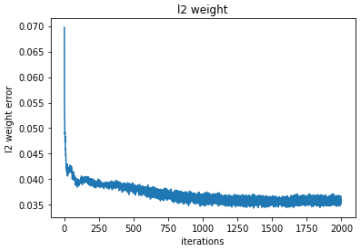
Figure A.9: L2+pen Proxy pipeline, graph size 20×30, 7 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.001



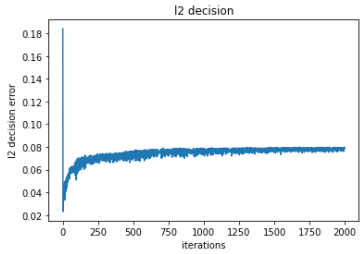
(a) regret



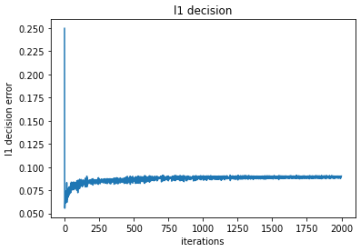
(b) L1 weight error



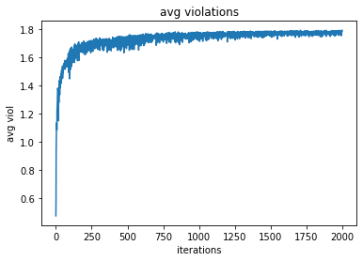
(c) l2 weight error



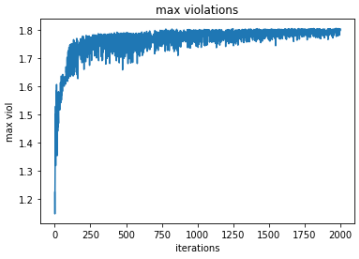
(d) l2 decision



(e) l1 decision

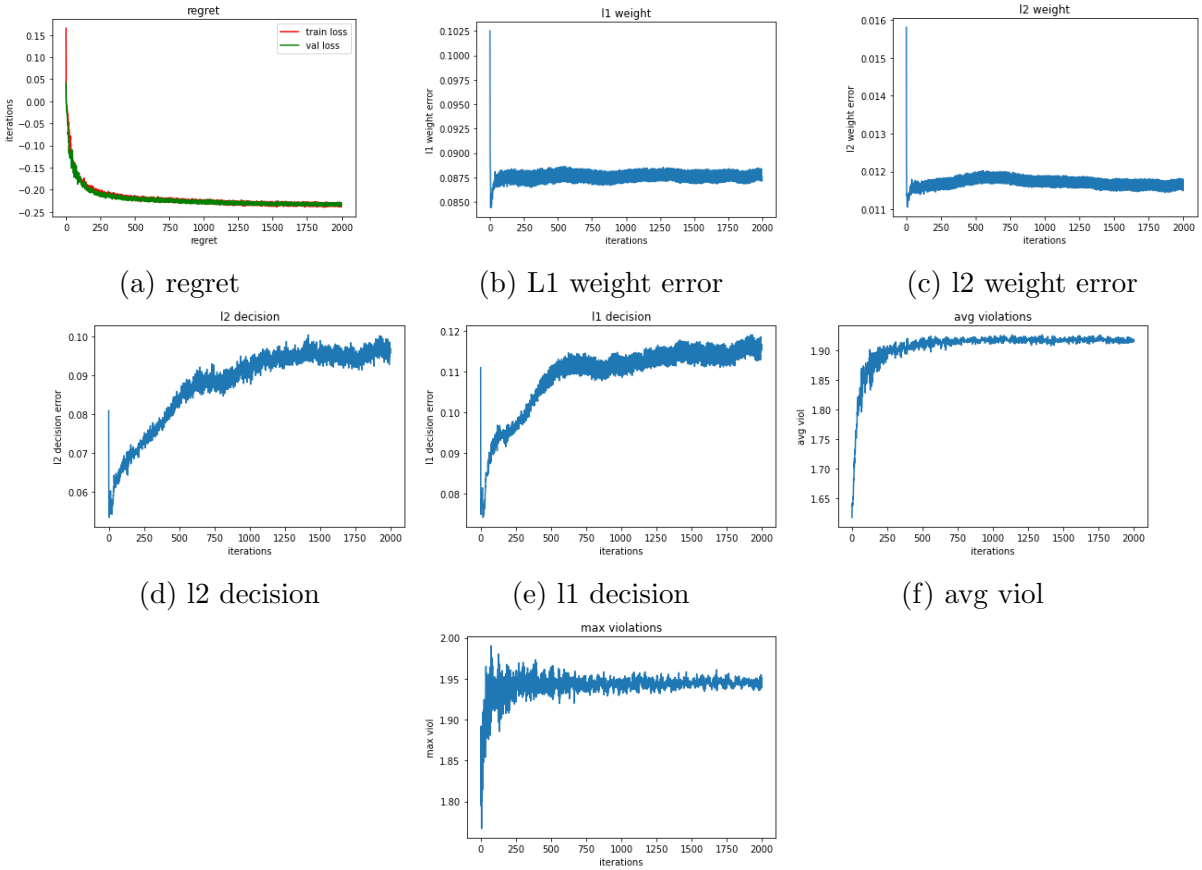


(f) avg viol



(g) max viol

Figure A.10: L1 Proxy pipeline, graph size 20×30, 7 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.001 (*generated after, not exactly the same as the pipeline results section*)

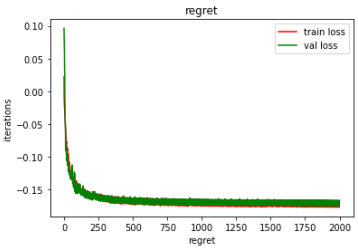


(a) regret (b) L1 weight error (c) l2 weight error

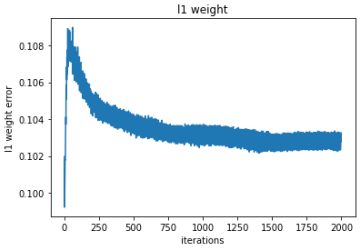
(d) l2 decision (e) l1 decision (f) avg viol

(g) max viol

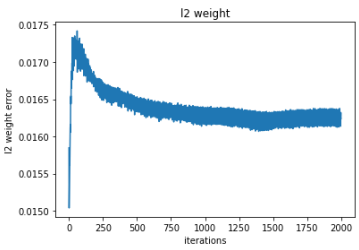
Figure A.11: L2+pen Proxy pipeline, graph size 80×130, 9 layer architecture + batch-norm + dropout, SGD optimizer, Lr 0.001



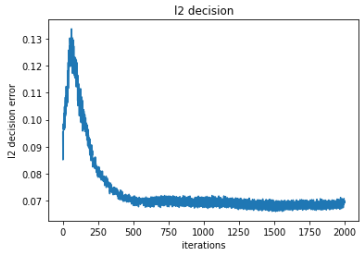
(a) regret



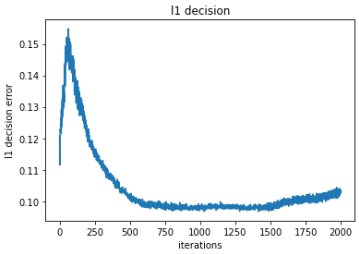
(b) L1 weight error



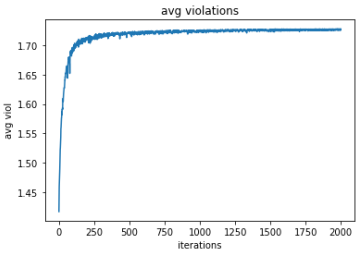
(c) l2 weight error



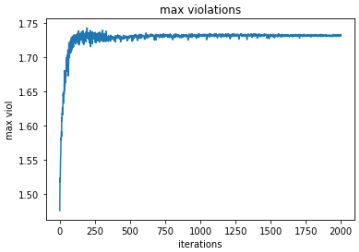
(d) l2 decision



(e) l1 decision



(f) avg viol



(g) max viol

Figure A.12: L1 Proxy pipeline, graph size 80 × 130, 9 layer architecture + batchnorm + dropout, SGD optimizer, Lr 0.001

Appendix B

B.1 Logarithmic Smoothing Term

We attempted to use a different smoothing term. The results of different smoothed functions can be found in the below tables, as in the results section we train proxies (using MSE) and integrate the best ones into the pipeline. S1 refers to first smoothing term $\epsilon||x||_2$ and S2 refers to $\epsilon x log(x)$. Unfortunately, we found this smoothing to be insensitive to any substantial improvements. Furthermore, adding this term causes a singularity in the objective function at its optimal values (i.e, 0). To make this work the constraints would have to be changed to $[\epsilon + 0, 1 - \epsilon]$. Pursuing further tests with this smoothing term seemed to be a fruitless endeavor. It did show a slight improvement in some aspects but nothing to suggest that we should move forward with it.

A	Architecture	$L2$	$L1$	$Violations$
5×8	$S1$	0.0045	0.0457	0.3089
	$S1_{+Pen}$	0.0048	0.0474	0.2979
	$S1_{AugmentedLag}$	0.0049	0.0545	1.7280
	$S2$	**0.0038**	**0.0476**	**0.2596**
	$S2_{+Pen}$	0.0039	0.0482	0.2632
	$S2_{AugmentedLag}$	0.0049	0.0548	1.7183

Table B.1: Proxy results appendix

A	Architecture	$L2_{weights}$	$L2_{decision}$	$Violations$	$Inf\ time$
5×8	$S1solver - Decision$	0.031	0.190	0.000	0.331
	$S1proxy_{+Pen} - Decision$	0.079	0.061	1.275	**0.003**
	$S2solver - Decision$	0.032	0.187	0.000	0.600
	$S2proxy_{pen} - Decision$	0.052	0.026	1.095	**0.006**
	$2Stage$	**0.009**	**0.106**	0.000	0.331

Table B.2: Pipeline results appendix

Bibliography

[1] Brandon Amos and J Zico Kolter. "Optnet: Differentiable optimization as a layer in neural networks". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 136–145.

[2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. "Machine learning for combinatorial optimization: a methodological tour d'horizon". In: *European Journal of Operational Research* (2020).

[3] Brilliant.org. *Linear Programming*. N/A. URL: `https://brilliant.org/wiki/linear-programming/`.

[4] Laura I Burke. "Neural methods for the Traveling Salesman Problem: insights from operations research". In: *Neural Networks* 7.4 (1994), pp. 681–690.

[5] Boris Burkov. *Introduction to Linear Programming*. 2021. URL: `http://borisburkov.net/2021-12-10-1/`.

[6] George B Dantzig. *Maximization of a linear function of variables subject to linear inequalities*. 1951.

[7] Fabrizio Detassis, Michele Lombardi, and Michela Milano. "Teaching the old dog new tricks: Supervised learning with constraints". In: *arXiv preprint arXiv:2002.10766* (2020).

[8] Steven Diamond et al. *CVXPY: A Python-Embedded Modeling Language for Convex Optimization*. `https://www.cvxpy.org`. 2016.

[9] Priya L Donti, Brandon Amos, and J Zico Kolter. "Task-based end-to-end model learning in stochastic optimization". In: *arXiv preprint arXiv:1703.04529* (2019).

[10] Khalil B Elias et al. “Learning to Run Heuristics in Tree Search”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI17*. 2017, pp. 659–666.

[11] Favio Favata and Richard Walker. “A study of the application of Kohonen-type neural networks to the travelling salesman problem”. In: *Biological Cybernetics* 64.6 (1991), pp. 463–468.

[12] Aaron Ferber et al. “Mipaal: Mixed integer program as a layer”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2020.

[13] Ferdinando Fioretto et al. “Lagrangian Duality for Constrained Deep Learning”. In: *European Conference on Machine Learning and Knowledge Discovery in Databases*. Vol. 12461. Lecture Notes in Computer Science. Springer. 2020, pp. 118–135. DOI: 10.1007/978-3-030-67627-2_8.

[14] Maxime Gasse et al. “Exact Combinatorial Optimization with Graph Convolutional Neural Networks”. In: *arXiv preprint arXiv:1906.01629* (2019).

[15] Geoffrey J. Gordon. *10725 Advanced Topics in Deep Learning: Lecture 3*. 2012. URL: https://www.cs.cmu.edu/~ggordon/10725-F12/scribes/10725_Lecture3.pdf.

[16] Prateek Gupta et al. “Hybrid models for learning to branch”. In: 2020.

[17] John Hopfield and D Tank. “Neural computation of decisions in optimization problems”. In: *Biological cybernetics* 52 (1985), pp. 141–152.

[18] Pascal Van Hentenryck James Kotary Ferdinando Fioretto and Bryan Wilder. “End-to-End Constrained Optimization Learning: A Survey”. In: *survey* (2022).

[19] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. “An efficient graph convolutional network technique for the travelling salesman problem”. In: *arXiv preprint arXiv:1906.01227* (2019).

[20] Elias Khalil et al. “Learning to Branch in Mixed Integer Programming”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 2016.

[21] Nina Mazyavkina et al. “Reinforcement learning for combinatorial optimization: A survey”. In: *Journal of [Journal Name]* (2021).

[22] Eugene Ndiaye et al. “Gap safe screening rules for sparsity enforcing penalties”. In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 4671–4703.

[23] Yeesian Ng et al. “Statistical learning for dc optimal power flow”. In: *2018 Power Systems Computation Conference (PSCC)*. IEEE. 2018, pp. 1–7.

[24] Alex Nowak et al. *Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks*. 2018. arXiv: `1811.08451 [cs.LG]`.

[25] *Optimization Problem Types*. `https://neos-guide.org/guide/types/`.

[26] Michael JD Powell. *Optimization*. 1969. Chap. A method for nonlinear constraints in minimization problems, pp. 283–298.

[27] Elea Prat and Spyros Chatzivasileiadis. “Learning active constraints to efficiently solve bilevel problems”. In: *arXiv preprint arXiv:2010.06344* (2020).

[28] Jialin Song et al. “A general large neighborhood search framework for solving integer linear programs”. In: *Advances in Neural Information Processing Systems*. 2020, pp. 20012–20023.

[29] Lieven Vandenberghe Stephen Boyd. *Convex Optimization*. Cambridge university press, 2004.

[30] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. “Reinforcement Learning for Integer Programming: Learning to Cut”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 9367–9376.

[31] Andrew I Vakhutinsky and Bruce L Golden. “A hierarchical strategy for solving traveling salesman problems using elastic nets”. In: *Journal of Heuristics* 1.1 (1995), pp. 67–76.

[32] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in Neural Information Processing Systems*. Google Brain, Google Research, University of Toronto. 2017, pp. 5998–6008.

[33] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". In: *arXiv preprint arXiv:1506.03134* (2017).

[34] Wikipedia. *Branch and bound*. https://en.wikipedia.org/wiki/Branch_and_bound. 2021.

[35] Wikipedia. *Lagrangian Relaxation*. 2021. URL: https://en.wikipedia.org/wiki/Lagrangian_relaxation.

[36] Bryan Wilder, Bistra Dilkina, and Milind Tambe. "Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 1658–1665.

[37] Ming Zhao. *Mixed-Integer Programming*. N/A. URL: https://ming-zhao.github.io/Business-Analytics/html/docs/optimization/mixed-integer_programming.html.

YEHYA FARHAT

yehyafarhat@outlook.com | https://www.linkedin.com/in/yehya-farhat/

EDUCATION

Syracuse University, College of Engineering & Computer Science, Syracuse, NY — August 2021 – May 2023
Master of Science in Computer Science

American University of Beirut, Faculty of Arts and Science, Beirut, Lebanon — August 2016 – December 2020
Bachelor of Science in Computer Science and a Minor in Mathematics

PROFESSIONAL/ACADEMIC EXPERIENCE

Teaching and Research Assistant, Syracuse University College of Engineering & Computer Science, Syracuse — January 2021 - Present

- Served as a TA for Advanced Computer Architecture, Spring 2022; Structured Programming and Formal Methods, Fall 2022/2023; Computer Organization & Programming Systems, Spring 2023
- Conducted in-depth research on combinatorial optimization problems, leveraging optimization modelling strategies and state-of-the-art machine learning techniques. Employed advanced neural network architectures to effectively model, solve, and develop state-of-the-art solutions for these complex problems

IT Consultant, Syracuse University College of Law – Syracuse, NY — October 2021 - Present

- Imaged and connected new laptops and desktop devices to university servers, maintained and managed old devices. Setting up new kiosk machines using the university's updated imaging policies, successfully navigating server migration and replacing old kiosk desktops
- Identified and resolved connectivity issues with all RoomWizard devices in the Law School by setting up and configuring a new proxy server, ensuring smooth and uninterrupted functionality

Data Scientist, Qnovo – Milpitas, CA — June 2022 – August 2022

- Worked as a full-stack engineer to design and build a battery testing platform and a corresponding user interface to query the database, reduced time spent on ad-hoc reporting from 2-3 days to 10 minutes. software is use by all battery engineers to manage tests and reporting
- Cleaned and performed exploratory data analysis on 5M + rows of time-series data. Our analysis was used to confirm the existence of a sensitivity problem in battery capacity estimates

Software and Automation Engineer, Precast FZCO – Dubai, UAE — April 2021 - July 2021

- Devised and implemented cloud-based Microarchitecture utilizing Microsoft azure to create a correspondence between our CMMS software and the clients IOT software, streamlined task creation between software systems, completely eliminating the need for manual labor and reducing task creation time to zero
- Designed and implemented a universal integration module to connect multiple types of ERP and IOT software to our CMMS software, the module included an interface to allow clients to specify which information to streamline between their software

PROJECTS

Optimal Stopping Problem with unknown variable — Fall 2021

- Analyzed one form of the optimal stopping problem known as the secretary problem which has a closed form statistical solution. The equation requires the number of candidates to be known beforehand. Collaborated with a team member to attempt to find an efficient solution to the same problem but without knowing the number of candidates.
- Utilized python to implement tree algorithms and unsupervised machine learning models for time series data representing unknown candidates, pitched up our AIs against the benchmark case and visualized results.
    - Project link: https://github.com/keith-leung/cis667-secretary-problem

University Petition system — Fall 2020

- Cooperated with 3 other team members to build a petition website for the American University of Beirut. Students submit their petition and transcripts; the transcripts are automatically parsed, and the chairperson is informed if students meet the requirements for their submitted petition. The project would allow the chairperson to save countless hours of organizing and reading through transcript to conclude if requirements are met; Technology stack: JavaScript, Node.js, React, MongoDB
    - Project link: https://github.com/yehyafarhat1/rpms

Power Monitor — Spring 2020

- Cofounded & developed power monitor, pitched to Lebanon's public government energy sector to raise money for full scale deployment. The project would allow the energy sector to transition from physical paper bills into digital bills, it would also cut down on the required transportation and employees for these deliveries, saving the sector thousands of dollars each month. Users could also cut down on energy usage by tracking their carbon footprint
- Programmed a raspberry pi to captures images of an electrical odometer, convert captured images into digits and send to database. Phone application then uses each user's data to calculate user's daily usage, monthly usage, electricity bill, and carbon footprint; Technology stack: Raspberry Pi, Python, Dart, Flutter, Google Firebase
    - Product website: https://powermonitoraub.wordpress.com/

Image-Manipulation tool (IMP) — Fall 2019

- Collaborated with a team to create an application where users can add any image for manipulation. changing image color, sharpening, cropping, zooming, and applying other Pixel manipulation algorithms
    - Project link: https://github.com/yehyafarhat1/IMP

TECHNICAL SKILLS

- C++; C; Python; SQL; C#,.NET; R; CUDA; JavaScript; Dart; Julia
- Visual studio; Microsoft SQL; XCode; Android Studio; Heroku; MongoDB; Google Firebase; Microsoft Azure; Selenium; REST API, GraphQL; Pytorch; Numpy; Keras; Flask; React; Flutter