Rheinisch-Westfälische Technische Hochschule Aachen
Diese Arbeit wurde am Lehrstuhl für Informatik 13 (Computer Vision) und am Lehrstuhl für Operations Research vorgelegt
Bachelor Thesis

Learning to Solve Combinatorial Optimisation Problems Effectively

Marawan Emara

Matriculation Number 391 003

11th April 2023

First Examiner: Univ.-Prof. Dr. Bastian Leibe
Second Examiner: Univ.-Prof. Dr. Marco Lübbecke

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

In this thesis, we take a look at tackling the challenge of solving complex combinatorial optimisation problems using a combination of machine learning and combinatorial optimisation techniques. As modern-day problems become increasingly intricate, it is essential to develop efficient and effective solutions to address them. We focus on the task of predicting shortest paths in the context of the Warcraft Shortest Paths dataset, a scenario where traditional machine learning approaches may not be sufficient for accurate predictions.

To begin with, we look at an overview of the fundamental concepts of machine learning and combinatorial optimisation and investigate various architectural approaches, including machine learning-based architectures, two-stage approaches utilising both machine learning and combinatorial optimisation separately, and combined pipelines. To evaluate the effectiveness of these approaches, I apply them to the Warcraft Shortest Paths dataset and analyse their performance using relevant metrics such as loss and gap.

The main contribution to this topic includes a comprehensive comparison of these three approaches applied to the same dataset, offering insights into their advantages, disadvantages, and potential improvements. This analysis also includes a brief discussion regarding the importance of the metrics used for evaluating different architectures, seen through the lens of the optimality gap and losses.

1 Introduction

1.1 Motivation

Complex combinatorial optimisation problems have become increasingly prevalent in the modern world, leaving us with no option but to solve them as efficiently and effectively as possible. One way of achieving this is by combining available tools, such as the use of combinatorial optimisation in conjunction with machine learning. By utilising the strengths of both sets of tools, it is possible to find optimised solutions to combinatorial optimisation problems in an efficient, and potentially effective, manner.

One practical application of this approach is the use of live, new satellite imagery to detect potential shortest paths for road navigation without the need for resource-intensive mapping. This process involves using computer vision to detect roads from satellite imagery and combinatorial optimisation to find the shortest paths. Such a data pipeline is commonly used in machine learning applications, and it is what I aim to achieve in this example.

However, it is important to note that pure machine learning approaches are not always effective at making predictions for problems that have embedded optimisation problems, as their loss does not always target the specific goal at hand [24]. Therefore, it is crucial to explore alternative solutions, such as two-stage approaches, where the algorithms makes predictions based on various features then solves the optimisation problem, or combined pipelines that are specifically designed with combinatorial optimisation problems in mind.

1.2 Overview

In this thesis, I will delve into the various tools and techniques used to solve complex combinatorial optimisation problems, particularly those that may require the use of machine learning due to their complexity and the need for blackbox combinatorial solvers. Beginning with Section 2, I will provide an overview of the fundamental concepts of machine learning and combinatorial optimisation. Specifically, in Section 2.1, I will examine the basic principles of machine learning and in Section 2.2, I will explore the role of combinatorial optimisation and the types of problems that it can be applied to.

In Section 3, I will investigate different architectural approaches that can be used to make predictions for problems that include optimisation, such as the one described in Section 1.1. This includes the use of architectures composed solely of machine learning layers in Section 3.4, two-stage approaches that use both machine learning and combinatorial optimisation separately in Section 3.5, and finally, combined pipelines in Section 3.6.

To understand the effectiveness of these different architectures, it is important to evaluate them using a dataset. In Section 3.1, I will discuss the significance of the metrics used for evaluation and how they differ. The results of this evaluation will be presented in Section 4.2, providing insight into the effectiveness of these various approaches for solving combinatorial optimisation problems that may require the use of machine learning.

1.3 Contributions

There are a variety of papers that have been published in recent years showcasing new techniques for dealing with prediction problems that integrate optimisation. However, these papers generally compare the pure machine learning approach to the two-stage approach, or the two-stage approach to the combined machine learning and combinatorial

optimisation architectures. Additionally, there is a lack of standardisation when it comes to the datasets being tested, making it difficult to determine the most effective methods.

In this thesis, I will examine all three approaches applied to one dataset, the Warcraft Shortest Paths dataset [22], which was introduced in Vlastelica et al. (2019) [23] and further used in Dalle et al. (2022) [3]. This will provide a baseline for comparing the different approaches to solving blackbox combinatorial problems. Additionally, I will explore how metrics such as the loss and the gap play a role in determining the effectiveness of an architecture. Furthermore, I will provide an in-depth analysis of the advantages and disadvantages of each approach and provide insights into how they can be improved.

Now that we have established the context and significance of the problem at hand, we will move on to the foundations section. This section will provide a deeper understanding of the fundamental concepts of machine learning and combinatorial optimisation, which will serve as the building blocks for the architectural approaches that I will discuss later on. By delving into the basics of these tools, we will be better equipped to evaluate and understand the different approaches for predicting blackbox combinatorial problems in an efficient and effective manner.

2 Foundations

In order to gain a deep understanding of the underlying concepts and techniques used to solve blackbox combinatorial problems in tandem with machine learning, we must first obtain a sufficient grasp of the foundational knowledge of both machine learning and combinatorial optimisation. This sort of knowledge will give us an understanding of both tools, what problems they solve, and how they solve them.

In Section 2.1, I will examine what machine learning is, and the different techniques and algorithms used in it, such as supervised and unsupervised learning, neural networks, and deep learning. I will also explore the various applications of machine learning and its limitations.

Moving on, and in Section 2.2, I will delve into the field of combinatorial optimisation, which includes the types of problems that it can be applied to and the various techniques used to solve them. I will also examine the different approaches used in combinatorial optimisation and their advantages and disadvantages. By obtaining a thorough understanding of these foundational concepts, we will be better equipped to evaluate and understand the different approaches for predicting blackbox combinatorial problems in an efficient and effective manner, with the help of machine learning for automated decision-making and improved results.

2.1 Machine Learning

2.1.1 Motivation

The use of machine learning has become increasingly prevalent in recent years due to its ability to solve complex, large-scale problems through learning. With the rise of big data and the increasing need for automated decision-making, machine learning has become a crucial tool for solving otherwise difficult problems, ones that traditional algorithms may struggle to solve, or problems which would require a great deal of effort.

Machine learning algorithms are designed to gain knowledge and adapt to changes in data, making them viable solutions for tasks such as predictions, classifications, and

automation. These algorithms can be trained on vast amounts of data and continue to improve their performance over time, making them ideal for handling large sets of data and making automated decisions with high accuracy and precision.

How the algorithm learns could also differ, wherein the idea of gaining knowledge in machine learning could either come from supervised, unsupervised or reinforcement learning. With supervised learning, also called learning by imitation, an algorithm is given a specific task and a set of labels, while in unsupervised learning, the algorithm is given a set of data and has to begin to learn the labels by itself. Reinforcement learning, also called learning by experience, on the other hand, involves an algorithm interacting with an environment and receiving rewards or penalties based on its actions. Each of these categories has its own advantages and disadvantages, and applying them would depend on the task at hand. For the scope of this paper, my main focus will be on learning through imitation, also known as supervised learning. Through this approach, the algorithm is provided with a specific task and a labelled dataset, allowing it to learn from examples and improve its performance over time. This learning method is particularly useful for classification and regression tasks, thus making it useful for assigning costs to different segments of a map.

2.1.2 Neural Networks

One of the most basic forms of neural networks is the Perceptron, which is capable of solving binary classification problems [19]. However, when dealing with image classification problems, the Perceptron's capabilities are limited. This is where the use of a non-linear activation function, such as the softmax function, comes into play.

When considering the outputs of an image classification problem, $y(x)$, a Perceptron must learn a weight vector $\mathbf{w}$ and apply it to the input vector $\mathbf{x}$. Additionally, there is an element of bias, denoted as b, which is incorporated into the linear equation

$$y(\mathbf{x}) = \mathbf{w}^{\intercal}\sigma(\mathbf{x}) + b \tag{1}$$

Without the bias, the decision boundary would always pass through the origin, which may not be the optimal location for separating the classes in the data. The bias term allows for more flexibility in adjusting the position of the decision boundary, improving the model's ability to fit the data and make accurate predictions [19]. This linear equation, shown in Equation 1, separates a linear space into two groups, such that any point on one side of the line created by the equation belongs to one class, while any point on the other side belongs to the other class.

It is important to note that this equation further incorporates an activation function, denoted as $\sigma(\mathbf{x})$, which is used to introduce non-linearity into the model. Without the use of a non-linear activation function such as the softmax function, the model would only be able to capture linear relationships in the data, limiting its ability to generalise to unseen data [2]. The softmax function (seen in Figure 2), also known as the normalised exponential function, is commonly used in image classification problems as it provides a probability distribution over the possible classes. The function takes in the output of the final fully connected layer, also known as the logits, and maps it to a probability distribution over the classes.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{2}$$

Where x_i is the output of the final fully connected layer for class i, and k is the total number of classes. The softmax function provides a probability score for each class, making it an ideal choice for image classification problems where the goal is to assign an image to one of several classes [18].

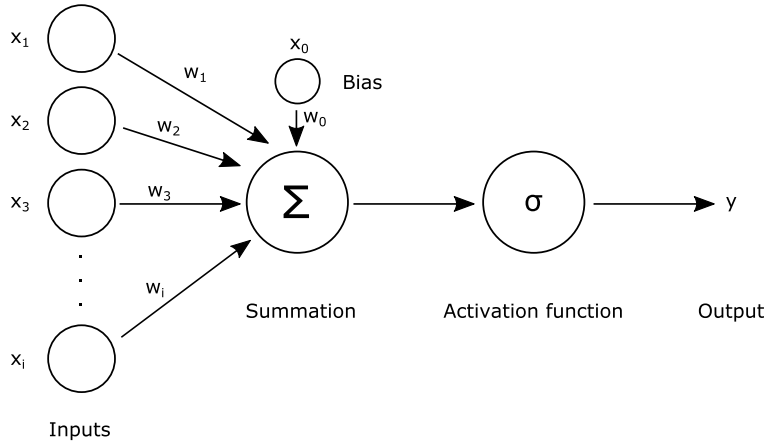

Figure 1: The idea behind a simple perceptron. Here we see how the inputs are combined with the bias, then put through an activation function, which is usually non-linear.

Determining the weight vector is a crucial aspect of this entire process, seeing that this determination of the weights and errors is what one would consider as *learning*. The calculation of the weights is achieved through the use of loss functions, also known as error or cost functions, which measure the difference between the predicted output and the actual output. These functions guide the learning process of the neural network by minimising this difference. The ideal loss function for classification would adjust the weights whenever an error is made and leave them unchanged otherwise.

In order to minimise the difference between the real output and the prediction, we must be able to find a global, or local, minimum for the loss function, implying that it should also be differentiable. Nonetheless, the ideal misclassification error function is non-differentiable, thus alternative error functions that are as close as possible to the ideal must be sought, while still resulting in a similar output.

There have been various differentiable loss functions presented over time, such as the mean squared error, cross-entropy, and hinge loss [11]. Each of these loss functions has its own advantages and disadvantages, and their choice of use depends on the specific problem at hand. For example, mean squared error is commonly used in regression problems, while cross-entropy is typically used in classification problems. For the sake of this paper, I will focus on the mean squared error. The mean squared error, also known as the quadratic loss, measures the difference between the predicted output and the actual output by squaring the difference and taking the average over all instances

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3}$$

where y is the actual output, $\hat{y}$ is the predicted output, and n is the total number of instances [11]. This loss function is widely used in regression problems because it is differentiable and has a global minimum, which makes it easy to optimise using gradient descent. One of the problems with the mean square error is its sensitivity to outliers,

which results from the fact that it heavily penalises large errors. This can lead to overfitting, which occurs when a model is trained too well on the training data, to the point where it starts to memorise the training data instead of generalising to new unseen data. Consequently, this also leads to high variance in the model's predictions.

Combining the aforementioned tools, namely a single-layer neural network along with an activation function and a loss function is a common approach in many machine learning tasks. In our case, where we used the mean squared error (MSE) and the softmax function, we could use this combination in solving multi-class regression problems. However, this approach may not be sufficient for more complex image classification and segmentation tasks. In these scenarios, it may be necessary to assign different costs to different segments of the image or to take into account relationships between different segments. This is where the Perceptron, with its single layer of neurons, may not be enough. To capture these more complex relationships and patterns, it may be necessary to use multiple layers of neurons, also known as multi-layer perceptrons (MLPs). This allows for a deeper level of abstraction and a greater ability to model complex relationships in the data [2].

2.1.3 Multi-Layer Perceptrons

In Section 2.1.2, I discussed the basic principles of a single-layer perceptron and its limitations in capturing complex relationships in data. To overcome these limitations and improve the ability of the model to generalise to unseen data, we turn to multi-layer perceptrons (MLP). An MLP is a type of feedforward neural network that consists of multiple layers of interconnected perceptrons. Each layer, also known as a hidden layer, is composed of multiple neurons, each with its own set of weights and biases. These layers work together to extract features from the input data, making it possible to model more complex relationships in the data [10].

As we begin to model more complex relations, we also introduce more complex functions that the neural networks maps. The complexity of those functions and their number of inputs makes it more difficult to simply differentiate and minimise said functions, making us rely more on other means in order to achieve gradient descent. The main idea here would be to introduce an error function, $E(\mathbf{W})$, such that the function incorporates a loss $L(x)$ and a regulariser $\Omega(x)$, such as the mean squared error loss $L(y(\mathbf{x};\ \mathbf{W})) = MSE(y(\mathbf{x};\ \mathbf{W}), \hat{y})$ in Equation 3 and the regulariser

$$\Omega(\mathbf{W}) = \frac{\lambda}{2}\|\mathbf{W}\|^2 \tag{4}$$

Where λ is the regularisation parameter, and $\|\mathbf{W}\|$ is the L2-norm of the weight vector [11]. This specific regulariser is commonly known as weight decay, and helps to prevent overfitting by penalising large weight values, which is done by adding the regulariser term to the overall error function and optimising for both the loss and the regularisation term simultaneously.

Using this new tool, we should be able to begin calculating and updating the weights $W_{ij}^{(k)}$ by descending the entire function in the direction of the steepest decline $\frac{\delta E(\mathbf{W})}{\delta W_{ij}^{(k)}}$ in a process known as backpropagation. Such a process allows us to efficiently update the weights in our neural network in order to minimise the error function $E(\mathbf{W})$ through iterative application, thus training our network to accurately model complex relations using the data input.

The idea behind backpropagation would be to compute the gradients of the error function with respect to each layer in the network. This is done by starting at the output layer, computing the gradient with respect to the weights there, and then propagating this gradient backwards through the layers, computing the gradients for each weight as we go [20].

In essence, what we end up with is a forward pass, which computes the output of the neural network for a given input, and a backward pass, which computes the gradients of the error function with respect to the weights in the network. The algorithms for both of which can be seen in Algorithms 1 and 2 [10].

Algorithm 1 Forward Pass in MLP Backpropagation

Require: Input features $\mathbf{x}$, weight matrices $\mathbf{W}^{(k)}$, activation functions g_k, loss function L, regularisation parameter σ, regularisation function Ω
Ensure: Output predictions $\mathbf{y}$, total error E
$\mathbf{y}^{(0)} = \mathbf{x}$
for $k = 1, \ldots, l$ do **do**
$\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{y}^{(k-1)}$
$\mathbf{y}^{(k)} = g_k(\mathbf{z}^{(k)})$
end for
$\mathbf{y} = \mathbf{y}^{(l)}$
$E = L(\mathbf{t}, \mathbf{y}) + \lambda\Omega(\mathbf{W})$

Algorithm 2 Backward Pass in MLP Backpropagation

Require: Weight matrices $\mathbf{W}^{(k)}$, output predictions $\mathbf{y}^{(k)}$, input of layer k $\mathbf{z}^{(k)}$, loss function $L(\mathbf{t}, \mathbf{y})$, $\sigma\Omega(\mathbf{W})$
Ensure: Gradient of the loss function with respect to the weight matrix $\frac{\delta E}{\delta \mathbf{W}^{(k)}}$
$\mathbf{h} \leftarrow \frac{\delta E}{\delta \mathbf{y}} = \frac{\delta}{\delta \mathbf{y}} L(\mathbf{t}, \mathbf{y}) + \sigma \frac{\delta}{\delta \mathbf{y}} \Omega$
for $k = l, l-1, \ldots, 1$ **do**
$\mathbf{d}i \leftarrow \mathbf{d}i + 1 \times \mathbf{W}i + 1^T \times f'(\mathbf{Z}_i)$
$h \leftarrow \frac{\delta E}{\delta \mathbf{z}^{(k)}} = \mathbf{h} \cdot g'(\mathbf{y}^{(k)})$ ▷ Here, $\cdot$ is the element-wise product
$\frac{\delta E}{\delta \mathbf{W}^{(k)}} = \mathbf{h}\mathbf{y}^{(k-1)\intercal} + \sigma \frac{\delta \Omega}{\delta \mathbf{W}^{(k)}}$
$\mathbf{h} \leftarrow \frac{\delta E}{\delta \mathbf{y}^{(k-1)}} = \mathbf{W}^{(k)\intercal}\mathbf{h}$
end for

With $y_j^{(k)}$ being the output of layer k and $z_j^{(k)}$ being the input of layer k. These two algorithms work in tandem by first performing the forward pass, where the input features are processed through the layers of the network using the weight matrices and activation functions. The output predictions, total error, and input of each layer are calculated during this pass. Then, in the backward pass, the gradient of the loss function with respect to the weight matrices is calculated by working backwards through the layers, using the output predictions, input of each layer, and the loss function.

To put it all into perspective, in order for a neural network to learn, the weights and biases of the network must be defined and optimised. This is accomplished by utilising a loss function, which calculates the discrepancy between the predicted output of the network and the actual output, as determined by labeled data in supervised learning. Achieving the most accurate neural network possible means that the output of the loss

function must be minimised through the use of gradient descent, particularly when dealing with large scale neural networks. Backpropagation facilitates this process by facilitating the calculation of the gradient of the loss function with respect to the weight matrices, by working backwards through the layers of the network. Overall, a neural network essentially works by combining three different layers of complex mathematical tools, each helping adjusted its core weights to achieve more accurate prediction on unseen data.

Despite their effectiveness, there still exists several limitations to Multi-Layer Perceptrons that can affect their performance [9]. One major limitation that remains to be addressed is how MLPs suffer from the vanishing gradient problem, which occurs when the gradients of the loss function become too small during the backward pass, making it difficult for the network to update its weights and biases. This can lead to slow convergence or even stagnation in the training process. To overcome this limitation, researchers have developed a newer type of neural network called Convolutional Neural Networks (CNNs), which have been proven to be very successful in image recognition tasks, and are able to effectively learn features from images.

2.1.4 Convolutional & Residual Neural Networks

As previously noted in Section 2.1.3, Multi-Layer Perceptrons have been demonstrated to effectively classify input data, however, are limited by the vanishing gradient problem. In contrast, Convolutional Neural Networks (CNNs) have been specifically designed for image and video processing tasks, which are particularly relevant for this study due to the nature of the dataset in Section 3.3 being comprised of image-related problems. In this section, I will delve into the intricacies of CNNs, including their architecture, training process, and common variants such as ResNet.

The architecture of Convolutional Neural Networks, specifically designed for image-related problems, incorporates convolutional and pooling layers to facilitate the extraction and preservation of significant features in an image. The training process of a CNN follows the standard procedure of forward and backward passes and gradient descent optimisation, as seen in Multi-Layer Perceptrons. However, the architecture of a CNN allows for the adaptation and refinement of the weights and biases of the filters and kernels in the convolutional layers, resulting in improved performance in image classification and analysis [17].

Looking into the reason why Convolution Neural Networks work better than Feedforward Neural Networks in image classification, we begin to see that as a problem becomes more complex, it will require significantly more hidden layers in an MLP in order to mirror the function that could solve the problem. In a standard neural network with two hidden layers, for example, an image with a resolution of 24 by 24 pixels would require approximately $24 + 2 \cdot 24^2$ parameters, which can result in a challenging training and initialisation process, particularly in terms of computational time and power. In order to minimise the number of parameters and accommodate deeper networks capable of extracting more features, CNNs employ the use of convolutional layers.

Convolutional layers work by sliding a filter, also known as a kernel, across the input image and performing an element-wise matrix multiplication between the values of the filter and the input image. This produces a new matrix known as the feature map, which highlights the important features within the image that the filter was looking for. In a Feedforward Neural Network, such features would not actually be a flat representation of the input data, losing the spatial structure and relationships between the pixels in

the process. Convolutional layers allow for the preservation of the spatial structure and relationships between the pixels, enabling the network to learn more meaningful features in the image data, leading to better classification results.

On top of the convolutional layers, CNNs utilise pooling layers, which serve the purpose of down-sampling the feature maps produced by the convolutional layers. This helps to reduce the spatial dimensionality of the feature maps and make the features more robust and invariant to small translations in the input, which in turn reduces the number of parameters in the network and helps prevent overfitting by making the features more generalised [17]. The most commonly used pooling operation is max-pooling, where the maximum value in a region of the feature map is taken and used as the new value for that region.

The convolutional and max-pooling layers are repeated in the neural network, building up abstract representations and extracting increasingly complex features. This is achieved through the use of increasingly sophisticated filters, or kernels, in each subsequent layer of the network. These kernels are able to capture increasingly complex relationships within the input data, leading to a more accurate representation of the features contained within the image.

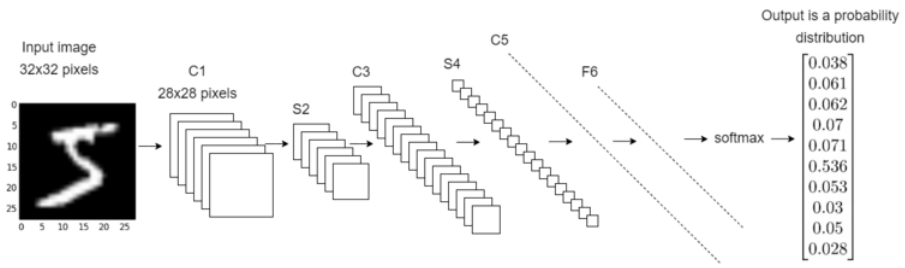

Figure 2: LetNet-5 architecture [25]. A simple Convolutional Neural Network introduced by LeCun et al. in 1998. This CNN includes three layers of pairs of convolutional and max-pooling layers, then a fully connected layer.

Subsequent to the sequential arrangement of convolutional and max-pooling layers, a fully connected layer is incorporated into the CNN architecture. This layer enables the classification of the features that have been extracted from the input images. The fully connected layer operates in a traditional manner, with each neuron connected to every neuron in the preceding layer, leading to the output of this layer being processed through a softmax activation function. The output of the activation function produces a probability distribution over the target classes, thereby indicating the class to which the input image belongs. These final steps collectively form the process of image classification through the use of a Convolutional Neural Network [17]. An example of such a CNN can be seen in Figure 2.

Despite CNNs being a major improvement on traditional MLPs when it comes to tasks like image classification, they still possess several limitations. One major issue with deeper networks is the phenomenon known as the vanishing gradient problem. As the depth of the network increases, the gradients become smaller and more difficult to propagate, leading to a decrease in performance and accuracy. This issue is due to the difficulty in optimising deeper models with multiple non-linear transformations.

Addressing these limitations, the introduction of Residual Networks (ResNets) [12] helped tackle some of the problems that CNNs face. ResNets are a modification of CNNs,

specifically designed to tackle the vanishing gradient problem that arises in deeper networks. Rather than using a conventional, linear network structure, ResNets incorporate a residual structure, such as the one seen in Figure 3, where the output of one layer is added to the input of the following layer. This structure facilitates the easier flow of gradients and enables simpler information propagation, even in deeper networks, thus allowing for the training of significantly deeper models without encountering the vanishing gradient problem. The residual structure has been shown to greatly improve the performance and accuracy of deep neural networks [12].

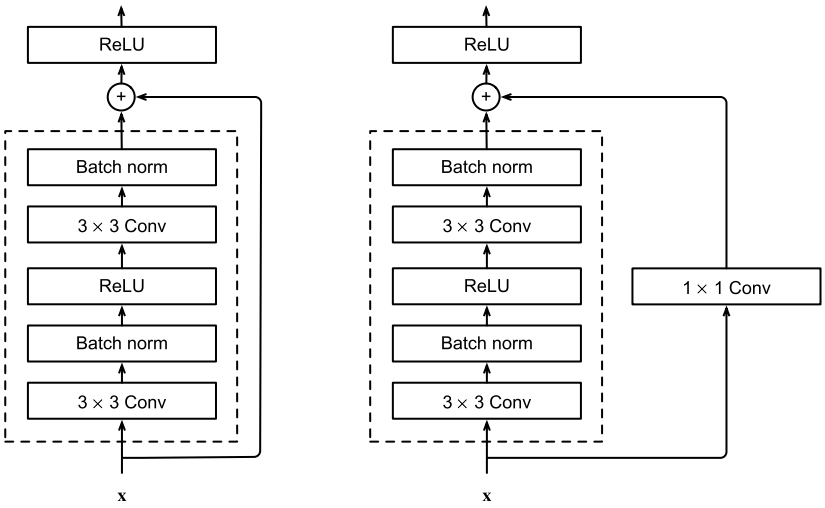

Figure 3: On the left, we have a simple Convolutional Neural Network. On the right is a Residual Network that uses the same Convolutional Neural Network as its foundation [26].

The residual blocks that make up ResNets allow the neural network to maintain information from early layers and build upon it, thus enabling accurate predictions even in deeper networks. However, research conducted by Veit et. al in 2016 [21] found that the effectiveness of ResNets is not solely dependent on their depth, but rather on the utilisation of ensembles of shallower networks with residual connections. These results highlight the crucial role that residual blocks play in constructing deep neural networks and demonstrate their potential for improving accuracy and performance in computer vision tasks [21].

With a solid foundation in the basics of machine learning and deep learning, as well as how basic computer vision tasks could be solved, we are now equipped to further build on our foundational knowledge in preparation for Section 3 through the introduction of combinatorial optimisation. As we move forward, I will introduce the concept of combinatorial optimisation and its applications in solving complex optimisation problems. Through this exploration, I will demonstrate why traditional computer vision techniques may not be adequate to tackle these optimisation challenges and suggest alternative approaches.

2.2 Combinatorial Optimisation

2.2.1 Motivation

In tackling today's optimisation problems, combinatorial optimisation helps address prevalent optimisation problems that may otherwise require a substantial amount of time and resources. Various mathematical techniques and algorithms have been developed under

that guise, such as graph algorithms and linear programming, both of which have proven to be powerful tools in solving optimisation problems in a wide range of domains.

Take, for example, the shortest paths problem, which is the exact problem I will be attempting to solve further on in 3, and which involves finding the path with the minimum distance between two nodes in a graph. Such a problem can be seen in a wide range of areas such as transportation and logistics, where the optimisation of routes can lead to substantial cost savings. Nonetheless, traditional algorithms for solving the shortest path problem can become computationally expensive as the size of the graph grows, making it vital to develop more efficient solutions. This is where combinatorial optimisation can prove to be valuable, as it allows for the efficient computation of solutions even in large and complex graphs.

In Section 2.2.2, I will delve deeper into the topic of graph algorithms and their use in finding optimal solutions to the shortest paths problem. To further expand our knowledge of combinatorial optimisation, I will also examine the concept of linear programming in Section 2.2.3, which provides another powerful tool for solving optimisation problems.

2.2.2 Graph Theory

Graph algorithms play a substantial role in solving optimisation problems such as the shortest paths problem. In this section, I will briefly examine two popular algorithms in the field, namely Dijkstra's algorithm and Bellman-Ford algorithm, both of which have been proven to be effective in finding optimal solutions to the shortest paths problem. Through this brief discussion, we aim to gain some understanding of how graph algorithms can be utilised in combinatorial optimisation and how they contribute to finding optimal solutions to the shortest paths problem, which will be my main focus here due to its relevancy to the problem discussed later on in Section 3.

To begin, consider a finite graph $G = (V, E)$, where $V = v_1, \ldots, v_n$ represents the set of vertices or nodes, and $E = e_1, \ldots, e_m$ represents the set of edges, denoting the connections between nodes. The cardinality of the sets V and E are denoted by $|V| = n \in \mathbb{N}$ and $|E| = m \in \mathbb{N}$, respectively. If the graph is directed, the relationship between two nodes u and v is represented by $e \in E$, such that $e = (u, v)$ indicates that there is a directed edge from source node u to target node v. In contrast, undirected graphs contain edges with no direction and can be represented as an unordered pair $e = (u, v)$. Additionally, edges in a graph can be assigned weights, which can represent various quantities such as distances, costs, or even time. The weight of an edge e is denoted as $w(e) \in \mathbb{R}$.

Using graph representations, we could solve problems such as the shortest paths problem through algorithms such as Dijkstra's algorithm, as seen in Algorithm 3 [5]. Dijkstra's algorithm, though only applicable to graphs where the edge weights are non-negative, is one of the most widely used algorithms for solving such a problem.

Dijkistra's algorithm operates by maintaining a set of vertices that have been processed and a set of vertices that have yet to be processed. The algorithm starts at the source node and processes each vertex in increasing order of its distance from the source node. This is accomplished by maintaining a priority queue of vertices, where the priority of each vertex is given by its distance from the source node. At each iteration of the algorithm, the vertex with the lowest priority (i.e. the closest to the source node) is removed from the priority queue and processed. The algorithm then updates the distances of all of the vertices adjacent to the processed vertex and inserts them into the priority queue. This process continues until either all vertices have been processed or the destination vertex

Algorithm 3 Dijkstra's Algorithm

Require:
$G = (V, E)$ ▷ A finite graph with vertex set V and edge set E
$s \in V$ ▷ The source node
$w : E \to \mathbb{R}$ ▷ The weight function that assigns a weight to each edge
Ensure:
d ▷ A mapping of nodes to the shortest distance from the source node

$d(s) \leftarrow 0$ ▷ The distance to the source node is 0
$Q \leftarrow V$ ▷ The priority queue that holds all nodes
while $Q \neq \emptyset$ **do** ▷ While the priority queue is not empty
  $u \leftarrow \text{extractMin}(Q)$ ▷ Extract the node with the smallest distance
  for each node v adjacent to u **do** ▷ Iterate over all adjacent nodes to u
    $alt \leftarrow d(u) + w(u, v)$ ▷ Calculate the alternative distance to node v
    if $alt < d(v)$ **then** ▷ If the alternative distance is smaller than the current distance
      $d(v) \leftarrow alt$ ▷ Update the distance to node v
      $\text{decreaseKey}(Q, v, d(v))$ ▷ Update the priority of node v in the priority queue
    end if
  end for
end while

has been processed. The final result of the algorithm is the shortest distance from the source node to the destination node [5].

Another commonly used algorithm in solving the shortest path problem is the Bellman-Ford algorithm, see in Algorithm 4. This algorithm differs from Dijkstra's algorithm in that it is applicable to graphs with negative edge weights [7].

The Bellman-Ford algorithm operates by iteratively relaxing the edges of the graph and updating the distances to the adjacent nodes; it begins at the source node and assigns it a distance of 0, with all the other nodes receiving distances of infinity. In the algorithm, we then select a node with the minimum distance and relax all the edges originating from that node, meaning that we update the distance to the adjacent node if the distance through the selected node is shorter than the previously known distance. This process is then repeated until all nodes have been processed and the shortest distances to all the nodes have been found. Such an algorithm uses a dynamic programming approach to update the shortest distance of all nodes in the graph, which results in a time complexity of $O(n \cdot m)$ [7], where n is the number of nodes and m is the number of edges in the graph.

In comparison to the Bellman-Ford algorithm, Dijkstra's algorithm has a time complexity of $O(m + nlogn)$ [5], making the algorithm more efficient in most cases. Nonetheless, the Bellman-Ford algorithm has the advantage of being able to handle negative edge weights, which Dijkstra's algorithm cannot handle. Furthermore, in network routing scenarios where decentralised computation of shortest paths is required, the Bellman-Ford algorithm is more suitable due to its capability of performing local updates to node distances in a distributed manner, thus enhancing its usefulness in such circumstances. An example of that is the Routing Information Protocol, a distance-vector routing protocol based on the Bellman–Ford and the Ford–Fulkerson algorithms [13].

Further graph algorithms exist that help solve problems such as the maximum flow,

Algorithm 4 The Bellman-Ford Algorithm

Require:
$G = (V, E)$ ▷ A finite graph with vertex set V and edge set E
$s \in V$ ▷ The source node
$w : E \to \mathbb{R}$ ▷ The weight function that assigns a weight to each edge
Ensure:
d ▷ A mapping of nodes to the shortest distance from the source node
$\forall v \in V, d(v) \leftarrow \infty$ ▷ Initialise all distances to infinity
$d(s) \leftarrow 0$ ▷ The distance to the source node is 0
for $i = 1$ to $|V| - 1$ **do** ▷ Iterate $\|V\| - 1$ times
  for $(u, v) \in E$ **do** ▷ Iterate over all edges in E
    $alt \leftarrow d(u) + w(u, v)$ ▷ Calculate the alternative distance to node v
    if $alt < d(v)$ **then** ▷ If the alternative distance is smaller than the current distance
      $d(v) \leftarrow alt$ ▷ Update the distance to node v
    end if
  end for
end for

minimum cut, and minimum spanning tree. These algorithms can be modified to suit specific scenarios and are constantly evolving, making graph theory a dynamic and constantly developing area of study. By utilising these algorithms, we can effectively solve complex network problems, such as the minimum distance problem on Warcraft maps, which will be discussed in Section 4.

2.2.3 Linear Programming

Linear Programming (LP) is a mathematical optimisation technique used to find the best solution in a scenario where there are a set of constraints and objectives that need to be satisfied. This method involves finding the optimal values for a set of variables subject to a set of linear constraints, allowing for the efficient allocation of resources in various real-world applications. In linear programming, the objective is to either maximise or minimise a linear function of the decision variables, known as the objective function. The constraints in linear programming are represented as linear equations, ensuring that the solution adheres to all restrictions. A linear program would, therefore, exist in the canonical form shown in Equation 5 [1].

$$\begin{aligned} \min \quad & \mathbf{c}^\intercal \mathbf{x} \\ \text{s.t.} \quad & A\mathbf{x} \geq \mathbf{x} \\ & \mathbf{x} \geq \mathbf{b} \end{aligned} \tag{5}$$

Where $\mathbf{x} \in \mathbb{R}^n$ is the decision vector of variables, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, $\mathbf{b} \in \mathbb{R}^m$ is the constraint vector, and $\mathbf{c}^\intercal \in \mathbb{R}^n$ is the objective function vector in transpose form. The scalar product of the objective function and decision vector, $\mathbf{c}^\intercal \mathbf{x}$, represents the objective function to be minimised. The solution of this linear program aims to find the optimal value of the objective function that satisfies all constraints, represented by the inequality constraints $A\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{b}$.

Linear programs can be translated from graph problems by formulating the problem as a system of constraints and objectives. In the case of the shortest distance problem, as an example, the decision variables can be represented by the lengths of the edges in the graph. The constraints in the linear program can be represented by the conditions that must be satisfied for the lengths of the edges to represent the shortest distance between two nodes. For example, a constraint can be the requirement that the sum of the lengths of the edges along a path from node i to node j is less than or equal to a certain value. Equation 7 shows how it can be represented in a concrete linear program.

$$\sum_{(i,j)\in E} x_{i,j} \leq d_{i,j} \tag{6}$$

Where $x_{i,j}$ is the length of the edge from node i to node j, and $d_{i,j}$ is the shortest distance between nodes i and j. The objective of the linear program is then to minimise the sum of the lengths of all edges in the graph, which can be represented as in Equation 7.

$$\min \quad \sum_{(i,j)\in E} x_{i,j} \tag{7}$$

By formulating the shortest distance problem as a linear program, we can use linear programming algorithms to find the optimal solution efficiently and effectively. Knowing this, we also need to know what a solution might look like: A feasible solution of a linear program is a solution that satisfies all constraints of the problem, which are represented by the inequalities $A\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{b}$ in Equation 5. On the other hand, an infeasible solution is one that does not satisfy all constraints and thus does not meet the requirements of the problem.

Additionally, an optimal solution is a feasible solution that also provides the best possible value for the objective function, $\mathbf{c}^\intercal\mathbf{x}$. In other words, it is the solution that minimises the objective function while also satisfying all constraints of the problem. This is the goal of solving a linear program, to find an optimal solution that balances feasibility and optimality.

To gain a deeper understanding of the workings of linear programs and the solution space, it is advisable to examine a graphical representation. An example of this can be seen in Figure 4, which illustrates the feasible solution for the linear program outlined in Equation 8 [1]. The optimal solution on the graph can be found by considering the set of all points whose cost, represented as $\mathbf{c}'\mathbf{x}$, is equal to a scalar z. This relationship can be depicted as the line described by the equation $-x_1 - x_2 = z$.

$$\begin{aligned} \min \quad & -x_1 - x_2 \\ \text{s.t.} \quad & x_1 + 2x_2 \leq 3 \\ & 2x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{8}$$

It is important to note that this line is perpendicular to the vector $\mathbf{c} = (-1, -1)$. Different values of z correspond to different lines, all of which are parallel to one another. As z increases, the line $z = -x_1 - x_2$ moves in the direction of the vector $\mathbf{c}$. As the objective of linear programming is to minimise z, it is desirable to move the line as much

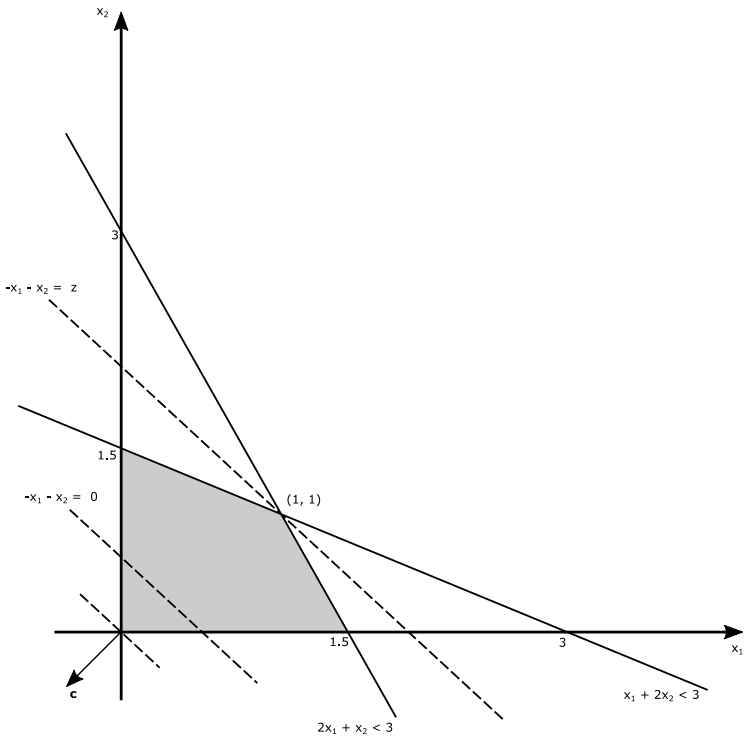

Figure 4: Graphical solution for Equation 8 [1].

as possible in the direction of $-\mathbf{c}$ without leaving the feasible region. The best possible value of z that can be achieved is $z = -2$, and the vector $\mathbf{x} = (1, 1)$ represents the optimal solution [1].

When it comes to linear programs with higher dimensions, the same principle applies. The solution space becomes a hyperplane in a higher dimensional space, and the optimal solution is found at the intersection of the feasible region and the hyperplane with the highest value of z. In this case, the line described by $-x_1 - x_2 = z$ is replaced by a hyperplane described by $\mathbf{c}'\mathbf{x} = z$, where $\mathbf{c}'$ is the transpose of the vector $\mathbf{c}$ and $\mathbf{x}$ is the vector of variables. The hyperplane is perpendicular to the vector $\mathbf{c}$, and the optimal solution is found at the intersection of the feasible region and the hyperplane with the highest value of z. In higher dimensions, the feasible region can be described by a set of constraints, such as linear equations and inequality constraints. The process of finding the optimal solution involves moving the hyperplane along the direction of $-\mathbf{c}$ until it reaches the boundary of the feasible region. The intersection point between the hyperplane and the boundary is the optimal solution for the linear program.

With an in-depth understanding of the graphical representation of linear programs, it becomes possible to systematically solve these problems using a well-established algorithm. The solution area of a linear program, as illustrated in Figure 4, can be visualised as a polyhedron, within which the set of feasible solutions exists. Recognising this geometric structure, George Dantzig developed the simplex algorithm to systematically search for the optimal solution within the feasible region [4].

The simplex algorithm begins by starting at a feasible solution, specifically one of the corners, which is considered a feasible solution in and of itself. From there, the algorithm iteratively moves to adjacent corners that result in improved solutions until it reaches the optimal solution. This is achieved by exploiting the geometric structure of the problem and

making a series of pivots that improve the solution until the optimal solution is reached [1].

The pivoting operation, which is central to the simplex algorithm, involves the selection of an appropriate corner and the pivot to the adjacent corner with the greatest potential for improvement. This process is repeated until the optimal solution is reached, or the algorithm determines that no further improvement can be made. The simplex algorithm represents a systematic and efficient approach to solving linear programs, leveraging the geometric structure of the problem to find the optimal solution [1]. Algorithm 5 shows how the simplex algorithm works in concrete terms.

Algorithm 5 Simplex Algorithm

1: **Input:** Linear program in standard form
2: **Output:** Optimal solution
3: Initialise the simplex tableau with the linear program in standard form
4: **while** there exists a negative value in the last row (the objective function) **do**
5: Choose a pivot column by selecting the most negative value in the last row
6: Choose a pivot row by selecting the minimum ratio between the last column and the pivot column
7: Perform row operations to transform the pivot row into an identity matrix
8: Perform row operations on all other rows to zero out the pivot column
9: **end while**
10: The optimal solution is the values in the last column corresponding to the basic variables

Keep in mind, however, that the simplex algorithm is not guaranteed to find the optimal solution in a finite number of steps, and may not always converge to the optimal solution [1]. In such cases, the algorithm may need to be modified or an alternative method may need to be used to find the optimal solution. Additionally, while the simplex algorithm is effective for solving small to medium-sized linear programs, larger and more complex problems may require alternative methods, such as interior-point methods, to find the optimal solution in a reasonable amount of time.

Both linear programming and graph theory play a significant role in the field of combinatorial optimisation. They are powerful tools that can be used to solve complex problems. As I move on to the main work, it will become evident that the combination of combinatorial optimisation and machine learning has the potential to yield optimised solutions for these problems at a much larger scale, one which combinatorial optimisation cannot solve by itself. In the next section, I will delve into the details of combining these fields of machine learning and combinatorial optimisation, as well as how that combination works in different settings and architectures.

3 Solving Blackbox Combinatorial Problems with Machine Learning

The field of combinatorial optimisation has seen numerous advances in recent years, with the integration of machine learning as a promising solution to tackle complex problems. In this section, I will delve into the details of using machine learning to solve blackbox combinatorial problems. This section will cover various metrics for evaluating the effectiveness

of machine learning models, the problem statement and the dataset used for my analysis. I will also examine different machine learning architectures and two-stage approaches to ML-CO problems. Finally, I will present a combined ML-CO pipeline.

3.1 Metrics

In the field of combinatorial optimisation, it is crucial to have a clear understanding of what constitutes an "effective" algorithm. This understanding is essential in determining the success of algorithms in solving complex combinatorial problems. This section focuses on examining the various metrics used to evaluate the performance of algorithms that solve combinatorial optimisation problems and emphasises the one that best captures the effectiveness of these algorithms.

One of the commonly used metrics for evaluating the performance of algorithms in combinatorial optimisation problems is the optimality gap, which is defined as the relative distance between the solution obtained from an algorithm and the ground truth optimal solution. This metric provides a single value that summarises the performance of the model on a specific problem and gives insight into how close the model is to the optimal solution and how efficient it is in finding the best solution. However, the optimality gap only measures the combined performance of both the learning and search components, making it difficult to determine where poor results may originate from, such as insufficient learning ability or weak search mechanisms [8]. To address this issue, there is a need for a second metric that is capable of evaluating the accuracy of the learning component against ground truth, independently of the search component [8].

The second metric commonly used in evaluating the performance of machine learning architectures used in combinatorial optimisation problems is losses. This metric represents the difference between the prediction made by the model and the actual ground truth label. Losses are calculated using various loss functions, such as mean squared error mentioned in Section 2.1.2 and Equation 3, which allow the model's accuracy to be measured in terms of its ability to learn the underlying relationship between the input and output data. However, while losses in machine learning can provide useful information on the accuracy of the model's learning component, they also cannot be used as the sole metric to evaluate the performance of algorithms in combinatorial optimisation problems. This is because the objective in combinatorial optimisation problems is to find the optimal solution, not just to fit the data.

However, relying solely on either the loss or the optimality gap is not sufficient in evaluating the performance of algorithms that solve combinatorial optimisation problems. This is why it is important to use both metrics to gain a more comprehensive understanding of the algorithm's performance. The optimality gap measures the difference between the solution provided by the algorithm and the optimal solution and is a relative measure, making it an appropriate metric for evaluating the effectiveness of architectures in solving combinatorial optimisation problems. On the other hand, the loss measures the accuracy of the model's learning component. By combining the information provided by both metrics, a more complete evaluation of the algorithm's performance can be obtained. This can help improve the algorithm's performance over time and lead to the development of more effective algorithms for solving combinatorial optimisation problems.

3.2 Problem Statement

Machine learning algorithms have been instrumental in transforming the way real-world problems are addressed, by enabling the automatic discovery of predictive models from historical data. The process begins with the collection and observation of relevant data, serving as the foundation for generating predictions. These predictions play a crucial role in informing decision-making. The data-to-prediction-to-action pipeline results in well-calibrated forecasts and diagnoses, providing recommendations for action in the real world [14][15]. The likelihoods produced by the predictive models form the basis for decision analysis, weighing the costs and benefits of various actions.

The pipeline for solving combinatorial optimisation problems at scale typically involves a two-stage approach, which separates the prediction and decision-making processes. Firstly, a machine learning model is used to make predictions based on a given dataset, and then these predictions are fed into a tailored optimisation algorithm that makes decisions. However, this approach creates a disconnect between the two layers as they operate based on different goals. In practice, solving real-world problems requires simultaneous consideration of both objectives; obtaining the right parameters for the optimisation algorithm while ensuring appropriate decision-making. Any mistake in the input parameters will result in a suboptimal solution, emphasising the importance of seamless integration between the prediction and decision-making processes.

One of the primary difficulties in combinatorial optimisation is that maximising accuracy, or minimising the loss, does not necessarily result in maximising the quality of the decision, or minimising the optimality gap, as outlined in Section 3.1. To address this challenge, one potential approach is to optimise based on the optimality gap as a measure of loss. However, this presents a problem in that the machine learning architecture must optimise a step-wise function, which can be difficult and computationally intensive. As previously discussed in 2.1.2, it is necessary to optimise a differentiable loss function that the machine learning model can learn from and adjust. Finding an appropriate differentiable loss function that accurately represents the optimality gap, however, can be challenging and may not always yield optimal results. An alternative solution is to incorporate the optimisation algorithm into the training loop, enabling the two to learn from each other and improve in real time.

In order to gain a deeper understanding of this challenge, I will examine three distinct architectures: a pure machine-learning architecture consisting of a ResNet18 [12], a two-stage architecture in which a ResNet18 is utilised to make predictions that are subsequently passed to a blackbox combinatorial solver that makes the decisions, and a combined pipeline incorporating novel loss functions that integrate the concept of blackbox combinatorial solvers into its fundamental mechanism. These architectures will be tested on a specific problem, namely the shortest paths problem in Warcraft maps, which will be discussed in further detail in Section 3.3.

3.3 Dataset

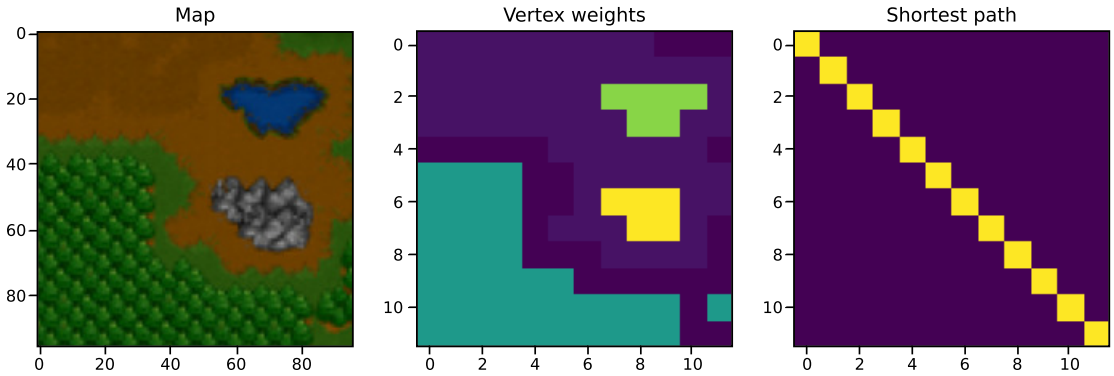

Figure 5: An overview of the Warcraft Shortest Paths dataset, where a map is taken, its vertex weights are calculated based on the terrain, and lastly the shortest path is computed based on those weights [22].

The Warcraft Shortest Path dataset [22] consists of randomly generated maps of varying sizes with the optimal shortest path from the top left vertex to the bottom right vertex. These maps are represented in Figure 5 with three different visualisations. The first version is the normal map, which simply displays the different types of tiles present in the map and their arrangement. The second visualisation is the vertex weights, which represent the cost of traversing each tile in the map. This cost is calculated based on the type of tile present at a particular vertex. Finally, the third visualisation is the shortest path, which represents the optimal path from the top left vertex to the bottom right vertex based on the calculated vertex weights.

To calculate the vertex weights, the cost of traversing each tile is taken into consideration. The cost of each tile is determined based on the type of tile present at a particular vertex. For example, if the tile is a dirt road tile, the cost is relatively low, while if it is a rocky tile, the cost is relatively high. These costs are then used to calculate the vertex weights, which are then used to calculate the shortest distance.

The Warcraft Shortest Path dataset is useful in providing a visual representation of the relationships between the normal map, vertex weights, and shortest path. This dataset can be used as a test case for algorithms that seek to find the shortest path in a given map, and it is an excellent demonstration of how the combination of the normal map and vertex weights can be used to find the optimal path from one vertex to another.

Lastly, the shape of the training images is represented by $(10000, 96, 96, 3)$, while the training labels and weights have the shape of $(10000, 12, 12)$. Here, 10000 represents the number of images in the training set, 96 and 96 represent the height and width of each image in pixels, and 3 represents the number of colour channels in each image, i.e., red, green, and blue (RGB). The shape of the training labels and weights, $(10000, 12, 12)$, indicates that there are $10,000$ samples in both the ground truth shortest path information and weight sets. Each sample is represented as a matrix of 12×12.

3.4 Machine Learning Architecture

One of the most straightforward ways to find the shortest path for this problem is to simply use computer vision tools. To create the embedding, the first five layers of the ResNet18

model, namely convolution, batch normalisation, ReLU activation, max pooling, and the first ResNet block are used. Then, an adaptive max pooling layer is employed to obtain a (12x12x64) tensor for each input image. Subsequently, the third axis of size 64 is averaged to obtain a (12x12x1) tensor per image. The element-wise *neg_exponential_tensor* function is applied to obtain cell weights of appropriate sign for the shortest path algorithms to be applied. Lastly, a squeeze function is applied to forget the last two dimensions. This methodology is used in an existing repository[1], which employs the ResNet18 architecture, as previously mentioned, for machine learning-based solutions to the problem. The repository provides the following code to create the Warcraft embedding:

```
function create_warcraft_embedding()
    resnet18 = ResNet(18, pretrain = false, nclasses = 1)
    model_embedding = Chain(resnet18.layers[1][1:4],
        AdaptiveMaxPool((12,12)),
        average_tensor,
        neg_exponential_tensor,
        squeeze_last_dims,
    )
    return model_embedding
end
```

Listing 1: ResNet18 architecture used in all the different models

The pure machine learning approach relies on the learned Warcraft embedding to estimate the cell costs, which can then be used to compute the shortest path. The main objective here is to train the model in such a way that the computed shortest path is as close as possible to the true shortest path.

The training process for the pure machine learning architecture involves several components. Firstly, the dataset is created by importing the Warcraft map data and splitting it into a training and testing set. The learning pipeline is then defined, which includes the Warcraft embedding as the encoder, a regularised generic function as the maximiser, and the mean squared error (MSE) loss function. The regularised generic function is composed of the true maximiser, a scaled half-square norm, and an identity function.

```
pipeline = (
    encoder=create_warcraft_embedding(),
    maximiser=RegularisedGeneric(true_maximiser, scaled_half_square_norm,
    identity),
    loss=Flux.Losses.mse,
)
```

Listing 2: Pipeline used for training an L2-Normalised Mean Squared Error model

The training function, $train\_function!$, is responsible for training the encoder model on the training dataset and evaluating it on the test dataset. This function minimises the $flux\_loss$, which is computed using the learning pipeline defined earlier. The training process involves iterating through the dataset for a specified number of epochs and updating the model parameters using the ADAM optimiser, which is an efficient and effective optimisation algorithm that adapts the learning rates for each parameter during training. Its adaptive nature and robust performance make it a popular choice for training deep learning models [16]. During training, the average train and test losses are stored, as well as the average cost ratios between computed and true shortest paths.

[1] https://github.com/LouisBouvier/WarcraftShortestPaths.jl/

```
function train_function!(;encoder, flux_loss, train_dataset, test_dataset,
    options::NamedTuple)
    ...
end
```

Listing 3: Training function used for training all the models

After training, the performance of the pure machine learning approach can be visualised by plotting the loss and gap between the computed and true shortest paths over time. This helps assess the quality of the learned embedding and its ability to approximate the true shortest paths. Finally, the trained model can be used to predict the cell costs for a test image and compute the corresponding shortest path.

All of this, as previously mentioned, comes from the WarcraftShortestPaths.jl library. These various functions are also used in our other two architectures, with any changes made being described in Sections 3.5 and 3.6.

3.5 Two-Stage Approaches to ML-CO Problems

In contrast to the pure machine learning architecture discussed in the previous subsection, a two-stage approach can be adopted to solve the ML-CO problems. The first stage involves learning a predictive model using machine learning techniques, while the second stage focuses on optimisation based on the predictions made by the model. One notable approach to address the two-stage problems is the Smart Predict-then-Optimise (SPO+) algorithm, introduced by Elmachtoub and Grigas [6].

3.5.1 Predict, then Optimise Framework

The "Predict, then Optimise" framework is at the core of many practical optimisation applications. This framework assumes that there is a nominal optimisation problem with a linear objective, in which the decision variable $w \in \mathbb{R}^d$ and feasible region $S \subseteq \mathbb{R}^d$ are well-defined and known. However, the cost vector $c \in \mathbb{R}^d$ is unavailable when the decision must be made, and an associated feature vector $x \in \mathbb{R}^p$ is available instead. The goal is to solve a contextual stochastic optimisation problem, as shown in Equation (1) in the original paper [6].

The predict-then-optimise framework relies on using a prediction for $\mathbb{E}_{c\sim D_x}[c|x]$, denoted by $\hat{c}$, and solving the deterministic version of the optimisation problem based on $\hat{c}$. This paper focuses on defining suitable loss functions for the framework, examining their properties, and developing algorithms for training prediction models using these loss functions.

3.5.2 Key Ingredients of the Framework

The framework comprises several key ingredients. First, the nominal (downstream) optimisation problem is of the form $P(c) : z^*(c) := \min_{w\in S} c^T w$. Second, the training data consists of pairs $(x_1, c_1), (x_2, c_2), \ldots, (x_n, c_n)$, where $x_i \in X$ is a feature vector representing contextual information associated with c_i. Third, there is a hypothesis class H of cost vector prediction models $f : X \to \mathbb{R}^d$, where $\hat{c} := f(x)$ is interpreted as the predicted cost vector associated with the feature vector x. Lastly, a loss function $l(\cdot, \cdot) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$ is used to quantify the error in making prediction $\hat{c}$ when the realised (true) cost vector is actually c [6].

According to the empirical risk minimisation (ERM) principle, a prediction model $f^* \in H$ should be determined by solving the optimisation problem in Equation (3) from the original paper [6].

3.5.3 Smart Predict-then-Optimise (SPO+) Algorithm

The SPO+ algorithm, introduced by Elmachtoub and Grigas [6], builds on the "Predict, then Optimise" framework by defining loss functions that consider the underlying structure of the optimisation problem, as opposed to standard loss functions, which are independent of the problem structure. By leveraging problem structure, the SPO+ algorithm aims to minimise decision errors when predicting cost vectors.

The SPO+ algorithm has three main components. First, it employs a convex surrogate loss function that measures the difference between the optimal decision and the decision made using the predicted cost vector. This loss function serves as an upper bound for the true loss function, which is typically non-convex. Second, a training algorithm is used to minimise the surrogate loss function in order to learn a predictive model for the cost vector, employing techniques such as stochastic gradient descent. Finally, an optimisation algorithm is applied that uses the learned predictive model to make decisions in the second stage, taking into account the specific structure of the optimisation problem [6].

By employing the SPO+ algorithm in the context of the Warcraft shortest path problem, we can leverage the problem structure to improve the quality of our predictions and the resulting decisions. In the first stage, we train a predictive model for the cost vector using a surrogate loss function tailored to the problem. In the second stage, we apply the learned model to new instances and solve the optimisation problem based on the predicted cost vector.

Using the SPO+ algorithm, we can expect to achieve better performance compared to a pure machine learning architecture, as it takes into account the specific structure of the optimisation problem. This can potentially lead to more accurate predictions and improved decision-making in finding the shortest path for the Warcraft maps dataset.

3.5.4 Integrating SPO+ Into WarcraftShortestPaths.jl

In order to integrate the SPO+ algorithm into the existing WarcraftShortestPaths.jl framework, we need to modify the learning pipeline and the loss function in the main.jl file. The pipeline should now include the encoder, maximiser, and the SPOPlusLoss instead of the FenchelYoungLoss. This change enables the pipeline to benefit from the SPO+ algorithm's advantages in solving ML-CO problems. SPOPlussLoss implements the previously mentioned SPO+ algorithm, and is already included in the InferOpt.jl framework[2]. The modified pipeline is as follows:

```
pipeline = (
    encoder=create_warcraft_embedding(),
    maximiser=identity,
    loss=SPOPlusLoss(true_maximiser),
)
```

Listing 4: Modified pipeline with SPO+ integration

[2]https://github.com/axelparmentier/InferOpt.jl/

The remainder of the main.jl code can be left unchanged, as the training function, performance evaluation, and visualisation functions are analogous to the specific loss function employed.

By integrating the SPO+ algorithm into the existing framework, we can take advantage of the two-stage approach in solving the Warcraft shortest path problem. The first stage trains the predictive model for the cost vector using the SPOPlusLoss function, which considers the underlying structure of the optimisation problem. In the second stage, the trained model is used to make predictions for new instances and solve the optimisation problem based on the predicted cost vector.

3.6 Combined ML-CO Pipeline

In this section, we discuss the development of a combined machine learning and combinatorial optimisation (ML-CO) pipeline to tackle blackbox optimisation problems. The primary goal is to exploit the strengths of both ML and CO techniques to efficiently learn and optimise complex combinatorial problems that are difficult to solve using traditional or combined methods, both of which were introduced in Sections 3.4 and 3.5.

3.6.1 Incorporating CO into ML Frameworks

To incorporate combinatorial optimisation (CO) methods into machine learning (ML) frameworks, we use the concept of probabilistic CO layers, which were introduced by Dalle et al. in their 2022 paper, "Learning with Combinatorial Optimisation Layers: a Probabilistic Approach" [3]. These layers make use of CO oracles to generate solutions to the combinatorial problems and provide the gradients required for training ML models using gradient-based optimisation methods like stochastic gradient descent.

The main idea behind probabilistic CO layers is to associate a probability distribution over the vertices of a polytope V with a parameter θ. By perturbing the objective direction θ and regularising the output, we can compute the gradients of these probability distributions with respect to the parameters θ. This allows us to update the parameters using gradient-based optimisation methods, thus enabling the training of ML models on blackbox combinatorial optimisation problems.

3.6.2 Training ML Models with Blackbox CO Problems

To train ML models on blackbox combinatorial optimisation problems, we follow the approach proposed in the paper by Dalle et al., which consists of three main steps [3]:

1. *Problem Representation:* First, we represent the combinatorial optimisation problem as an instance of a probabilistic CO layer. This representation includes specifying the problem's combinatorial structure, objective function, and constraints, as well as the CO oracle that generates feasible solutions.

2. *Incorporating Regularisation:* Next, we introduce regularisation into the problem to ensure that the gradients can be computed efficiently. Regularisation can be applied using various techniques, such as additive or multiplicative perturbations or through explicit regularisation functions. The choice of regularisation method depends on the problem's structure and the desired level of smoothness.

3. *Gradient-Based Optimisation:* Finally, we train the ML model using gradient-based optimisation methods, such as stochastic gradient descent or ADAM. The gradients of the probabilistic CO layer are computed using the perturbation and regularisation techniques introduced in the previous step, allowing us to update the model's parameters efficiently.

By following these steps, we can train ML models on blackbox combinatorial optimisation problems, effectively combining the strengths of both ML and CO techniques to efficiently learn and optimise complex problems that are difficult to solve using traditional methods alone. These steps can also be seen in the source code for WarcraftShortestPath.jl, which was previously discussed in Sections 3.4 and 3.5.

We need to adapt the models to effectively use ML models in the context of combinatorial optimisation problems to generate useful information for solving the CO instances. One common approach is to use ML models to predict the values of problem parameters θ that guide the search for optimal solutions in the CO problem. By learning these parameters, the ML model can provide valuable insights and guidance to the CO solver, thus improving its performance and enabling the efficient exploration of the search space. By learning these parameters, the ML model can provide valuable insights and guidance to the CO solver, thus improving its performance and enabling the efficient exploration of the search space.

3.6.3 Learning by Imitation

Learning by imitation is an approach where additional information is utilised to guide the training process. In this method, for each input sample $x^{(i)}$, a target $t^{(i)}$ is provided, enabling the loss to consider the target as an additional argument. This section in Dalle et al.'s paper discusses imitation losses that are appropriate for hybrid ML-CO pipelines and explains how to compute their gradients [3].

There are two primary types of targets: a high-quality solution $\bar{t} = \bar{y}$, and the true objective direction $\bar{\theta}$, from which $\bar{y} = f(\bar{\theta})$ and $t = (\bar{\theta}, \bar{y})$ can be deduced. It is crucial not to focus solely on replicating the targets while learning by imitation, as this can be misleading. The reason for this is the different consequences that overestimating or underestimating θ may have on the quality of the downstream solution. Therefore, a loss function that accounts for the optimisation step is required [3].

The imitation learning loss functions discussed in the original paper [3] are built upon the same components, making it possible for the optimisation problem to play a role in the loss. Minimising the gap between the regularised CO problem encourages the output of a solution y that is close to the target $\bar{y}$. This loss function is convex with respect to θ, and a subgradient can be obtained using Danskin's theorem.

Several prominent loss functions from the literature are special cases of the loss function presented in the main paper, including Structured Support Vector Machines (S-SVM), Smart "Predict, then Optimise" (SPO+), and Fenchel-Young losses. Each of these losses has specific properties and subgradient calculations, making them suitable for different scenarios and applications. The generic loss function presented in the original paper [3] provides a foundation for further exploration and testing in future work, especially through the implementation of the mathematical concepts into the InferOpt.jl library.

3.6.4 Integrating the Different Imitation Losses

In order to adapt the WarcraftShortestPaths.jl code to suit the different maximisers and regularisers, I made several changes to the model settings in the src/main.jl file. These changes included modifying the encoder, maximiser, and loss functions, as well as adjusting the hyperparameters, such as the number of samples M and the learning rate lr_start.

Below are the four different model configurations that I experimented with, which are also adopted from the paths.jl file from the InferOpt.jl library[3]:

```
pipeline = (
encoder=create_warcraft_embedding(),
maximiser= identity,
loss=FenchelYoungLoss(PerturbedMultiplicative(true_maximiser; varepsilon=
    options.epsilon, nb_samples=options.M)),
)

```

Listing 5: Fenchel Young Loss with Perturbed Multiplicative Noise

```
pipeline = (
encoder=create_warcraft_embedding(),
maximiser= identity,
loss=FenchelYoungLoss(PerturbedAdditive(true_maximiser; varepsilon=options.
    epsilon, nb_samples=options.M)),
)

```

Listing 6: Fenchel Young Loss with Perturbed Additive Noise

```
pipeline = (
encoder=create_warcraft_embedding(),
maximiser= PerturbedMultiplicative(true_maximiser; varepsilon=options.
    epsilon, nb_samples=options.M),
loss=Flux.Losses.mse,
)

```

Listing 7: Mean Squared Error Loss with Perturbed Multiplicate Noise

```
pipeline = (
encoder=create_warcraft_embedding(),
maximiser= PerturbedAdditive(true_maximiser; varepsilon=options.epsilon,
    nb_samples=options.M),
loss=Flux.Losses.mse,
)

```

Listing 8: Mean Squared Error Loss with Perturbed Additive Noise

Through the combination of ML and CO techniques, the proposed ML-CO pipeline can effectively learn and optimise complex combinatorial problems that are difficult to solve using traditional methods alone. By adapting ML models to predict problem parameters and leveraging these predictions in the CO search process, we can efficiently explore the search space and find optimal or near-optimal solutions to challenging CO instances.

[3] https://github.com/axelparmentier/InferOpt.jl/blob/main/test/paths.jl

4 Evaluation

4.1 Environment

I conducted the experiments, the results of which can be found in Section 4.2, in this research on a Google Cloud virtual machine (VM) to ensure a consistent and controlled environment. The machine configuration was set to an e2-standard-8 type, which provides a well-balanced combination of performance and cost efficiency. The vCPUs to core ratio was set at 1 vCPU per core, which allows for efficient parallel processing.

To facilitate remote access and monitoring, a display device was enabled on the VM, allowing the use of screen capturing and recording tools. This feature proved helpful in visualising the progress of experiments and understanding the intermediate results. The virtual machine did not include any dedicated GPUs, as my research did not require the substantial processing capabilities typically provided by GPUs.

The operating system installed on the VM was a headless Ubuntu, specifically the machine learning package provided by Google Cloud. This package includes a variety of pre-installed tools and libraries, making it an ideal choice for conducting machine learning experiments. Furthermore, the VM was equipped with 16 GB of RAM, which provided ample memory capacity to handle the computational requirements of the experiments.

To run the WarcraftShortestPaths.jl, I installed Julia and the necessary libraries, including the main focus of my paper, InferOpt.jl. As the Ubuntu machine was headless, some changes had to be made to Julia in order for it to download any images. I modified the .julia/etc/startup.jl file by adding the line ENV[“GKSwstype”] = “100”.

This environment offered a stable and powerful platform to conduct my experiment, enabling efficient exploration and optimisation of the ML-CO pipeline. The use of Google Cloud VM ensured a reliable and controlled setting, eliminating potential issues related to hardware or software inconsistencies that could arise in other environments.

4.2 Results & Analysis

To determine the optimal hyperparameters for my models, I conducted experiments with two different epoch sizes, 100 and 20, and two different learning rates, 0.1 and 0.001. I tested these four combinations with each of the architectures shown in Table 1.

In the course of my experiments, I encountered a ”Negation Assertion” error when attempting to apply the SPO+ loss function with Dijkstra’s algorithm. Consequently, this combination was not included in the analysis. Furthermore, given the extensive range of results obtained from the experiments, I will specifically focus on discussing the outcomes with the Bellman-Ford algorithm, using 100 epochs and learning rates of 0.1 and 0.001. This decision is based on the fact that the Bellman-Ford algorithm demonstrated compatibility with all architectures, including those employing the SPO+ loss function, thus providing a comprehensive representation of the model’s performance.

The analysis of the optimality gap for models employing the Fenchel Young loss and the SPO+ loss function reveals that they achieved the best performance, as seen in Figure 6, exhibiting roughly similar results on both the training and test datasets. Furthermore, the optimality gap results indicate that the pure machine learning architecture with non-regularised mean squared error and the additive perturbation produced comparable outcomes in both the training and test sets, which were among the least favorable results. This observation supports the claims made in the original paper [3], which reported that non-satisfactory outcomes were obtained for 80 training samples.

Table 1: The different architectures used

Shortest Path Algorithm	Probabilistic CO Layer	Loss
Dijkstra's Algorithm	Non-regularised	Mean square error
	Multiplicative perturbation	Mean square error
	Additive perturbation	Mean square error
	Multiplicative perturbation	Fenchel Young
	Additive perturbation	Fenchel Young
Bellman-Ford Algorithm	Non-regularised	Mean square error
	Non-regularised	SPO+
	Multiplicative perturbation	Mean square error
	Additive perturbation	Mean square error
	Multiplicative perturbation	Fenchel Young
	Additive perturbation	Fenchel Young

It is noteworthy that the multiplicative perturbation regularised model outperformed the pure machine learning architecture and the additive perturbation model in the training set, as seen in the left figure within Figure 6; however, this superior performance was not observed in the test set, as seen in the right figure within Figure 6. This discrepancy may suggest that the multiplicative perturbation regularised, MSE loss model is more prone to overfitting the training data, leading to reduced generalisation when applied to the test set.

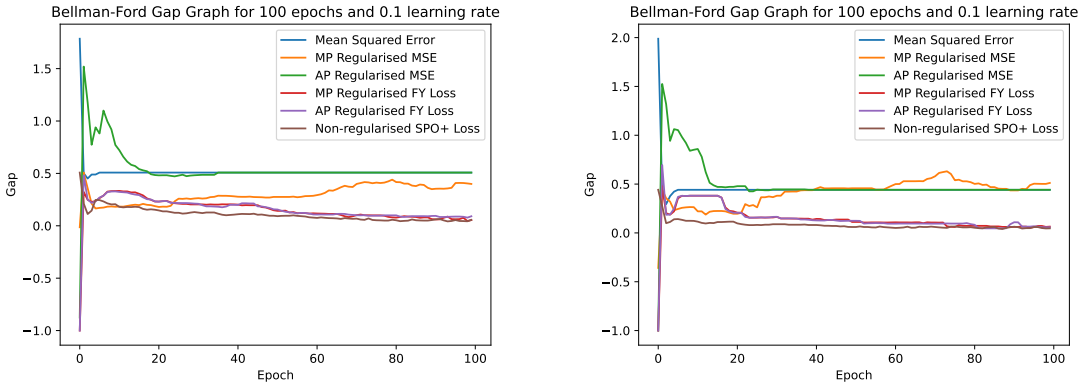

Figure 6: Gap training and test results using the Bellman-Ford algorithm for all the different architectures from Table 1. Here, the learning rate is 0.1 and the number of epochs is 100.

The loss graphs for both the training and test sets exhibit peculiar patterns, as seen in Figure 27, particularly for the SPO+, namely the "predict then optimise", architecture. In this case, the loss experiences a significant spike before rapidly dropping down to zero. This behavior might be attributed to the SPO+ optimisation process, which may initially cause an increased divergence from the optimal solution before converging as the

optimisation progresses. On the other hand, the remaining architectures demonstrate a different pattern, where the loss reaches almost zero after just a few epochs. This could be a result of the specific learning dynamics of these models, where they quickly adapt to the training data and manage to minimise the loss early on during the training process. However, it is essential to consider the possibility of overfitting in such cases, as rapid convergence could be indicative of the model fitting too closely to the training data, potentially compromising its generalisation capabilities. This could be the case in the multiplicative perturbation regularised MSE loss model, as previously mentioned.

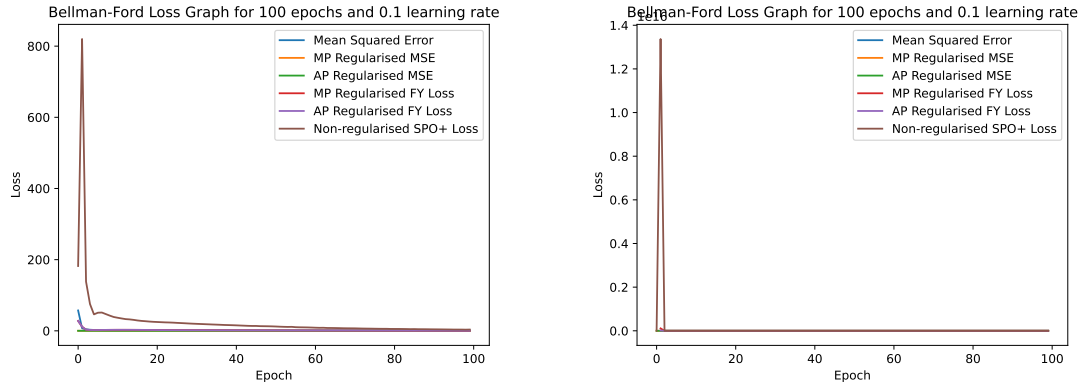

Figure 7: Loss training and test results using the Bellman-Ford algorithm for all the different architectures from Table 1. Here, the learning rate is 0.1 and the number of epochs is 100.

The observed differences in performance between the various architectures, particularly when examining the gap and loss graphs, shows that several conclusions can be drawn. Firstly, the architectures that employed the Fenchel Young loss and the SPO+ loss function consistently demonstrated superior performance, showcasing their effectiveness in handling the optimisation problem at hand. This finding reinforces the importance of selecting appropriate loss functions tailored to the specific problem domain, as it can significantly impact the model's overall performance and generalisation capabilities.

Secondly, the peculiar patterns observed in the loss graphs, such as the initial spike in the SPO+ architecture and the rapid convergence to near-zero in other architectures, underscore the complexity of the learning dynamics in these models. These observations highlight the importance of thoroughly investigating and understanding the interplay between various factors, such as model architecture, hyperparameters, initialisation, and learning rate scheduling, which can influence the training process and loss behaviour. A deeper understanding of these aspects can help identify potential issues, such as overfitting or suboptimal learning dynamics, and guide the development of more robust and generalisable models.

The results observed here align with those reported by Dalle et al. [3] in several key aspects. In both cases, the SPO+ approach demonstrated impressive performance, achieving near-zero average optimality gaps in both the train and test sets. This can be attributed to the fact that SPO+ leverages the true cell costs and the problem structure within the loss during training, leading to effective learning even with a small dataset.

Additionally, the results achieved here further corroborate the findings of Dalle et al., wherein architectures employing Fenchel-Young losses outperformed those using MSE losses [3]. The superior performance of the Fenchel-Young losses can be explained by their

incorporation of the optimisation problem into the loss definition, while MSE losses only seek to imitate the target paths without explicitly considering the solution cost.

Furthermore, the results I obtained support the notion that learning by experience with the small sub-dataset is indeed possible. The combined architecture, which incorporates perturbation techniques and Fenchel-Young losses, yields better performance in terms of optimality gaps compared to learning by imitation using an MSE loss. This demonstrates the potential of leveraging learning by experience in combination with CNNs for combinatorial optimisation problems [3].

5 Conclusion

5.1 Summary

In this thesis, we have looked at the performance of various architectural approaches to solving complex combinatorial optimisation problems that require the use of machine learning, with a focus on the optimality gap as the primary metric. Three key approaches were explored: the two-step architecture (SPO+), the combined architecture (using Fenchel Young loss and perturbed noise), and the pure machine learning approach (mean squared error loss).

The analysis demonstrated that both the SPO+ and combined architectures outperformed the pure machine learning approach in terms of the optimality gap, indicating their effectiveness in handling combinatorial optimisation problems, and supporting conclusions made by Dalle et al. [3], as well as others such as Wilder et al. [24]. While it is difficult to definitively declare either the two-step approach or the combined architecture as superior specifically for the limited dataset used, the choice between them may ultimately depend on the specific problem, dataset, and computational requirements.

The evaluation of these approaches using the Warcraft Shortest Paths dataset provided valuable insights into their advantages and disadvantages, as well as the role of metrics such as the loss and the gap in determining their effectiveness. By examining all three approaches applied to a single dataset, this thesis contributes to the standardisation of methods for solving blackbox combinatorial problems and offers a deeper understanding of the fundamental concepts of machine learning and combinatorial optimisation.

In summary, this thesis has highlighted the importance of considering the optimality gap as the main metric when evaluating architectural approaches to solving combinatorial optimisation problems. The SPO+ and combined architectures have demonstrated their ability to effectively handle these problems, with their relative advantages depending on the specific problem, dataset, and computational requirements. By thoroughly investigating these approaches and providing insights into their strengths and weaknesses, this work contributes to the ongoing efforts to develop efficient and effective solutions to the complex combinatorial optimisation problems faced in the modern world.

5.2 Future Work

In terms of future work, there are several areas of research that can be explored to enhance the understanding and performance of machine learning approaches in solving combinatorial optimisation problems. One area of interest is to experiment with various combinatorial optimisation datasets to test the viability of combined architectures versus two-step architectures. By comparing the performance of these architectures on different datasets, we

can gain a better understanding of their strengths and limitations, and determine their suitability for various optimisation problems.

Additionally, investigating alternative loss functions is another direction for future work. While this thesis focused on the mean squared error loss, Fenchel Young loss, and SPO+, there may be other loss functions that better capture the combinatorial structure of the problem. Investigating and comparing alternative loss functions can provide a more comprehensive understanding of their impact on the performance of the architectural approaches.

Adapting and evaluating the architectures for other combinatorial optimisation problems, such as the travelling salesman problem or vehicle routing problem, can help generalise the findings and contribute to a broader understanding of their applicability. This would involve modifying the existing architectures and testing their performance on new problem instances, which would offer insights into the versatility of these approaches in addressing various optimisation challenges.

Moreover, future research can focus on developing hybrid architectures that combine the strengths of different approaches discussed in this thesis. By incorporating elements from the two-step architecture (SPO+), the combined architecture (perturbation and Fenchel Young loss), and the pure machine learning approach (mean squared error loss), it may be possible to create more powerful and versatile models capable of handling a wider range of combinatorial optimisation problems. This could involve integrating SPO+ with perturbation techniques or exploring ensemble methods that combine the predictions of different architectures.

Lastly, investigating the interpretability and explainability of the architectures is crucial. As machine learning models become increasingly complex and powerful, the need for interpretability and explainability becomes more important. Assessing the interpretability of the proposed architectures and developing techniques to enhance their explainability can help to build trust and provide valuable insights into their inner workings, which can ultimately lead to improved decision-making in combinatorial optimisation problems.

By pursuing these avenues of research, we can further advance the understanding and performance of machine learning approaches in solving combinatorial optimisation problems that require the use of machine learning. These efforts can contribute to the development of more efficient, effective, and interpretable models, ultimately helping to address some of the most challenging optimisation problems faced in the modern world.

References

[1] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[2] Christopher Bishop. In *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] Guillaume Dalle, Léo Baty, Louis Bouvier, and Axel Parmentier. Learning with Combinatorial Optimization Layers: a Probabilistic Approach. `https://doi.org/10.48550/arXiv.2207.13513`. 2022.

[4] George B Dantzig. The simplex method for linear programming. *Management Science*, 2(3):197–201, 1947.

[5] Edsger W Dijkstra. *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematik, 1959.

[6] Adam N. Elmachtoub and Paul Grigas. Smart "Predict, then Optimize". `https://arxiv.org/abs/1710.08005`. 2020.

[7] L. R. Ford Jr and D. R. Fulkerson. A method for the solution of the shortest-route problem. *Operations Research*, 6(2):225–229, 1958.

[8] Antoine François, Quentin Cappart, and Louis-Martin Rousseau. How to Evaluate Machine Learning Approaches for Combinatorial Optimization: Application to the Travelling Salesman Problem. `https://arxiv.org/abs/1909.13121`. 2019.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. `http://proceedings.mlr.press/v9/glorot10a.html`. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. In *Deep Learning*. *http://www.deeplearningbook.org/*. MIT Press, 2016.

[11] Simon Haykin. In *Neural Networks and Learning Machines*. Prentice Hall, 2009.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. `https://arxiv.org/abs/1512.03385`. 2015.

[13] C. Hedrick. *Routing Information Protocol*. *https://doi.org/10.17487/RFC1058*. 1988.

[14] Eric Horvitz. From Data to Predictions and Decisions: Enabling Evidence-Based Healthcare. Computing Community Consortium 6, Microsoft Research, 2010.

[15] Eric Horvitz and Tom Mitchell. From Data to Knowledge to Action: A Global Enabler for the 21st Century. Computing Community Consortium 11, Microsoft Research, 2010.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. `https://arxiv.org/abs/1412.6980`. 2020.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. In *Proceedings of the 86th Institute of Electrical and Electronics Engineers. https://ieeexplore.ieee.org/document/726791*, 1998.

[18] André F. T. Martins and Ramón Fernandez Astudillo. From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification. https://arxiv.org/abs/1602.02068. 2016.

[19] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 1958.

[20] David Rumelhart, Geoffrey Hinton, and Robert Williams. Learning Representations by Back-propagating Errors. https://doi.org/10.1038/323533a0. *Nature*, 1986.

[21] Andreas Veit, Michael J. Wilber, Serge J. Belongie, and Ross B. Girshick. Residual Networks Behave Like Ensemble of Relatively Shallow Networks. https://arxiv.org/abs/1605.06431. 2016.

[22] Marin Vlastelica. Differentiation of Blackbox Combinatorial Solvers Dataset. https://doi.org/10.17617/3.YJCQ5S. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, 2021.

[23] Marin Vlastelica, Anselm Paulus, Vít Musil, Georg Martius, and Michal Rolínek. Differentiation of Blackbox Combinatorial Solvers Optimization. https://doi.org/10.48550/arXiv.1912.02175. In *Proceedings of the 2020 International Conference on Learning Representation (ICLR 2020)*, 2019.

[24] Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the Data-Decisions Pipeline: Decision-Focused Learning for Combinatorial Optimization. https://arxiv.org/abs/1809.05504v2. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, 2018.

[25] Thomas Wood. Convolutional Neural Network. https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network.

[26] Aston Zhang, Zack C. Lipton, Mu Li, and Alex J. Smola. Residual Networks and ResNeXt. https://d2l.ai/chapter_convolutional-modern/resnet.html.

Appendices

A Epoch Size and Learning Rate Effects on Models

A.1 Non-regularised Mean Squared Error Model

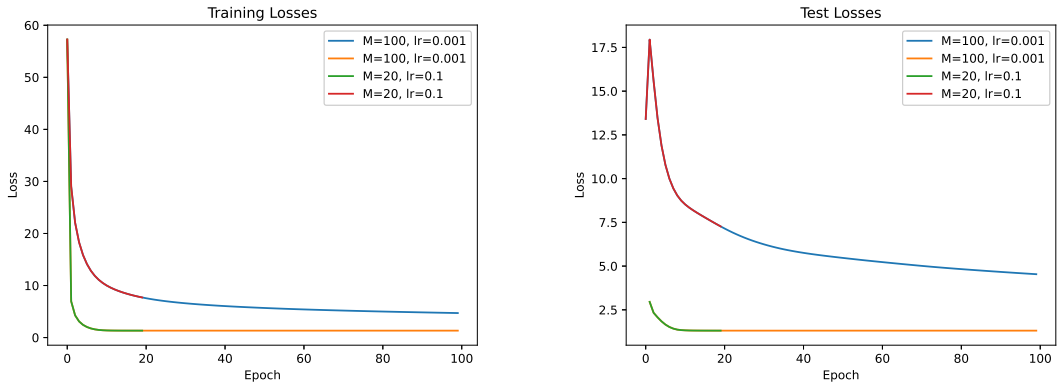

Figure 8: Training and test losses for the pure machine learning architecture using Mean Squared Error Loss

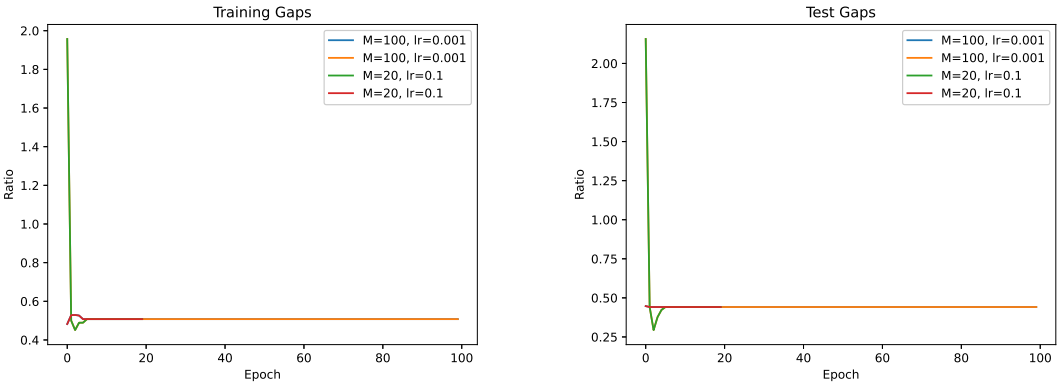

Figure 9: Training and test gaps for the pure machine learning architecture using Mean Squared Error Loss

A.2 Non-regularised SPO+ Model

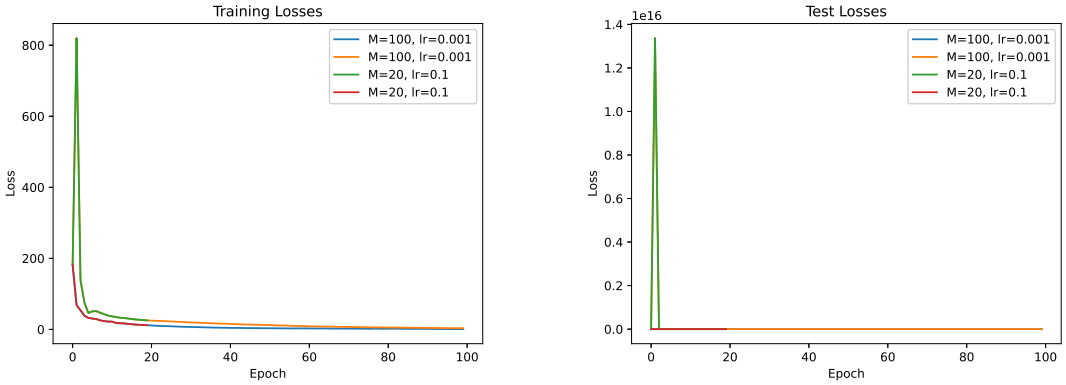

Figure 10: Training and test losses for the Non-regularised SPO+ architecture

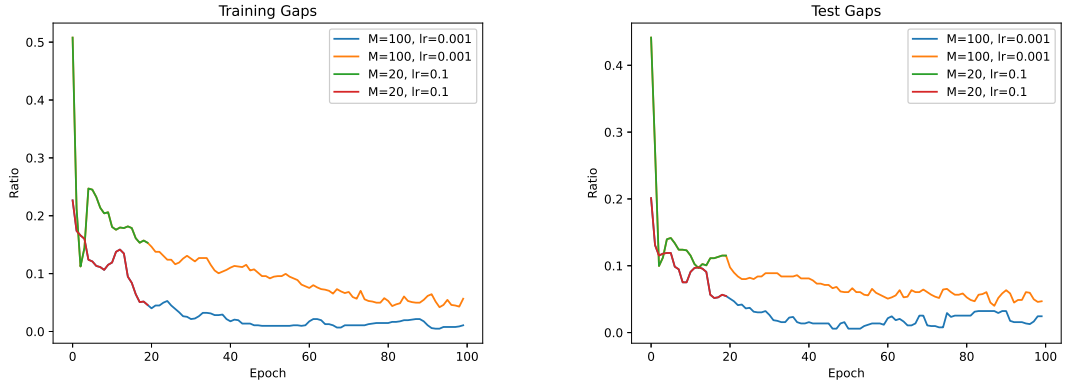

Figure 11: Training and test gaps for the Non-regularised SPO+ architecture

A.3 Additive-Perturbation-Regularised Mean Squared Error Loss Models

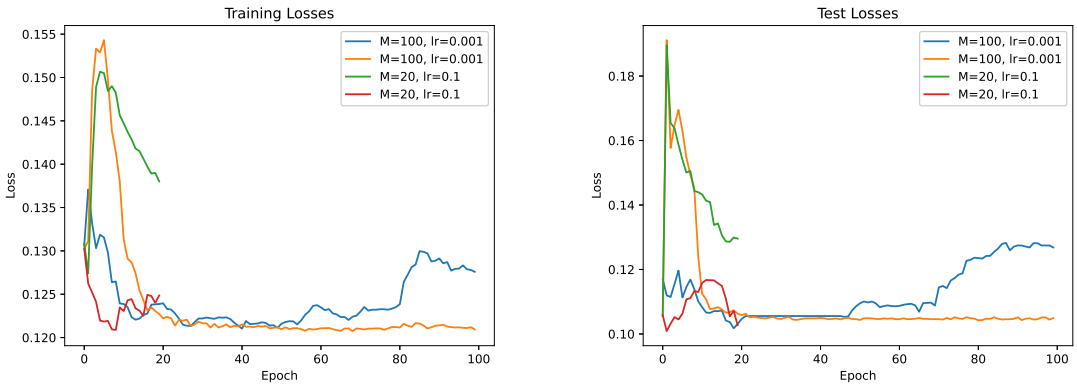

Figure 12: Training and test losses for the Additive-Perturbation-Regularised Mean Squared Error Loss using Dijkstra's Algorithm

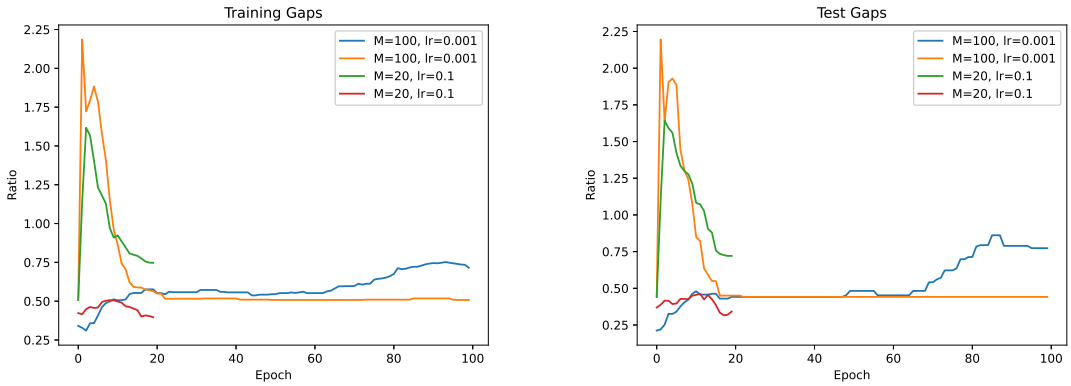

Figure 13: Training and test gaps for each of the Additive-Perturbation-Regularised Mean Squared Error Loss using Dijkstra's Algorithm

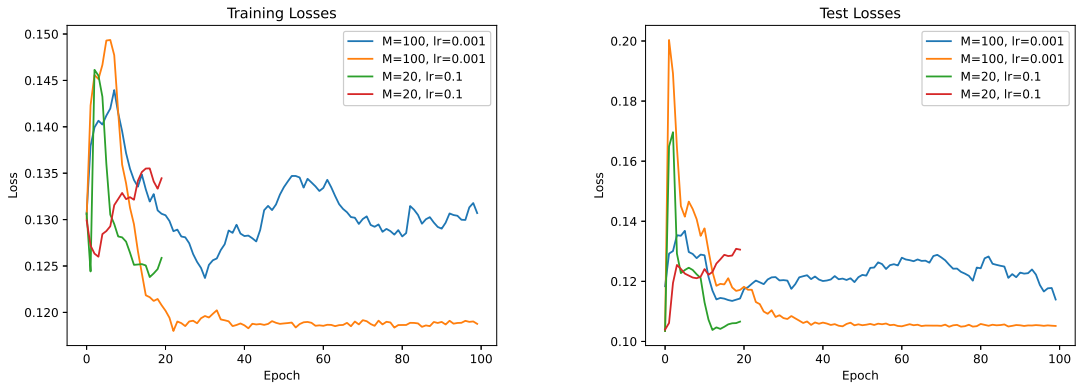

Figure 14: Training and test losses for each of the Additive-Perturbation-Regularised Mean Squared Error Loss using the Bellman-Ford Algorithm

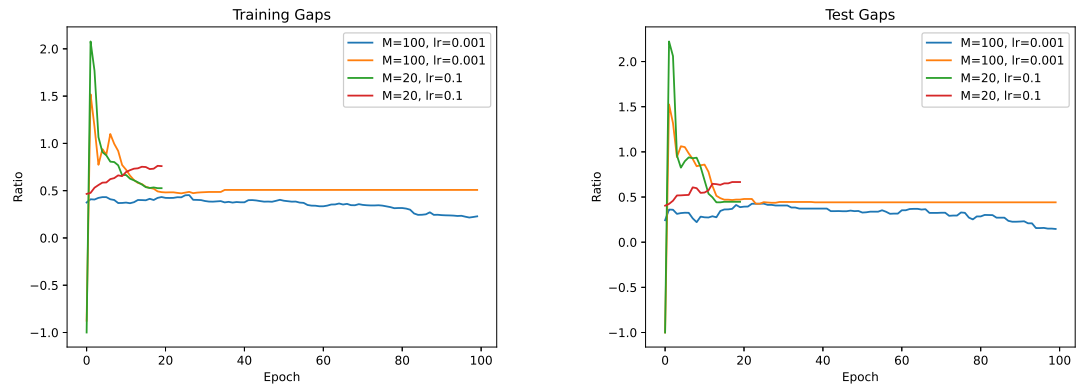

Figure 15: Training and test gaps for each of the Additive-Perturbation-Regularised Mean Squared Error Loss using the Bellman-Ford Algorithm

A.4 Multiplicative-Perturbation-Regularised Mean Squared Error Loss Models

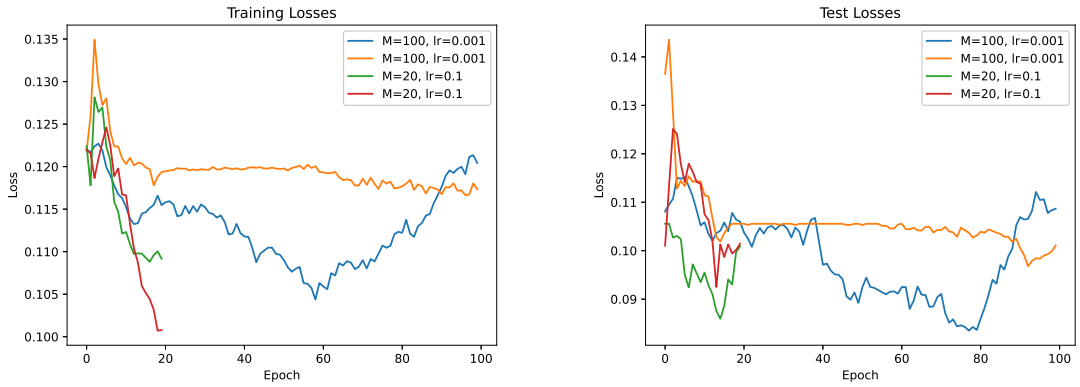

Figure 16: Training and test losses for the Multiplicative-Perturbation-Regularised Mean Squared Error Loss using Dijkstra's Algorithm

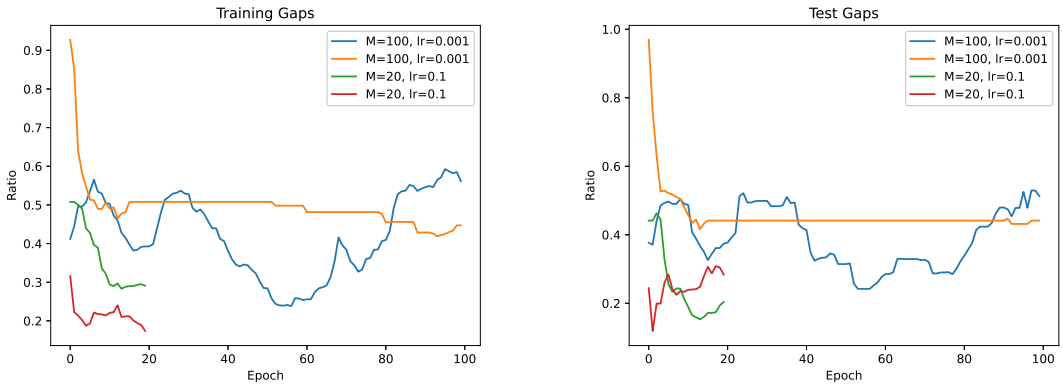

Figure 17: Training and test gaps for the Multiplicative-Perturbation-Regularised Mean Squared Error Loss using Dijkstra's Algorithm

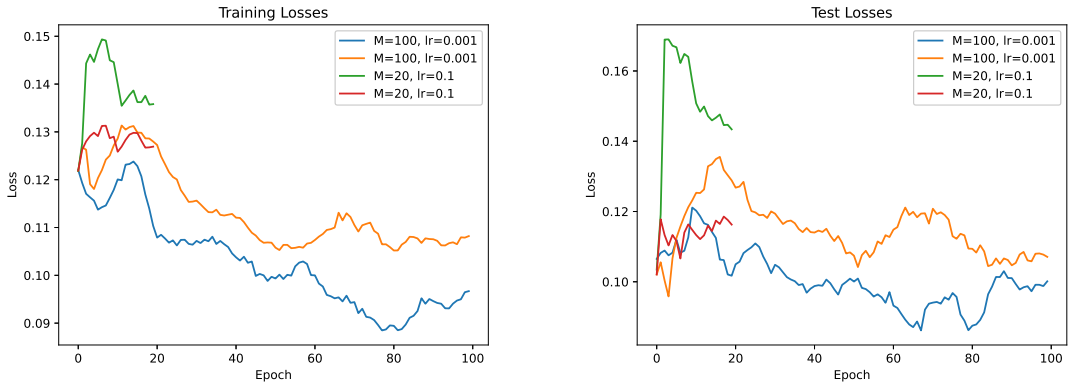

Figure 18: Training and test losses for the Multiplicative-Perturbation-Regularised Mean Squared Error Loss using the Bellman-Ford Algorithm

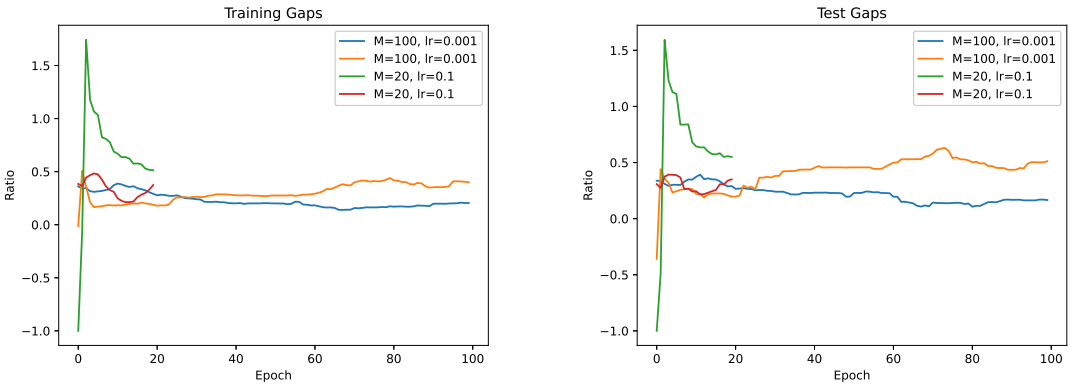

Figure 19: Training and test gaps for the Multiplicative-Perturbation-Regularised Mean Squared Error Loss using the Bellman-Ford Algorithm

A.5 Multiplicative-Perturbation-Regularised Fenchel Young Loss Models

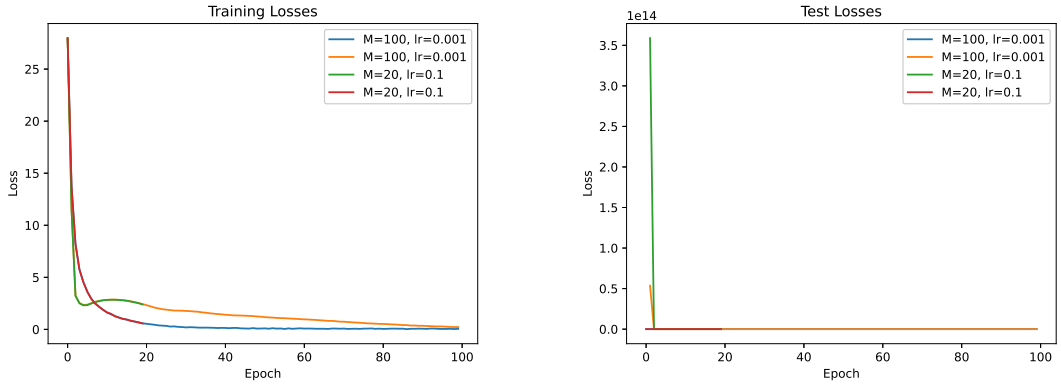

Figure 20: Training and test losses for the Multiplicative-Perturbation-Regularised Fenchel Young Loss using Dijkstra's Algorithm

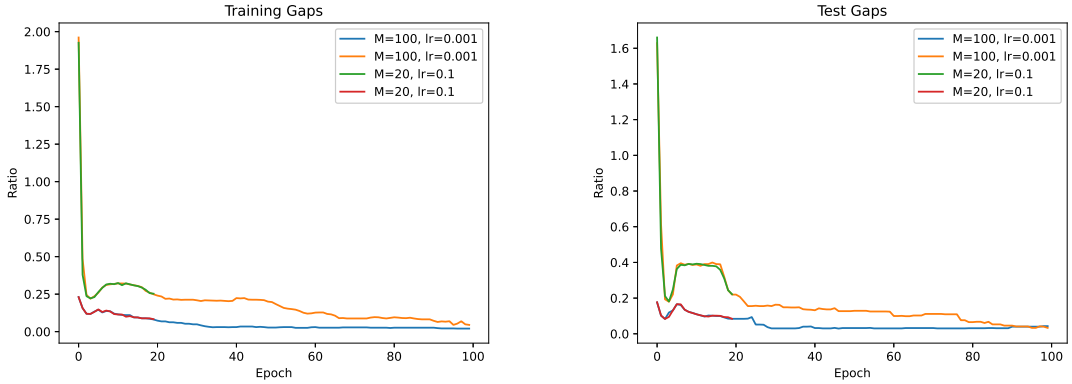

Figure 21: Training and test gaps for the Multiplicative-Perturbation-Regularised Fenchel Young Loss using Dijkstra's Algorithm

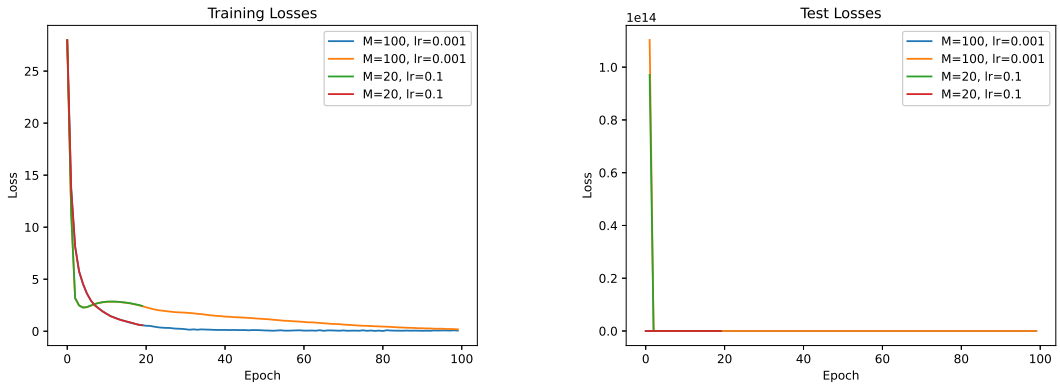

Figure 22: Training and test losses for the Multiplicative-Perturbation-Regularised Fenchel Young Loss using the Bellman-Ford Algorithm

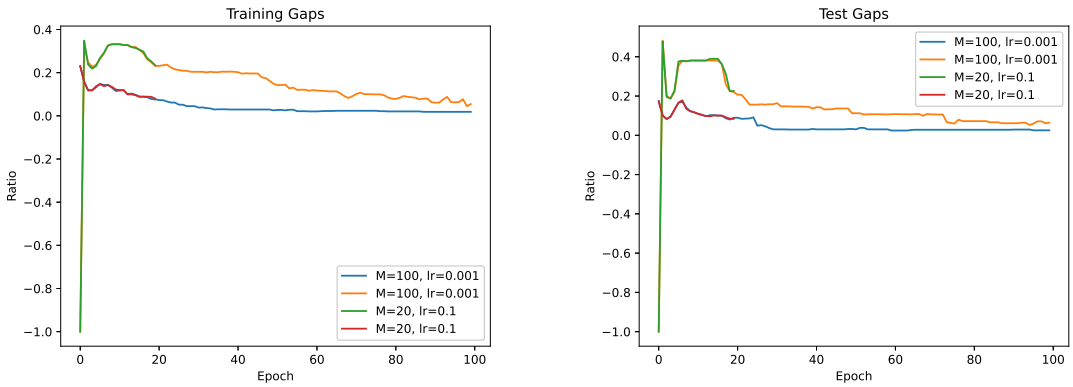

Figure 23: Training and test gaps for the Multiplicative-Perturbation-Regularised Fenchel Young Loss using the Bellman-Ford Algorithm

A.6 Additive-Perturbation-Regularised Fenchel Young Loss Models

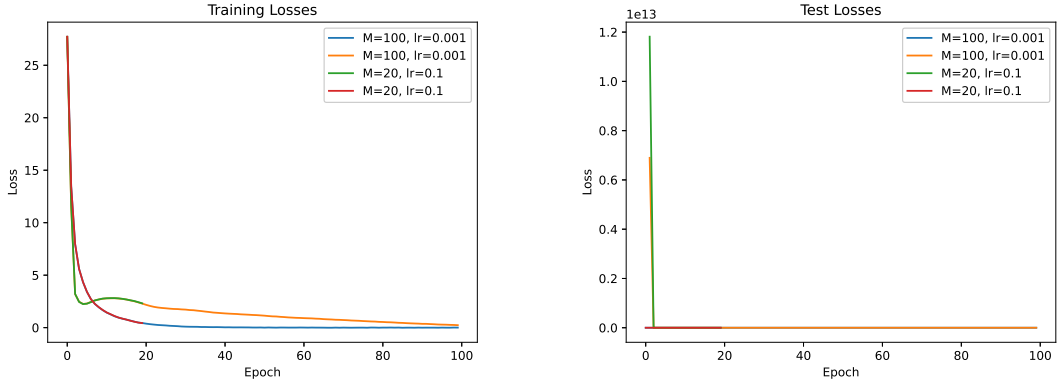

Figure 24: Training and test losses for the Additive-Perturbation-Regularised Fenchel Young Loss using Dijkstra's Algorithm

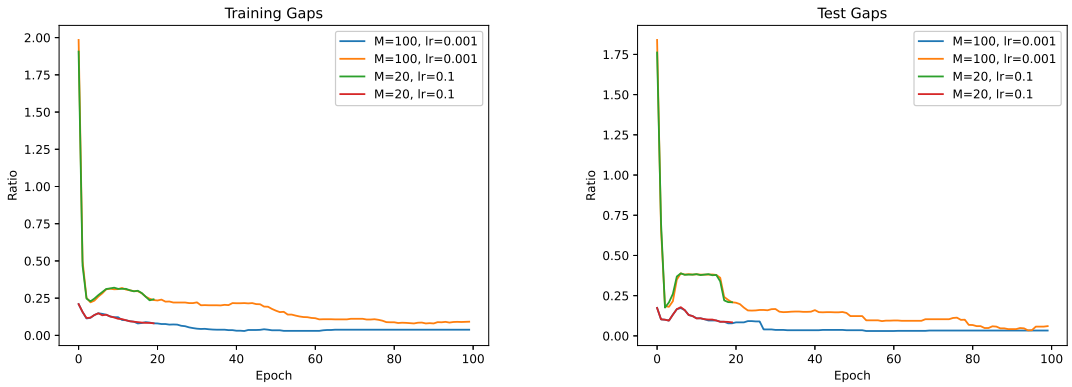

Figure 25: Training and test gaps for the Additive-Perturbation-Regularised Fenchel Young Loss using Dijkstra's Algorithm

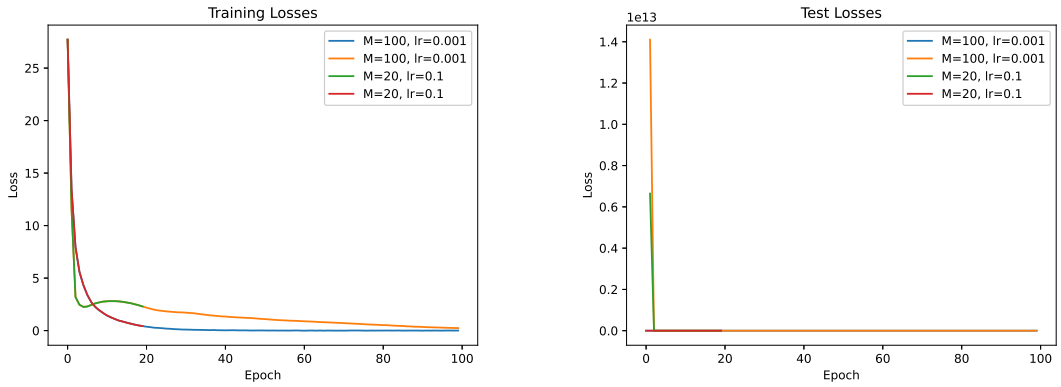

Figure 26: Training and test losses for the Additive-Perturbation-Regularised Fenchel Young Loss using the Bellman-Ford Algorithm

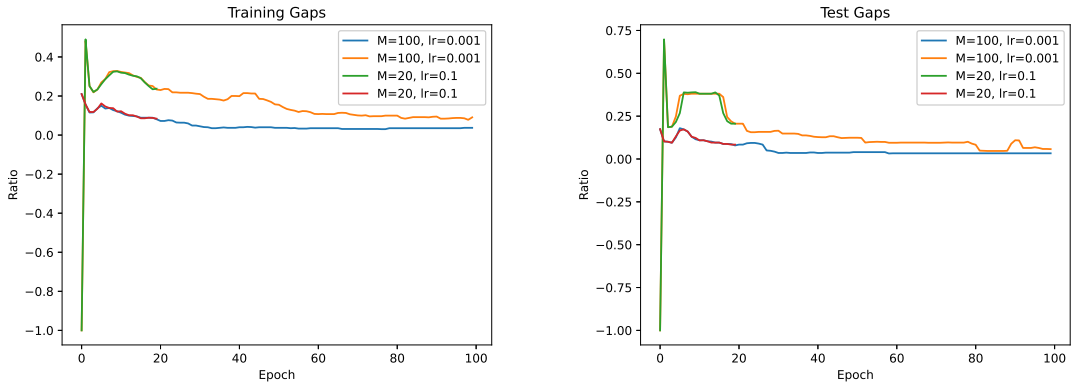

Figure 27: Training and test gaps for Additive-Perturbation-Regularised Fenchel Young Loss using the Bellman-Ford Algorithm

B Comparing the Different Architectures

B.1 Using Dijkstra's Algorithm

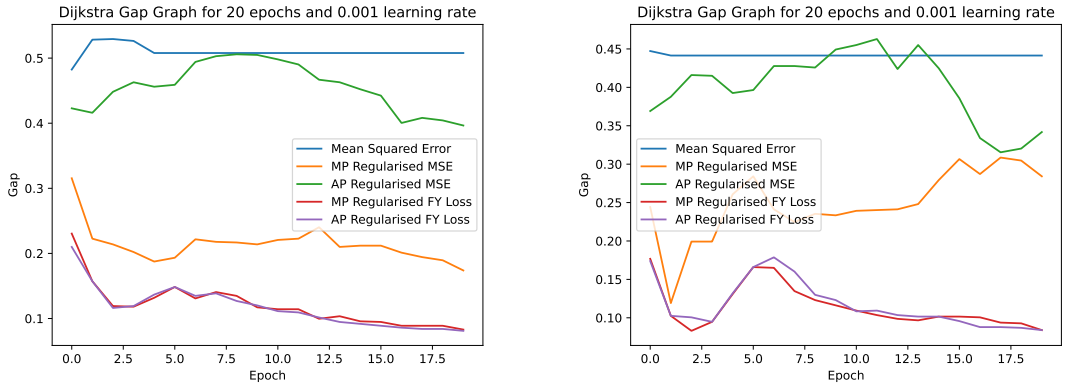

Figure 28: Training and test gaps for all architectures with an epoch size of 20 and a learning rate of 0.001

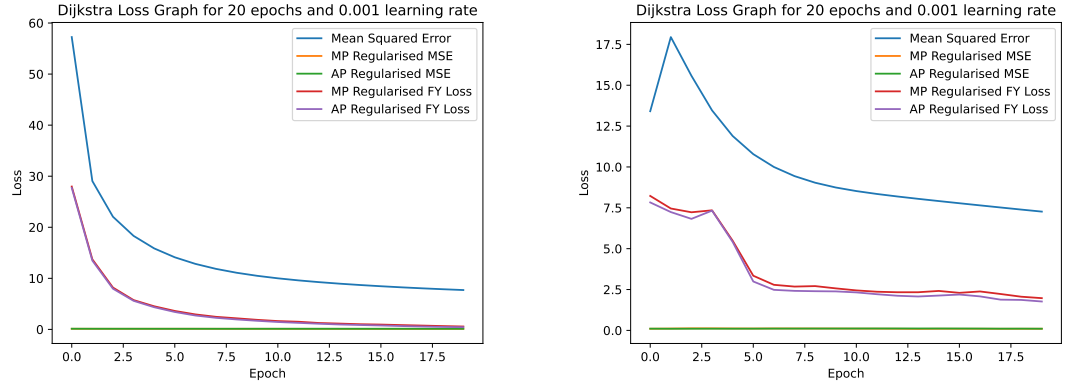

Figure 29: Training and test losses for all architectures with an epoch size of 20 and a learning rate of 0.001

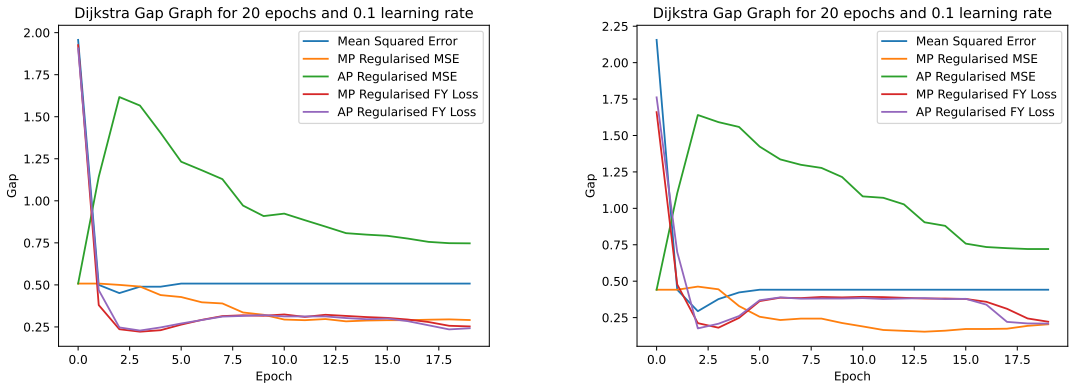

Figure 30: Training and test gaps for all architectures with an epoch size of 20 and a learning rate of 0.1

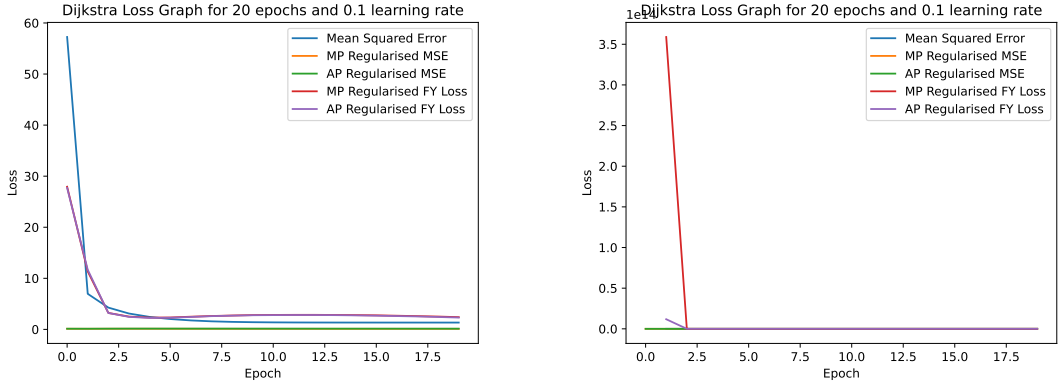

Figure 31: Training and test losses for all architectures with an epoch size of 20 and a learning rate of 0.1

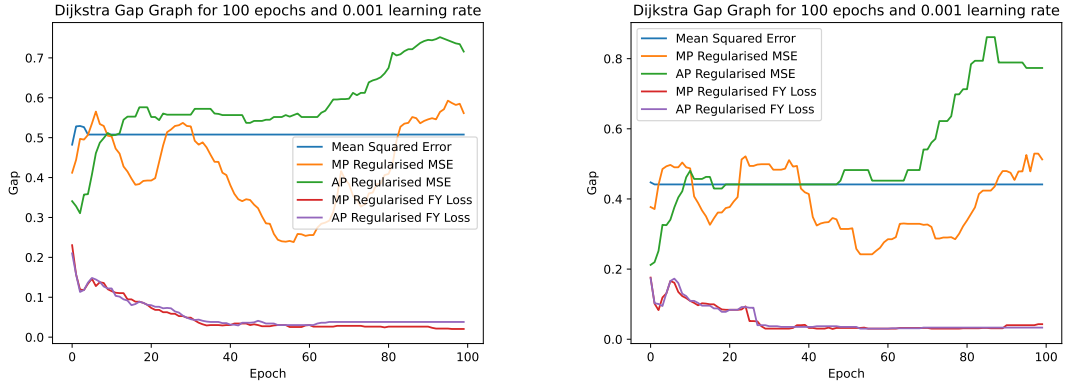

Figure 32: Training and test gaps for all architectures with an epoch size of 100 and a learning rate of 0.001

B.2 Using The Bellman-Ford Algorithm

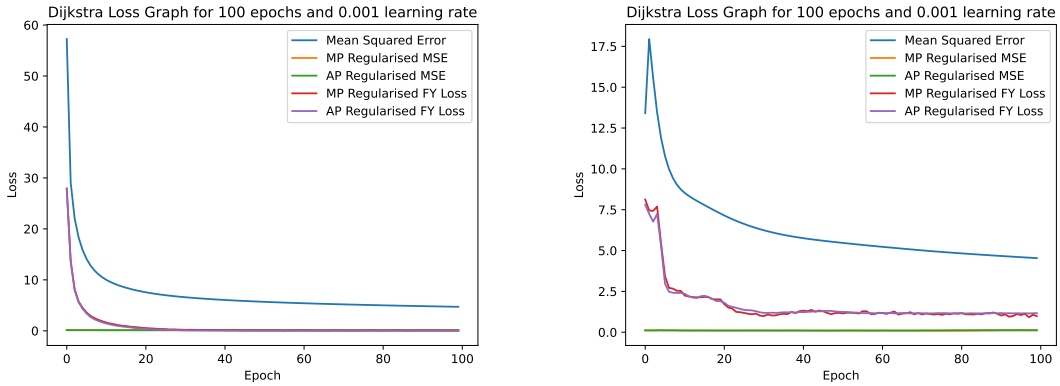

Figure 33: Training and test losses for all architectures with an epoch size of 100 and a learning rate of 0.001

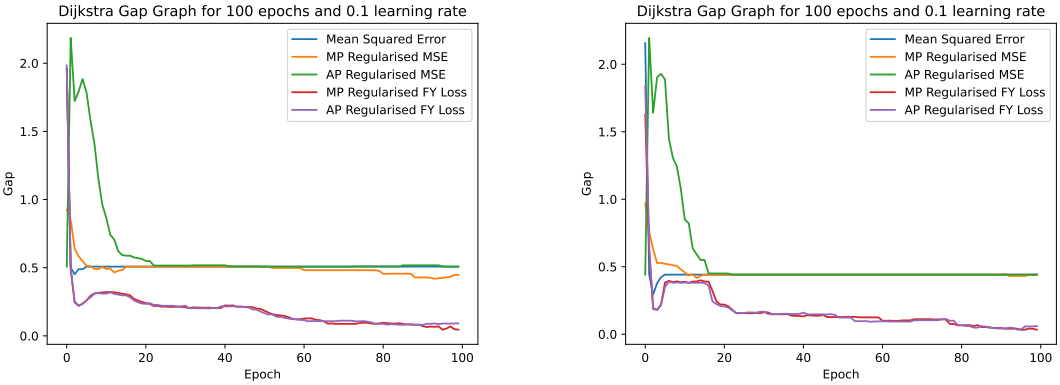

Figure 34: Training and test gaps for all architectures with an epoch size of 100 and a learning rate of 0.1

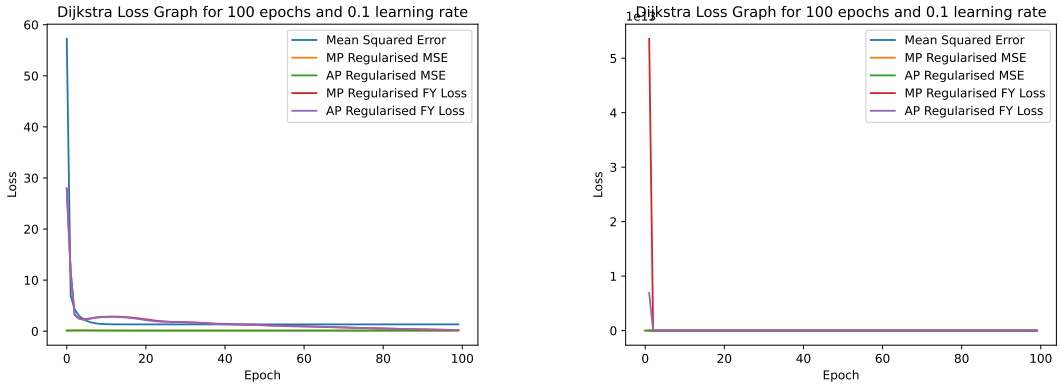

Figure 35: Training and test losses for all architectures with an epoch size of 100 and a learning rate of 0.1

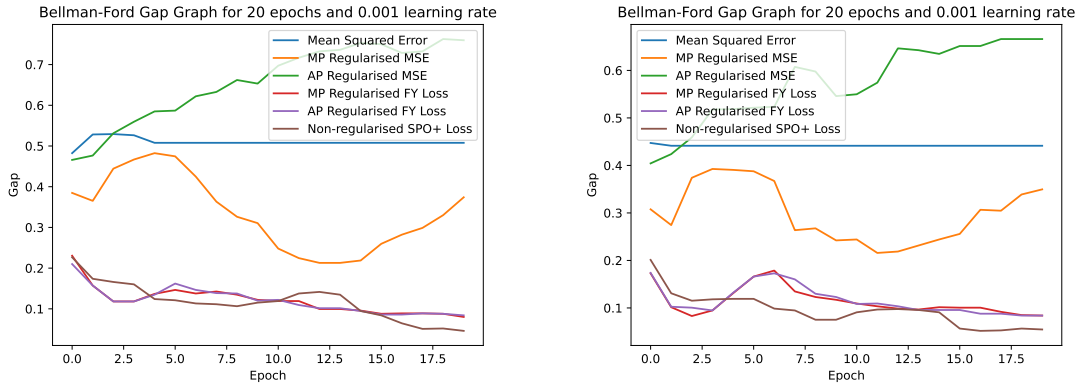

Figure 36: Training and test gaps for all architectures with an epoch size of 20 and a learning rate of 0.001

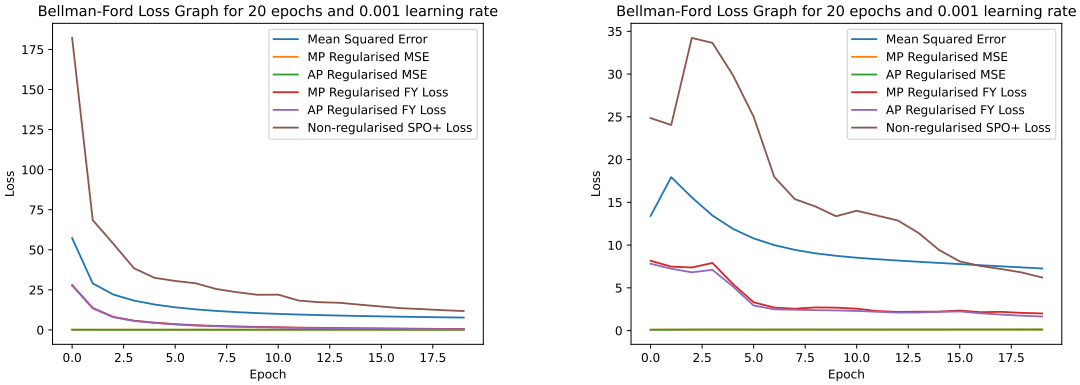

Figure 37: Training and test losses for all architectures with an epoch size of 20 and a learning rate of 0.001

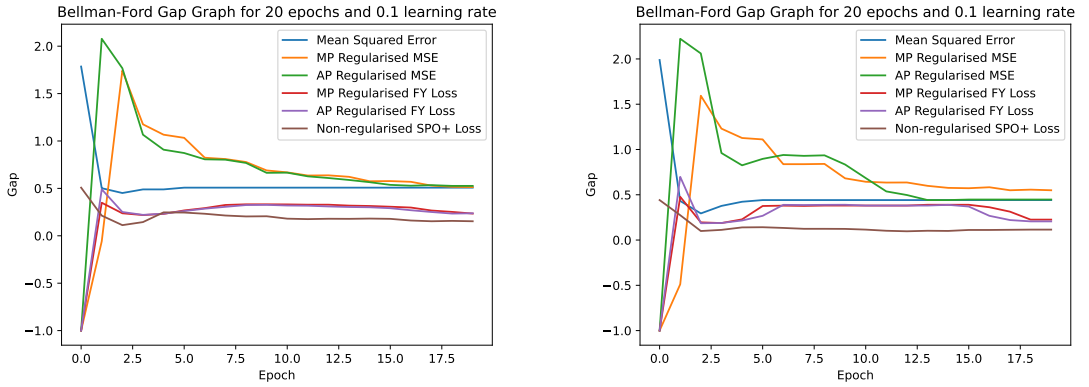

Figure 38: Training and test gaps for all architectures with an epoch size of 20 and a learning rate of 0.1

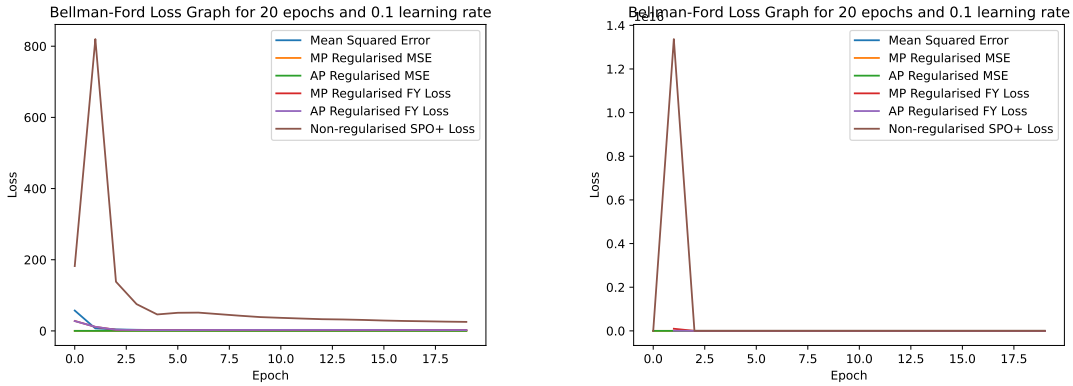

Figure 39: Training and test losses for all architectures with an epoch size of 20 and a learning rate of 0.1

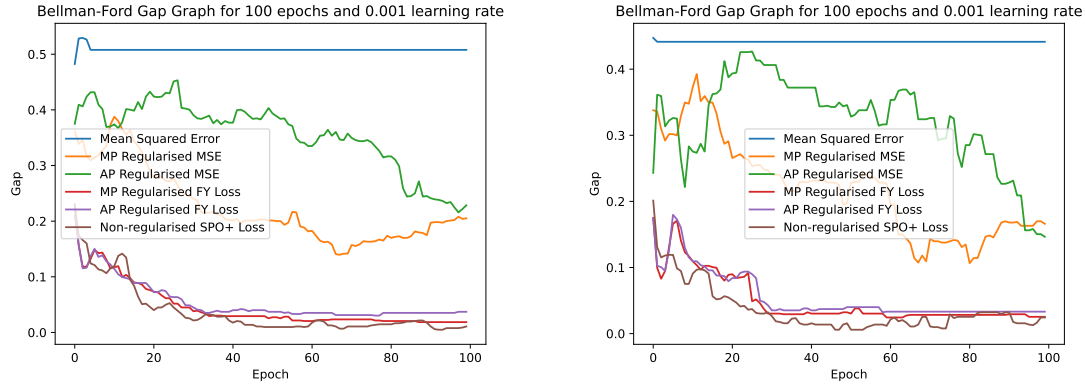

Figure 40: Training and test gaps for all architectures with an epoch size of 100 and a learning rate of 0.001

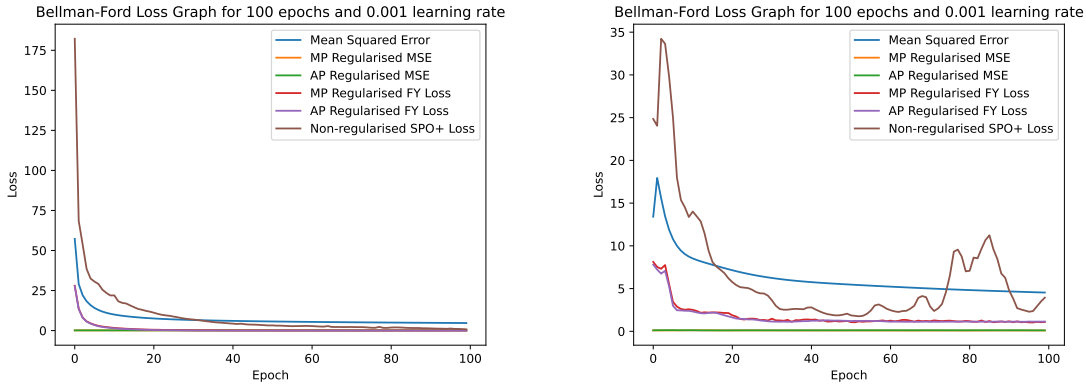

Figure 41: Training and test losses for all architectures with an epoch size of 100 and a learning rate of 0.001

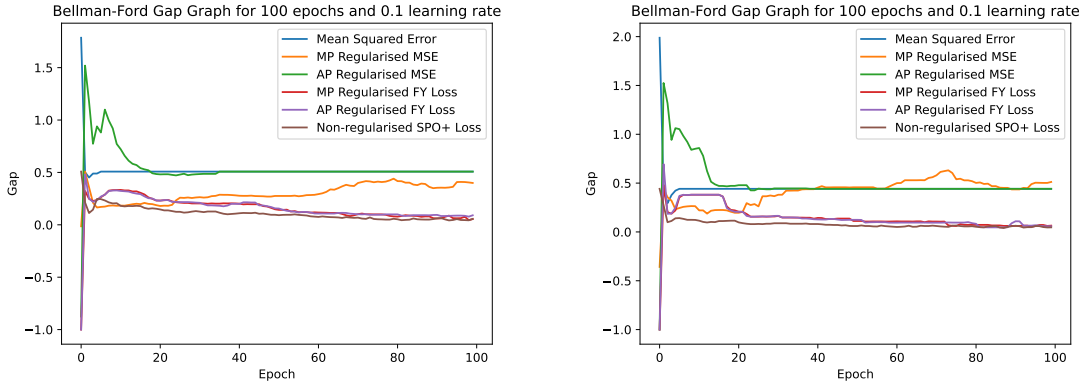

Figure 42: Training and test gaps for all architectures with an epoch size of 100 and a learning rate of 0.1

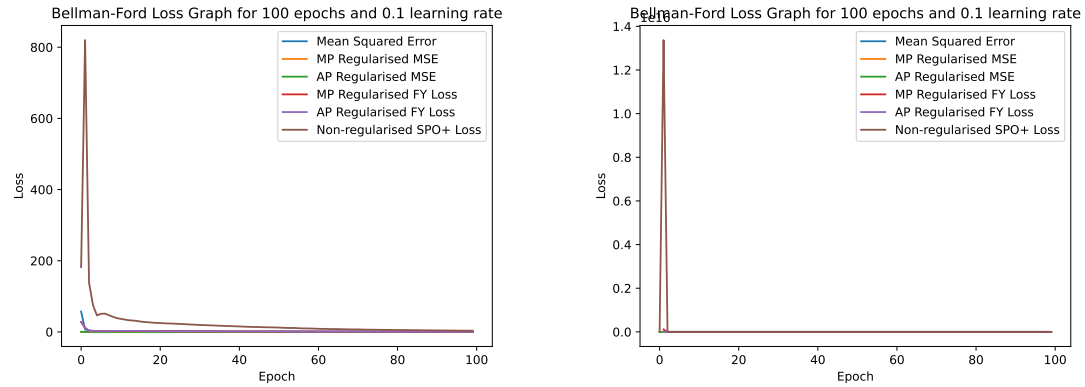

Figure 43: Training and test losses for all architectures with an epoch size of 100 and a learning rate of 0.1