CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Combinatorial Optimization with Reinforcement Learning

Solving Heterogenous Capacitated Vehicle Problem Iteratively with Attention Networks and Reinforcement Learning

Master's thesis in Computer science and engineering

ALADDIN PERSSON HIJAZI
SANNA PERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Combinatorial Optimization with Reinforcement Learning

Solving Heterogenous Capacitated Vehicle Problem Iteratively with Attention Networks and Reinforcement Learning

ALADDIN PERSSON HIJAZI
SANNA PERSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Combinatorial Optimization with Reinforcement Learning
Solving Heterogenous Capacitated Vehicle Problem Iteratively with Attention Networks and Reinforcement Learning
ALADDIN PERSSON HIJAZI SANNA PERSSON

Supervisor: Peter Damaschke, Computer Science and Engineering
Examiner: Jean-Philippe Bernardy, Philosophy, Linguistics and Theory of Science

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2023

Combinatorial Optimization with Reinforcement Learning
Solving Heterogenous Capacitated Vehicle Problem Iteratively with Attention Networks and Reinforcement Learning
ALADDIN PERSSON HIJAZI
SANNA PERSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This master's thesis delves into the topic of solving combinatorial optimization problems with methods based on reinforcement learning, and specifically, we explore the potential of iterative route decoding and gradient updates in enhancing the performance of route decoding. In this context, route decoding refers to determining the most efficient route for a set of destinations, a combinatorial optimization problem often encountered in logistics and transportation planning. We introduce two methods for iteratively updating solutions for the heterogeneous capacitated vehicle routing problems. They are built upon a reinforcement learning algorithm with an attention graph encoder and use previously computed routes for an instance to improve solution quality. Our results show improved performance, in particular, on out-of-distribution data, which suggests the practical applicability of the methods. In particular, our results show that a pre-trained route planner can, with a few gradient updates with a policy gradient method, significantly improve on out-of-distribution data.

Keywords: Combinatorial optimization, reinforcement learning.

Acknowledgements

We would like to express our gratitude to our supervisor Peter Damaschke for stepping in the role as our main supervisor at a critical point in our project and being a key for the completion of our thesis.

We would also like extend a thank you to NaN Intelligence for providing us with GPUs for our experiments.

Aladdin Persson Hijazi, Sanna Persson, Gothenburg, 2023-09-24

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

There is growing experimental evidence that deep neural networks can learn effective planning functions for games with imperfect information and complex system dynamics [1]. Despite this, limited research has been conducted on the application of these modern techniques in the field of combinatorial optimization [2]. Deep learning methods have demonstrated remarkable improvements in solution speed for combinatorial optimization problems compared to heuristics [3]. However, state-of-the-art heuristics still outperform deep learning in terms of accuracy for many combinatorial optimization problems, indicating a performance gap that can be bridged through deep learning, similar to how computer vision and natural language processing have evolved over the past decade.

There are many problems in scheduling and route planning in the field of combinatorial optimization, which are prohibitively expensive to solve exactly for large problem instances. For some such problems, there has been significant progress in developing approximate or exact algorithms that use the problem structure to find near-optimal solutions. For many problems, it is still an open research area to find such solvers, and there is great interest in developing fast solvers for more complex routing and scheduling tasks that can give near real-time solutions with acceptable precision. An extension of this is to produce a solver that can produce iteratively more accurate solutions depending on a given solution time. For traditional heuristic algorithms developed on combinatorial optimization problems, it is common that these solvers do not scale well with the problem size, posing issues for some industrial applications. A modern take on developing approximate algorithms for combinatorial optimization problems has been to look at the recent advances in deep learning. With this approach, the internal heuristics and algorithms are not hand-designed but instead learned from data.

Recent experiments [3] have shown promising results with models using attention mechanisms popularized by language models or graph networks [4]. However, these models have yet to be tested on large-scale combinatorial optimization problems, possibly due to the quadratic memory requirements of attention models [5].

We hypothesize that reinforcement learning and geometric neural networks can lead to a model that performs well on problem sizes that may otherwise be intractable

with traditional and our work considers the problem of solving combinatorial optimization problems approximately with deep learning. The project mainly focused on vehicle routing problems and solving them with a deep learning approach that can improve its solutions iteratively.

1.2 Problem definition

Combinatorial optimization problems are a class of problems in computer science that involve selecting the best possible solution from a finite set of options. In this report, we will focus on vehicle routing problems and explore the use of deep learning techniques to solve these problems. Specifically, we are tackling ways to iterate on an already defined solution to improve the objective.

1.2.1 Combinatorial optimization problems

The field of combinatorial optimization encompasses a wide range of problems, including vehicle routing problems, scheduling problems, and finding spanning trees. These problems are often difficult to solve because they involve many variables and constraints. Additionally, many combinatorial optimization problems are NP-hard (definition: 2.2.3), which means that it is not possible to find an exact solution in a reasonable amount of time for large problem sizes. Instead, researchers often focus on developing approximate algorithms that can find high-quality solutions in a reasonable amount of time. The subject of combinatorial optimization is further introduced in Chapter 2.

1.2.2 Deep learning in combinatorial optimization

A common approach in practice to solving NP-hard combinatorial optimization problems is through approximations. It is an active area of research finding more accurate and faster approximate algorithms for these problems. The quality of approximate algorithms can be measured by their optimality gap to an optimal solution and the running time for different instances. One promising approach to solving combinatorial optimization problems is to use deep learning techniques. Deep learning models can be used to guide the search process in combinatorial optimization problems by predicting the quality of a potential solution or identifying promising areas of the solution space. There are examples of deep learning models [3], [4] that can solve vehicle routing problems with adequate optimality gaps and running times, though there are still open questions within the field. Traditional methods are still state-of-the-art for optimality gaps on many problems [3], deep learning methods still face challenges on larger instances, and further, there is a question of how to generalize a solver to different combinatorial optimization problems.

1.2.3 Graph neural networks

In vehicle routing problems, the instance is represented by a graph, and one of the challenges with solving them with deep learning is to encode the graph for

the model to learn valuable representations for route planning. There are neural network architectures developed explicitly for graph problems, however, they are traditionally developed for problems different than routing problems [6]–[8] and in Lei et al. [4] they modified a specific type of graph neural networks for the task of route optimization. There is, however, yet to be a strong consensus within the research community of the best architectures for different combinatorial optimization problems [9].

1.2.4 The problem with supervision

A challenge for training deep learning models for combinatorial optimization problems is training data. It is prohibitively expensive to generate a large labeled dataset for the larger instances because there are no fast solvers. Cappart et al. [9] addresses an additional limitation of supervised learning for combinatorial optimization: the problem of producing difficult labeled examples. Due to computational costs, it is common to base labeled data on simpler instances that are solvable in polynomial time and then extend them. The resulting dataset can then give a skewed sample of the entire problem space [9]. A model trained with supervised learning on such a dataset will then mainly learn to solve the problems on simpler instances, which may limit the applicability to real-world instances. Another approach used in recent deep learning approaches [3], [4] is to train the model on simulated data with reinforcement learning. With this method, the exact solution does not need to be computed to produce training data. Training data can then be produced by randomly sampling instances, and reinforcement learning does not need the ground-truth labels for its learning algorithm.

1.3 Objective

In this thesis, the aim is to determine if improved approaches can be developed for large-scale combinatorial optimization problems, with a focus on iterative solution approaches. The primary focus will be on the vehicle routing problem, and it will involve building an architecture based on graph neural networks to encode problem instances and using reinforcement learning to train the model to find approximate solutions iteratively. The research will concentrate on iterative solution methods and the reinforcement learning training paradigm. The resulting model will be evaluated on common benchmarks for vehicle routing problems.

1.4 Research questions

The thesis work aims to address the following questions:

- Does conditioning on a feasible solution improve solution quality?
- Can solution quality be augmented by iteratively updating the solution?
- Is it possible to adapt modern reinforcement learning algorithms, such as MuZero (see section 3.4.3), for route planning?

- Can the graph-attention layer be enhanced for routing problems?

1.5 Limitations

We will primarily focus on experimental results on standard benchmarks for the models we develop and test. In particular, we will not be delving into conventional metrics commonly used to evaluate optimization algorithms, such as complexity or approximation ratios. Additionally, we have excluded worst-case convergence bounds from our scope, as real-world problem instances typically exhibit better complexity than the worst-case scenario [10]. Furthermore, we will not prove complexity bounds for the proposed model architecture and search strategies.

2

Combinatorial optimization - problems and algorithms

2.1 Combinatorial optimization

Combinatorial optimization (see Korte and Vygen [11]) is a field of mathematics and computer science including problems with the objective to find the best solution from a finite set of discrete choices. It deals with optimization problems where the objective is to find the best combination of elements from a given set of elements to satisfy certain criteria. These criteria can be to maximize or minimize some objective function, depending on the problem at hand. In other words, combinatorial optimization involves finding the most efficient way of combining different elements to solve a problem where the set of elements is finite, and the solution must be chosen from this set. This can be applied to various real-world problems, such as scheduling, network design, resource allocation, and many others.

The problems in combinatorial optimization are often NP-hard (defined below), meaning that no known algorithm can solve them in polynomial time. As a result, researchers in the field often develop heuristics or approximation algorithms that can find near-optimal solutions in a reasonable amount of time. These algorithms systematically explore the search space, guided by some criterion, such as the objective function or problem constraints.

2.2 Complexity: P vs NP

A decision problem is that which can be answered with a single yes/no. One example is: given a number of cities on a map, can a tour be found that visits each city only once and returns to the starting town? This particular problem is known as the Traveling Salesman problem and will be defined in detail later.

Definition 2.2.1 (NP)**.** NP (nondeterministic polynomial time) is the class of decision problems for which a solution can be verified in polynomial time [11].

If the problem is also solvable in polynomial time, it belongs to the class P. In the current state of research, there are many problems for which it is unknown whether they can be solved in polynomial time. Specifically, we will define the class of NP-complete problems that is important in complexity theory.

Definition 2.2.2 (NP-completeness)**.** A problem, X that is NP-complete has the properties that

1. $X \in NP$
2. For all $Y \in NP$, $Y \neq X$, Y is polynomially reducible to X.

Polynomially reducible means that there is a polynomial-time algorithm that reduces any instance of Y into an instance of X such that the original instance of Y is a 'yes' instance if and only if the resulting instance of X is a 'yes' instance. Furthermore, we define the class of NP-hard problems:

Definition 2.2.3 (NP-hardness)**.** A problem, X, belongs to the class NP-hard if it has the property that all problems $Y \in NP$ are polynomially reducible to X.

The difference to NP-complete problems is that an NP-hard problem does not need to be in the class of NP.

Any algorithm that solves one of the NP-hard problems in polynomial time can, therefore, solve the entire class of NP problems. It has been proven that $P \subseteq NP$ indicates that every problem that can be solved in polynomial time can also have its solution verified in polynomial time. However, whether these two complexity classes are identical remains an open question, i.e., whether $P = NP$. There is significant scepticism in computer science that this would be the case, though and in practice, approximation algorithms are a common approach for problems without known polynomial algorithms.

2.3 Graph problems

We introduce a general graph notation (see Korte and Vygen [11, Chapter 2] for further details), which will be used to define the specific problems that are of interest. We start by defining a graph in its simplest form:

Definition 2.3.1 (Undirected graph)**.** An undirected graph is a pair of sets: $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each edge $e \in E$ is defined as $e = (u, v)$ for $u, v \in V$ and $u \neq v$. That is, each edge consists of two connected nodes that are not the same node.

A directed edge is an edge $(u, v) \in E$ for which the edge can be traversed from u to v but not in the opposite direction.

Definition 2.3.2 (Directed graph)**.** A directed graph $G = (V, E)$ is a graph where V is the set of nodes and E is the set of directed edges between the nodes.

Definition 2.3.3 (Weighted graph)**.** A weighted graph $G = (V, E, w)$ is a graph where $w : E \to \mathbb{R}_+$ is a function that assigns a non-negative weight to each edge in E.

In a weighted graph, the weights can be interpreted as a cost for traversing the edges. We can also introduce more properties in a graph, such as node demands, where each node requires a certain load to be delivered to them, or time windows,

such that each node can only be visited within a certain time. These constraints often make graph problems harder to solve, which will become clear as we consider specific combinatorial optimization problems.

Many important problems of graphs address finding ways to traverse or search over them. Two important concepts for this are path and cycle.

Definition 2.3.4 (Path)**.** A path $P = (v_1, v_2, \ldots, v_k)$ in a graph, is a set of nodes where $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$. It also holds that no node is revisited in a path: $v_i \neq v_j$ for any $v_i, v_j \in P$.

Colloquially, a path in a graph can be described as one way to travel in the graph with no constraints on how many or which nodes are to be included.

Definition 2.3.5 (Cycle)**.** A cycle $C = (v_1, v_2, \ldots, v_k)$ in a graph, where $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$ and $(v_k, v_1) \in E$. It also holds that no node is revisited in a cycle: $v_i \neq v_j$ for any $v_i, v_j \in C$.

If every node in the graph is visited once in C, the cycle is referred to as a Hamiltonian cycle. The aforementioned definitions will be used to define vehicle routing problems on graphs.

2.4 Vehicle routing problems

Route planning problems often involve variations of the vehicle routing problem (VRP), where the objective is to find a set of routes in a graph that satisfy customer demands and start and end at a specific depot node [12].

A vehicle routing problem (VRP) can be represented by a graph $G = (V, E, c, d)$, where V is the set of nodes representing customers or depots, E is the set of directed edges between the nodes and $c : E \to \mathbb{R}+$ is a function that assigns a non-negative cost or distance to each edge in E. In addition, each node $v \in V$ is associated with a demand $d_v \in \mathbb{R}$ that represents the number of goods to be delivered or picked up at that node with positive numbers representing goods to be delivered. The objective of VRP is to find a set of routes for a fleet of vehicles that start and end at a depot, visit all customers exactly once, and satisfy the capacity constraints of the vehicles while minimizing the total cost or distance traveled.

2.4.1 Stochastic travelling salesman problem (TSP)

The stochastic travelling salesman problem is a combinatorial optimization problem that extends the travelling salesman problem (TSP).

Definition 2.4.1 (TSP)**.** Consider a graph $G = (V, E, c)$ where each edge $e \in E$ has cost c_e. The objective is to find a tour (Hamiltonian cycle) starting from and returning to a specific node $s \in V$ that visits all the nodes in the graph exactly one time. The optimal tour has minimal total edge costs.

A stochastic version of TSP is given in Toriello et al. [13] where each edge cost is realized once when the tour arrives at one of the nodes. That is, each edge has a

probability distribution of its cost, and only when the tour reaches a specific node, the actual cost of the connected edges are known. The true probability distribution of the edge costs is unknown to the routing algorithm. The goal is then to find a policy that decides on a tour depending on the current state in the instance, which gives the minimal expected cost of the tour [13]. The key is that the policy must decide on the tour with partial information of only the adjacent edge costs to the already visited nodes.

In simulated data of the stochastic TSP, the probability distribution of the edge costs must be carefully chosen to match the distributions seen in the intended applications.

2.4.2 Capacitated vehicle routing problem

One variation of VRP is the capacitated vehicle routing problem (CVRP), which involves satisfying delivery demands with a fleet of vehicles while minimizing the distance traveled [14]. The capacitated vehicle routing problem is a multi-vehicle routing delivery problem. There are several variants of the problem, and this section only considers one. A problem instance of CVRP can be represented by a graph such that each node is a customer with a demand. The goal is to satisfy the delivery demands with a fleet of vehicles while minimizing the total distance traveled by all vehicles in the fleet [14].

Following the definition of CVRP in [14], the problem can be formalized as:

Definition 2.4.2 (CVRP)**.** Input: graph $G = (X, E, c, d)$ and a fleet V of k vehicles. The set of nodes is denoted X and represents customers or depots, E is the set of directed edges between the nodes and $c : E \to \mathbb{R}+$ is a function that assigns a non-negative cost or distance to each edge in E. Each node $x \in X$ is associated with a demand $d_x \in \mathbb{R}+$. Each vehicle in the fleet has an associated capacity Q which cannot be exceeded.

The objective is to find a tour for each vehicle in the fleet with minimal total cost while fulfilling the following constraints

1. All vehicles start at and return to the same depot node.
2. Each node is visited once by one of the vehicles in the fleet, and its demand must be completely fulfilled by one delivery.
3. The total capacity Q for a vehicle $v \in V$ cannot be exceeded by the sum of the visited nodes' demands.
4. At most k vehicles can be used to satisfy the delivery constraints.

In the heterogeneous CVRP, some of the capacities for the vehicles are different [3].

2.4.3 Vehicle routing problem with time windows

A variation of VRP is the VRP with time windows (VRPTW), where each customer must be satisfied within a specified time window [12].

We consider VRPTW as an extension of CVRP. The two problems have the same objective: to find tours with a minimal total cost that fulfill specific constraints of delivery for the demand nodes given a vehicle fleet. According to the definition in [12] for each node $x \in X$, we introduce time constraints, $(x_a, x_b) \in \mathbb{R}^2+$. The time constraints are fulfilled if node x is visited after x_a and before x_b. To formalize this, we re-define the edge costs so that both a cost c_e and a time t_e are associated with each edge $e \in E$. The constraint we impose is then that the sum of all traversed edges before visiting x must exceed x_a and be less than x_b. If the delivery to x occurs at $x_i < x_a$, the vehicle has to wait until x_a, and the total time is then x_a.

There can be both soft and hard constraints for the time windows [12]. If the time constraint is soft, then there is only a penalty for exceeding the delivery time x_b for $x \in X$. For hard time limits, only solutions within the time limits are feasible.

2.5 Algorithms for routing problems

In this section, we will introduce a few important algorithms for route planning that go beyond the brute-force solution of the problem. The focus will be on approximation algorithms since these generally are faster in practice while providing sufficient optimality bounds for many applications.

2.5.1 Greedy algorithm for CVRP

The problem of CVRP is hard to solve exactly, but we propose a simple greedy algorithm for finding a feasible solution. We have a fleet of vehicles with limited capacities that need to serve a set of customers located in different geographical locations. The objective function is to minimize the total distance traveled by vehicles while ensuring that each customer is served exactly once and that each vehicle's capacity is not exceeded by the total demand it serves.

A greedy approach for the CVRP can be constructed as the following

1. Sort the customers in non-increasing order of their demand.
2. Initialize a set of empty routes and a set of unassigned customers.
3. While there are unassigned customers, do the following:
4. Select the customer with the largest demand that has not been assigned to a route.
5. Find the route with the smallest remaining capacity that can serve the selected customer and add the customer to the route.
6. If no route can serve the customer, create a new route with the selected customer.
7. Repeat steps 4-6 until all customers are assigned to a route.

The algorithm is further described in pseudocode in Algorithm 1.

Algorithm 1: Greedy Algorithm for Capacitated Vehicle Routing Problem

Input: An undirected graph G, integer c
Result: A node set U to delete from G.
Sort the customers in non-increasing order of their demand;
Initialize a set of empty routes and a set of unassigned customers;
while *there are unassigned customers* **do**
    Select the customer with the greatest demand that has not been assigned to a route.
    Find the route with the smallest remaining capacity that can serve the selected customer and add the customer to the route.
    if *no route can serve the customer* **then**
        Create a new route with the selected customer.
    end
end
Return the set of routes.

Note that the algorithm assumes that there is at least one vehicle with sufficient capacity to serve the largest customer demand. If this is not the case, the algorithm may fail to find a feasible solution. Additionally, the quality of the solution found by the algorithm depends on the order in which the customers are sorted, and there is no guarantee that the solution is optimal.

2.5.2 Traditional methods for vehicle routing problems

In this section, we introduce three different methods for solving CVRP that do not base their method on deep learning: SISR, VNS, and FA. We denote these as traditional methods in this context, but it does not reflect their performance. All these methods are used to report baseline performance on the CVRP problem due to their performance in terms of optimality gap on benchmarks.

2.5.2.1 Slack Induction by String Removals

J. Christiaens and G. Vanden Berghe [15] propose an approach for solving vehicle routing problems (VRPs) that uses a slack-induction heuristic. The slack induction by string removals (SISR) algorithm refers to strings as a sequence of customers in a graph that are part of a tour.

The slack-induction heuristic involves removing strings of nodes from a route and then reinserting them in a way that maximizes the slack available for future insertions. The authors propose criteria for choosing which and how many nodes are removed in a string.

The approach is tested on a set of benchmark instances and compared to several other state-of-the-art methods. The results show that the slack-induction heuristic with string removals performs competitively with other methods and outperforms some of them in certain instances.

2.5.2.2 Variable Neighborhood Search

The Variable Neighborhood Search (VNS) Algorithm was introduced by Xu et al. [16] and consists of two stages. In the first stage, an approximately optimized solution is found, and if it satisfies a certain quality, the second stage optimizes the solution further.

The algorithm introduces a special shaking method and a method for computing time difference excess. The shaking method is proven more effective than ordinary methods by experiments. The algorithm begins by generating an initial template solution. A template solution consists of all the customer points rather than only customer points served on multiple days. The quality of a solution is evaluated by its "cost", calculated by adding the penalty for infeasibility to the total travel time.

The algorithm then employs a local search process with three operations to shift the current template solution to a local optimum: relocation, exchange, and reverse. The cost change caused by the operations is calculated without resolving the template solution. The computation time is reduced by saving the optimum operations performed in each individual route and between each pair of routes.

The shaking step shifts points of the template solution to random positions for diversification. The number of shifted points determines a neighborhood for shaking. The second stage is then applied to make the solution feasible with regard to all the constraints.

2.5.2.3 Ant Colony System

Palma-Blanco et al. [17] presents an approach to solving the Heterogeneous Vehicle Routing Problem with Time Windows, Multiple Products, and Product Incompatibility (HVRPTWMPIC). This is a variant of the vehicle routing problem (VRP). Their method is a two-pheromone trail Ant Colony System (ACS) approach to find feasible solutions to minimize routing costs and vehicle fleet size.

The method uses a colony of cooperative agents (ants) to build feasible solutions for the HVRPTWMPIC. The process begins with the initialization of parameters and routes for each agent. Each agent then constructs a feasible solution based on a decision that evaluates the desirability of the next customer in the short term and long term.

The construction of feasible solutions involves each agent deciding between exploiting its obtained knowledge or exploring new possibilities. An exploration rate guides this decision, and once a feasible solution is found, two local search procedures are performed, similar to VNS.

3

Deep learning

3.1 Machine learning and feed-forward networks

Deep learning is an area in Machine learning, and we begin by introducing the topic of Machine learning. In Figure 3.1 the relationships between the different subjects in artificial intelligence are visualized. Machine learning is the field of studying learning algorithms, and a learning algorithm is a computer program that improves using data. Mitchell [18] defines a learning algorithm as a computer program that learns from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. An example of this is an algorithm that improves on the task of classifying images (T) by seeing images and their corresponding labels (E) as measured by the accuracy (P) of the classifications.

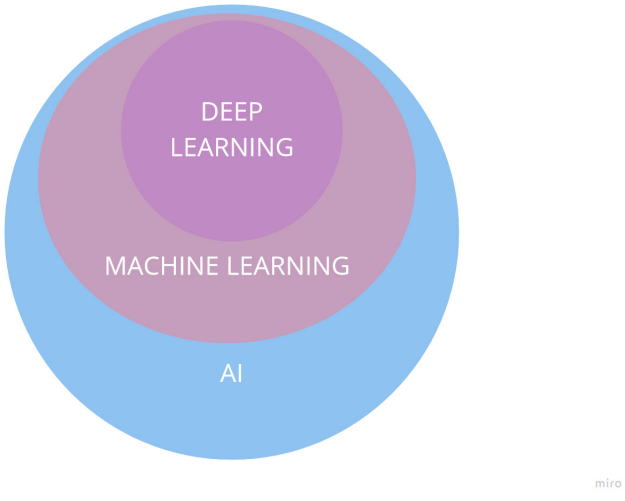

Figure 3.1: Venn diagram of relationships between the subjects of artificial intelligence (AI), machine learning and deep learning.

A foundational machine learning algorithm is the feed-forward neural network or multilayer perceptron. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ that learns the parameter $\boldsymbol{\theta}$ such that it best approximates the goal function, f^*, where

$y = f^*(\boldsymbol{x})$ [19]. Goodfellow et al. [19] state that the main characteristic of a feed-forward network is that the information flows from the input through intermediate computations in f to the output without any loops. In practice, the intermediate computations are commonly defined in the same way, and an example of a feed-forward network can be seen in Figure **??**.

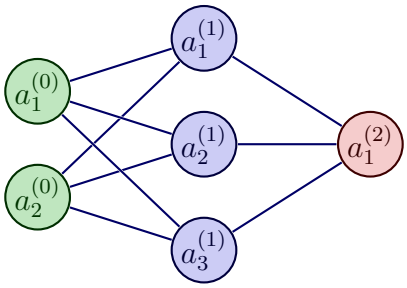

Figure 3.2: The figure illustrates a simple, fully connected network with one hidden layer. The node notation specifies the layer index and number of hidden nodes in each layer, $a^{layer}_{hiddendim}$.

The network in Figure **??** has an input layer, one hidden layer and an output node. Each line between the nodes represents a parameter $w_{ij}^{(k)}$. There is an additional parameter internally in each node denoted $b_j^{(k)}$, known as a bias. In the hidden layer, the computation for each node, $a_i^{(1)}$, is done in two steps. First an intermediate value $z_i^{(1)}$ is defined as

$$z_i^{(1)} = \left(\boldsymbol{w^{(1)}}^T \boldsymbol{a}^{(0)} + b^{(1)}\right)_i, \tag{3.1}$$

where

$$\boldsymbol{w^{(1)}} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

and

$$\boldsymbol{b^{(1)}} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

The input to the network $\boldsymbol{a}^{(0)} \in \mathbf{R}^2$ is a vector with two features in this example. To approximate a nonlinear mapping between the input and the output, it is common practice to apply an element-wise nonlinear function. In this example, g_1 is applied to each $z_i^{(1)}$ such that

$$a_i^{(1)} = g_1(z_i^{(1)}). \tag{3.2}$$

These nonlinear functions are known as activation functions, and a common function is the rectified linear unit and variations of it [19].

Definition 3.1.1 (Rectified Linear Unit)**.** The Rectified Linear Unit (ReLU) is defined as

$$g(x) = \max\{0, x\},$$

for variable x.

With g_1 defined as the ReLU function $a_i^{(1)}$ is computed as

$$a_i^{(1)} = \max\{0, z_i^{(1)}\}. \tag{3.3}$$

The output layer is, by convention, computed as an affine transformation of $a^{(1)}$ with parameters $w^{(2)}$ and $b^{(2)}$ with the activation function being an identity mapping:

$$a^{(2)} = z^{(2)} = \left(\boldsymbol{w^{(2)}}^T \boldsymbol{a}^{(1)} + b^{(2)}\right). \tag{3.4}$$

In equation (3.4), the parameters $w^{(1)}$, $b^{(1)}$, $w^{(2)}$ and $b^{(2)}$ which values have to be chosen carefully for the function which the network aims to approximate. A common learning algorithm is gradient-based learning with back-propagation and gradient descent. It is the most common learning paradigm used within most deep learning applications [19].

We demonstrate how back-propagation with gradient descent works on the small example in Figure **??**. Once the network is defined with its parameters, the learning algorithm needs to optimize them to minimize the difference between the predicted and true outputs. One way to do this is through the use of an optimization algorithm such as gradient descent. The idea behind gradient descent is to iteratively update the parameters in the direction of the steepest descent of the loss function [19]. The loss function measures the discrepancy between the predicted and true output and is typically chosen as a differentiable function.

In the case of the feed-forward network in Figure **??**, the loss function could be the mean squared error (MSE) between the predicted output $a^{(2)}$ and the true output y for a given input $a^{(0)}$ and corresponding label y:

$$\mathcal{L}\left(\boldsymbol{w^{(1)}}, \boldsymbol{b^{(1)}}, \boldsymbol{w^{(2)}}, b^{(2)}\right) = \frac{1}{2}\left(a^{(2)} - y\right)^2$$

The goal of the learning algorithm is to find the values of the parameters that minimize the loss function. To do this, we use gradient descent to update the parameters in the direction of the negative gradient of the loss function with respect to the parameters:

$$w_{i,j}^{(k)} \leftarrow w_{i,j}^{(k)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(k)}},$$

for the weights, where α is a scaling parameter of the gradient called the learning rate.

We can use the back-propagation algorithm to compute the gradient of the loss function with respect to the parameters. The back-propagation algorithm starts by computing the loss of the network given input and then works backwards to compute the gradients of the loss function with respect to each parameter. This is done using the chain rule of differentiation, which states that the derivative of a composite function is the product of the derivatives of its component functions. First, we compute the derivative of the loss with respect to $a^{(2)}$:

$$\frac{\partial \mathcal{L}}{\partial a^{(2)}} = a^{(2)} - y$$

Then, we can compute the derivative of $a^{(2)}$ for $z^{(2)}$ since the activation function in the output layer is the identity function, this derivative is simply 1:

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = 1$$

Using the chain rule, we can compute the gradient of the loss to $z^{(2)}$:

$$\frac{\partial \mathcal{L}}{\partial z^{(2)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} = a^{(2)} - y$$

The gradient of the loss with respect to $\boldsymbol{w^{(2)}}$ can then be calculated as:

$$\frac{\partial z^{(2)}}{\partial \boldsymbol{w^{(2)}}} = \boldsymbol{a^{(1)}}$$

Similarly, we can compute the gradient of the loss for $b^{(2)}$:

$$\frac{\partial z^{(2)}}{\partial b^{(2)}} = 1$$

The derivative of the loss with respect to $z^{(1)}$ using the chain rule is given by:

$$\frac{\partial \mathcal{L}}{\partial z^{(1)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial \boldsymbol{a^{(1)}}} \frac{\partial \boldsymbol{a^{(1)}}}{\partial z^{(1)}} = \left(a^{(2)} - y\right) \boldsymbol{w^{(2)}} g_1' \left(\boldsymbol{z^{(1)}}\right)$$

where $g_1' \left(\boldsymbol{z^{(1)}}\right)$ is the derivative of the activation function g_1 with respect to $\boldsymbol{z^{(1)}}$. Finally, we can compute the derivative of the loss with respect to $\boldsymbol{w^{(1)}}$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w^{(1)}}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z^{(1)}}} \frac{\partial z^{(1)}}{\partial \boldsymbol{w^{(1)}}} = \boldsymbol{a}^{(0)} \frac{\partial \mathcal{L}^T}{\partial \boldsymbol{z^{(1)}}}$$

Similarly, we can compute the derivative of the loss to $b^{(1)}$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z^{(1)}}} \frac{\partial \boldsymbol{z^{(1)}}}{\partial w^{(1)}} = \frac{\partial \mathcal{L}^T}{\partial \boldsymbol{z^{(1)}}}$$

The updates with gradient descent for all parameters are:

$$w_{i,j}^{(1)} \leftarrow w_{i,j}^{(1)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(1)}}$$
$$b_i^{(1)} \leftarrow b_i^{(1)} - \alpha \frac{\partial \mathcal{L}}{\partial b_i^{(1)}}$$
$$w_{i,j}^{(2)} \leftarrow w_{i,j}^{(2)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(2)}}$$
$$b^{(2)} \leftarrow b^{(2)} - \alpha \frac{\partial \mathcal{L}}{\partial b^{(2)}}$$

In this example, we consider how to perform gradient-based learning with back-propagation and gradient descent for a single input vector $\boldsymbol{a^{(0)}}$. In practice, the gradient updates are done on a dataset of N pairs $X = \{(x_0, y_0), ...(x_N, y_N)$ and if N is large, sampling of a batch of n samples is done in each step of the learning algorithm. If the gradient descent updates are done on samples in a dataset, it is called stochastic gradient descent (SGD) [19].

3.2 Transformer models

The transformer model, introduced by Vaswani et al. [20] has become a widespread architecture within the field of natural language processing (NLP) with its architecture being behind both the newly released GPT-4 [21] and InstructGPT [22] (commonly known as ChatGPT). There are also successful applications of transformers in computer vision [23], [24] as well as in the field of combinatorial optimization [3].

Transformers are a type of attention-based architecture, and the idea behind attention is to allow models to focus on relevant parts of the input. Attention models use a mechanism that selectively weighs the importance of each input element based on its relevance to the current task. This has been shown to be beneficial for longer inputs where one application is machine translation.

The transformer consists of an encoder and a decoder, both of which employ multi-head self-attention to capture the correlations between input and target sequences. In Vaswani et al. [20] they introduce the scaled-dot-product-attention, which is the core of the multi-head self-attention. The self-attention mechanism introduces the concepts of keys, queries and values. Both keys and queries are vectors of dimension d_k, and the values are a vector of dimension d_v [20]. In the implementation of the transformer, the keys, queries and values are different mappings of the inputs and labels/output. The self-attention mechanism then produces a weighting of input and previous output for the task of next-word prediction. The scaled-dot-product-attention function is computed as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where Q, K and V are queries, keys and values respectively and $Q, K \in \mathbb{R}^{d_k}$. The softmax function is computed as

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{d_k} e^{z_j}} \text{ for } i = 1, \ldots, d_k \text{ and } \mathbf{z} = (z_1, \ldots, z_{d_k}) \in \mathbb{R}^{d_k},$$

where $z \in \mathbb{R}^{d_k \times d_k}$ and the result has dimensions $\text{softmax}(\mathbf{z}) \in \mathbb{R}^{d_k}$ in the case of scaled-dot-product-attention. In multi-head attention, the input sequence is instead mapped to multiple queries, keys, and values, and the attention mechanism is applied to each of them in parallel. The output of multi-head attention is then concatenated and passed through a feed-forward neural network.

Attention-based models using transformer architectures have proven to be especially useful in contexts with long and complex inputs. A drawback of the original transformer architecture by Vaswani et al. [20] is the architecture's quadratic memory and time requirements. As a result, several papers have proposed modifications to the original transformer architecture to reduce its time and memory complexity while maintaining its effectiveness [25]–[27].

For the task of combinatorial optimization, the transformer architecture has, for example, been employed for the task of producing node embeddings in vehicle routing tasks. In [3] the coordinates of the nodes and their demands are modeled as a

sequence, and a representation is learned with a transformers model. Further, in vision transformers, [24], images are modeled as sequences of image patches showing the range of areas for which transformers have been applied.

3.3 Graph neural networks

The transformer architecture has been applied to a variety of tasks, one being graph problems; there are, however, also neural network architectures, especially developed applications on graphs. Neural network architectures for applications on graphs were introduced by, among others, Gori et al. [6]. The original graph neural networks based their architecture on the idea that nodes in a graph represent objects and edges represent their relationships. Many types of data, such as transactions, social relationships, and traffic routes, are best represented by graphs. A whole area of deep learning is therefore dedicated to learning prediction tasks on a graph. These models can be used to find fraudulent customers at a bank from a transaction network or predict possible matches on dating sites.

3.3.1 Graph convolutional networks (GCNs)

Graph convolutional neural networks (GCNs) introduced by Kipf and Welling [7] is one of the most used architectures to this day and one of, if not the most, cited paper within graph neural networks. The architecture introduced is a convolutional network for graphs, which is derived from results on approximations of spectral graph convolutions. Its usefulness was proved with experimental results and an improved time bound compared to previous research. The paper also presented a new approach for semi-supervised learning on graphs. For a graph, an example task is node classification. In many cases, labels may only be available for a fraction of the nodes, and their new architectures use the graph structure to include the information from the unlabeled nodes in the training of the model.

The context of the paper is to consider a graph $G = (V, E)$ with N nodes where the graph structure is described by its adjacency matrix A. One example of such a network is a road network where each node represents a paper, and the undirected edges correspond to one of the papers citing the other one. Each of the nodes has C features that describe it. Examples of node features are the authors, what subjects they publish in, and the paper's number of references. Some of the nodes in the graph also have a label of what subject they belong to. The task considered by the GCN paper is to classify each of the papers (nodes) into different subjects.

Graph convolutional network is based on the same principles as convolutional networks with a local computation for each node in the network. A layer of a graph convolution has a weight matrix W and the same weights are used in the computation for each node. The computation at a particular node v is essentially a mean of the node's and its 1-degree neighbors' features weighted by W. Let each node, v, have a feature representation h_v the the next representation for node v is computed

as

$$h_v := \sigma\left(W \cdot \sum_{u \in \mathcal{N}(v)} \frac{h_u}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} + B \cdot h_v\right), \tag{3.5}$$

where $\mathcal{N}(v)$ is the set of neighbors to V and $|\mathcal{N}(v)|$ is the degree of v. The term B is the weight term for the node v itself. The activation function σ is arbitrary, and in the experiments of the paper, ReLU was used [7].

A prominent feature of the GCN architecture is its time complexity that is $\mathcal{O}(|\mathcal{E}|FC)$ where $|\mathcal{E}|$ is the number of edges, C is the number of input features and F is the number of output features. Further, the design of the architecture has a message-passing property. In the first layer of a graph convolutional network, the resulting features of each node will contain information from its first-degree neighbors. In the second layer, however, the same computation will give information on the second-degree neighbors of a particular node. This can easily be seen in (3.5) if you consider that in the second layer, each h_u will have information from its first-degree neighbors.

3.3.2 Graph attention networks (GAT)

Graph attention networks (GATs) are a type of graph neural network that was introduced by Velickovic et al. [8]. Like GCNs [7], GATs are designed to operate on graphs and learn representations of nodes in the graph that can be used for various tasks such as node classification, link prediction, and graph clustering.

The key innovation of GATs is the introduction of attention mechanisms in the graph neural network. The attention mechanism assigns a weight to each neighbor based on the similarity between the features of the center node and the neighbor node. The authors cite the transformer model by Vaswani et. al. [20] as an inspiration for the GAT architecture.

The computation in one layer of a graph attention network can be expressed as follows. Let $h_v, h_u \in \mathbb{R}^{F''}$ be the feature representations of node v and u respectively, and $\mathbf{W}$ be the weight matrix for the linear transformation applied to every node. To every pair of connected nodes $(u, v) \in E$ an attention coefficient is computed with $a : \mathbb{R}^{F''} \times \mathbb{R}^{F''} \to \mathbb{R}$ where F'' is the number of features:

$$e_{uv} = a\left(\mathbf{W}h_u, \mathbf{W}h_v\right).$$

We normalize the coefficients with the softmax function that takes as input a real-valued vector and outputs a probability distribution.

Definition 3.3.1 (Softmax Function)**.** The softmax function $\sigma : \mathbb{R}^K \to (0,1)^K$ is defined as follows:

$$\sigma(\mathbf{x})_j = \frac{\exp(x_j)}{\sum_{k=1}^{K} \exp(x_k)} \quad \text{for } j = 1, \ldots, K \text{ and } \mathbf{x} = (x_1, \ldots, x_K) \in \mathbb{R}^K$$

where exp is the exponential function, x_j is the j-th element of the input vector $\mathbf{x}$, and $\sigma(\mathbf{x})_j$ is the j-th element of the output vector.

The computation in our example is as follows:

$$\alpha_{uv} = \text{softmax}_v\left(e_{uv}\right) = \frac{\exp\left(e_{uv}\right)}{\sum_{k\in\mathcal{N}_u}\exp\left(e_{uk}\right)},$$

where $\mathcal{N}_u$ is the set of first-degree neighbors to u. The next layer's hidden node representation is then defined as

$$h'_v = \sigma\left(\sum_{u\in\mathcal{N}(v)} a_{vu} W h_u\right), \tag{3.6}$$

where $\mathcal{N}(v)$ is the set of first-degree neighbors of node v, and a_{vu} is the attention weight assigned to neighbor u by the attention mechanism. The attention mechanism was in GATs defined as a layer of a feed-forward network with the LeakyReLU nonlinearity, defined below.

Definition 3.3.2 (LeakyReLU)**.** The Leaky Rectified Linear Unit (ReLU) is defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ Cx & \text{otherwise} \end{cases},$$

where C is a fixed constant.

GATs have been shown to outperform GCNs on several benchmark datasets for node classification and link prediction tasks and have become a popular choice for graph representation learning.

3.4 Reinforcement learning

Similar to supervised and unsupervised learning, reinforcement learning is a learning paradigm. Sutton and Barto [28] describe reinforcement learning as learning a mapping from a state to actions to maximize a reward function. Reinforcement learning has been successfully applied to different games to exceed human performance, with famous examples being AlphaZero and MuZero [29], [30].

3.4.1 General framework

Reinforcement learning is a machine learning approach that has gained in popularity in recent years for its success in solving complex decision-making problems [29]–[31]. In reinforcement learning, an agent learns to make decisions by interacting with an environment to maximize a cumulative reward signal.

Algorithms in reinforcement learning focus on how an agent can learn to make decisions in an environment by maximizing a cumulative reward. In this context, an agent is an entity that interacts with an environment, which can be thought of as a dynamic system that provides the agent with observations and possible rewards in response to the agent's actions. The agent's goal is to learn a policy that maps observations to actions that maximize the expected cumulative reward over time.

A policy maps states to probabilities of selecting actions, such that $\pi(a|s)$ is the probability of choosing action a in state s if the agent follows policy π.

The reinforcement learning problem can be described, based on Sutton and Barton [28], as follows. At each time step t, the agent observes the current state of the environment s_t, selects an action a_t based on the current policy π and receives a reward r_{t+1} from the environment. The environment then transitions to a new state s_{t+1}, and the process repeats. The goal of the agent is to learn a policy that maximizes the expected cumulative reward. The cumulative reward is calculated over an episode, a sequence of states, actions, and rewards ending in a terminal state.

The expected cumulative reward, or return, over an episode is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $\gamma \in [0, 1]$ is a discount factor that determines the importance of future rewards. In other words, the agent aims to maximize the expected sum of discounted rewards over time, i.e., $V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]$ usually referred to as the value.

Formally, let S denote the set of possible states, A denote the set of possible actions, and R denote the set of possible rewards. The agent then interacts with the environment according to the following dynamics:

1. At each time step t, the agent observes the current state $s_t \in S$ of the environment. The agent selects an action $a_t \in A$ based on the current policy $\pi(a_t|s_t)$, which is a probability distribution over actions given the current state.
2. The environment transitions to a new state $s_{t+1} \in S$ with probability $P(s_{t+1}|s_t, a_t)$, which is the probability of transitioning to state s_{t+1} given the current state s_t and action a_t.
3. The agent receives a reward $r_{t+1} \in R$ from the environment, which is a function of the current state s_t and action a_t, as well as the new state s_{t+1}, i.e., $r_{t+1} = R(s_t, a_t, s_{t+1})$.
4. The process repeats from the new state s_{t+1}.

One division of reinforcement learning algorithms is into model-based and model-free [28]. Model-based algorithms explicitly model the dynamics of the environment, i.e., the transition probabilities and reward function, and use this model to compute the optimal policy. Model-free algorithms instead learn the optimal policy directly from experience without modeling the environment with an explicit function.

3.4.2 Policy gradient methods

Reinforcement learning algorithms can also be distinguished by what they are optimizing for: the value $V_\pi(s)$ or the policy directly. Depending on the application, it can be of different importance to accurately estimate the resulting value. Policy

Gradient Methods, as proposed by Sutton et al. [32], form a significant part of reinforcement learning algorithms. These methods are designed to optimize the policies directly rather than determining the value function first. The primary focus of these methods is to improve the policy's performance by adjusting the policy parameters in the direction of increasing returns.

The policy gradient theorem, as introduced by Sutton et al., provides the foundation for these methods. The theorem states that the gradient of the expected return, with respect to the policy parameters, is equal to the expected value of the gradient of the log policy times the return.

Theorem 3.4.1 (Policy gradient theorem)**.** *For any Markov Decision Process, it holds that*

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a),$$

where ρ is the expected return under policy π, θ represents the policy parameters, $d^\pi(s)$ is the stationary distribution of states under policy π, $\pi(s,a)$ is the probability of taking action a in state s under policy π, and $Q^\pi(s,a)$ is the action-value function under π, which represents the expected return for taking action a in state s.

This theorem allows estimations of the gradient by sampling trajectories under the current policy, which can then be used to update the policy parameters.

The policy gradient methods can be categorized into two types: actor-only and actor-critic methods. The actor-only methods, such as REINFORCE, introduced by Sutton et al. [32] solely rely on the policy's performance to update the policy parameters. However, these methods often suffer from high variance, leading to unstable learning. On the other hand, the actor-critic methods, such as Advantage Actor-Critic (A2C)[33], incorporate a value function (the critic) to reduce the variance of the gradient estimate. The critic estimates the value of each state, which is then used to update the policy parameters (the actor).

In conclusion, policy gradient methods offer a direct approach to optimizing policies in reinforcement learning. They provide a solid theoretical foundation and practical algorithms for learning optimal policies. However, they also present challenges, such as high variance and slow convergence, which are active areas of research.

3.4.3 MuZero

The MuZero model, introduced by Schrittwieser et al. [30] combined ideas from both model-based and model-free learning. The model they developed only models the parts of the environment that are relevant to the next actions. The MuZero model was shown to learn to play complex games using a combination of deep neural networks and Monte Carlo tree search. The algorithm was designed to work without any prior knowledge or assumptions about the game being played, which makes it highly versatile and adaptable to a wide range of tasks. Unlike previous models such as AlphaZero [29] MuZero did not require explicit programming of the rules of the game, which was a step towards more generalizable algorithms.

The algorithm [30] consists of four main components: a representation learning module, a dynamics function, a prediction function, and a search algorithm. These components work together to allow the algorithm to learn a model of the environment and use it to make decisions in real-time.

The representation learning module is responsible for encoding the state of the game into a fixed-length vector that can be used by the other components of the algorithm. This module is trained using a combination of supervised learning and self-play, where the algorithm plays against itself and learns from its own experience.

The dynamics function takes the current state of the game and action as input and predicts the next state of the game. This function is also learned using a combination of supervised learning and self-play. Self-play is when the model plays any game with a version of itself as the opponent for the purpose of learning.

The prediction function takes the current state of the game as an input and predicts the value of the game (i.e., the probability of winning) and the policy at that state. This function is also learned using a combination of supervised learning and self-play.

Finally, the search algorithm is responsible for selecting the best action to take at each stage of the game. This algorithm uses a combination of Monte Carlo tree search and the predictions from the prediction function to explore different possible actions and evaluate their outcomes.

During training, the MuZero algorithm alternates between two phases: the prediction phase and the search phase. In the prediction phase, the algorithm uses the representation learning module, dynamics function, and prediction function to predict the value and policy of the game at a given state. In the search phase, the algorithm uses the search algorithm to select the best action to take at that state.

The MuZero algorithm has been shown to achieve state-of-the-art performance in a variety of games, including chess, shogi, and Go [30]. The algorithm is also able to learn to play new games without any prior knowledge. In this thesis, we apply it to the task of route planning.

3.5 Previous research in deep learning for vehicle routing problems

Previous research, such as Li et al. [3] has shown promising results applying deep learning to obtain approximate solutions to both TSP and CVRP. A common approach in deep learning for solving graph problems in combinatorial optimization is to use a transformer or graph neural network that produces an embedding, vector representation, of each node [3], [4]. The architecture that produces the embedding is often referred to as an encoder, from the terminology in natural language processing, because it encodes the graph to a vector space. From the encoding, routes can then be decoded with some scheme typically referred to as a decoder.

Compared to state-of-the-art heuristic methods, deep learning, in particular, offers improved running times [9]. In solving combinatorial optimization problems, there

is generally a trade-off between optimality and solution speed. The trade-off is exemplified in Li et al. [3] for the CVRP problem, where a heuristic solver achieves optimal or near-optimal solutions but with running times of several minutes for small instances. For the same instances, deep learning models, however, require less than a second with the drawback of a larger optimality gap [3].

Recent research [3], [4] uses attention-based deep learning models with a decoding strategy to choose routes and achieve improved results on benchmarks such as CVRPLib. To decode solutions, Lei et al. [4] introduce two strategies: greedy decoding and a sampling method. The greedy strategy chooses the next node in the route as the node with the highest probability, according to the model. For sampling, Lei et al., instead of choosing the node with the highest probability, sample the next node according to the probability distribution the model outputs. During testing, they sample multiple solutions and report the best result. The sampling method outperforms the greedy method on the CVRP with half the optimality gap on VRP100 instances but requires several hours of running time [4]. Commonly, for these previous methods, the route decoding happens once, and there is no possibility to adjust previous mistakes for a specific instance. We are particularly interested in developing methods for this purpose.

4

Methods

4.1 Overview

We built a model trained on simulated instances of routing problems. Our model consists of an encoder-decoder scheme that embeds the graph instance and a set of vehicles with capacity constraints. From the encoder, a graph embedding is produced with a real-valued vector for each node in the graph that represents the node's features and its relation to its neighbors. The decoded routes are iteratively produced from the graph embedding. The model was trained using reinforcement learning on simulated data. Our experiments were performed on the capacitated and heterogeneous capacitated vehicle problem but could be extended to other routing problems with little modifications. We model the CVRP problem by a complete graph $G = (X, d)$ where X is the set of nodes, and as in definition 2.4.2, d is a mapping of the demands for each node. Each node $j \in X$ has a position P_j in 2D-space. The set of edges are all pairs $u, v \in X$ such that $u \neq v$ and the edge costs are the Euclidean distances between each pair of nodes. The set of vehicles, V, has capacities Q such that vehicle V_i has capacity Q_i. We also define a depot node in $x_d \in X$ with $c(x_d) =)0$ where each vehicle starts its route. During training, we fixed the number of vehicles to 3 with the possibility of returning to the depot and refilling the capacity.

4.2 Base model

To promote the comparability of our results, we used Li et al. [3] model implementation as a base model for our experiments. The graph encoder is an attention model that uses multi-head attention to process the sequence of nodes and produce a node embedding for each node. The features that are input to the graph encoder for each node are the location in the plane and the capacity of each available vehicle. The graph embedding is computed once for each graph and re-used during the encoding of routes.

4.2.1 Graph encoder

The graph encoder in the base model is an attention network that encodes the graph as a sequence of nodes with each node's location and demands. The main

idea of the model is to leverage the attention mechanism to capture the dependencies between nodes in a graph, which can be of arbitrary distances apart. The multi-head-attention operation is the core of the model. In the context of graph-structured data, it allows the model to focus on different parts of the graph when generating the node embeddings. The multi-head-attention operation is defined as follows:

$$\begin{aligned} Q &= W_q \cdot q \\ K &= W_k \cdot h \\ V &= W_v \cdot h \\ \text{compatibility} &= \frac{QK^T}{\sqrt{d_k}} \\ \text{attn} &= \text{softmax}(\text{compatibility}) \\ \text{out} &= \text{attn} \cdot V, \end{aligned}$$

where q is the query, h is the input data, W_q, W_k, and W_v are the weight matrices for the query, key, and value, respectively, and d_k is the dimension of the key. The softmax function is applied to the compatibility scores to obtain the attention weights, which are then used to compute a weighted sum of the values. There is also a skip connection at the output, which is a simple addition of the input to the multi-head attention operation. The operation output is further normalized to have zero mean and unit variance.

The output of the Graph Attention Encoder is then a set of node embeddings, $E \in \mathbb{R}^{N \times F}$ where N is the number of nodes and F is the number of features for each node. These embeddings are then used for route decoding, and the graph encoding is only done once for each graph. The same graph encoder architecture is used in all our experiments detailed in this section.

4.2.2 Route decoder

The second part of the model is the decoder architecture, which produces the final route. The route is iteratively decoded, and in each decoding step, the model determines which vehicle and node is the next to be added to the solution [3], [4]. The route decoding algorithm takes as input the set of node embeddings and the overall decoding steps are shown as pseudo-code in Algorithm 2. The steps for the vehicle selection and node selection are detailed in this section.

Algorithm 2: Route Decoder algorithm

Result: Routes
Input: node embeddings, vehicle initial position and capacity
Initialize: route for each vehicle
while *unvisited nodes* **do**
    v = select vehicle(routes, embeddings);
    x = select node(routes, v, unvisited node embeddings);
end
return *routes*

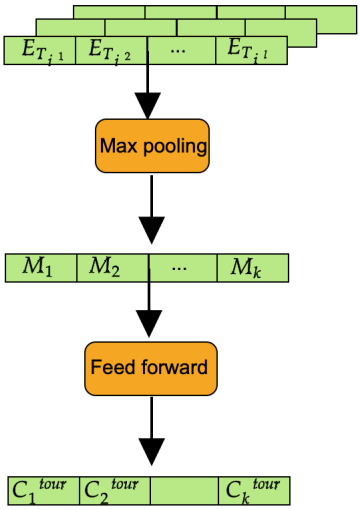

Figure 4.1: Figure of computation of a mean tour as context for vehicle selection. For each vehicle i, the partial tour is represented by the node embedding for each node j: $E_{T_j^i}$. The embeddings are concatenated, and through max-pooling and a feed-forward network, a representation for each of the k vehicles is formed: C_i^{tour}.

We implement decoding with the same principle as Li et al. [3]. Each part of the route is decoded by choosing the vehicle, V_i, and then the node, n, to visit. The nodes that have too high demand or have already been visited are masked such that they have zero probability.

The vehicle selection process begins by computing a mean tour (see Figure 4.1) for each vehicle, encapsulating the tours chosen so far. Specifically, for partial tours T_i for vehicle i we compute

$$M_i = \max_{j \in 1,\dots,l} \{[E_{T_i^1}; E_{T_i^2}; E_{T_i^3}; E_{T_i^4} ... E_{T_i^l}]\}, \tag{4.1}$$

where $E_{T_i^j}$ refers to the node embedding corresponding to node j in tour i and a max-pooling operation is applied to the concatenation of the embeddings yielding

a vector for each tour such that $M_i \in \mathbb{R}^F$. Further, the current position for each vehicle, V_i, as well as its capacity, is concatenated and passed through a feed-forward network, θ_{veh} to form a vehicle context such that

$$C_i^{veh} = \theta^{veh}([V_i^x; V_i^y; Q_i]/S_i), \tag{4.2}$$

where V_i^x and V_i^y are the position coordinates for vehicle i, Q_i its capacity and S_i the speed of the vehicle. These features, derived from the routes and each vehicle, are then concatenated to form a context for each vehicle

$$C_i = [C_i^{veh}; \theta^{route}(M_i)] \tag{4.3}$$

To generate probabilities for each vehicle, we apply a feed-forward network to the concatenated features $C = [C_1; C_2...; C_k]$:

$$P = \text{softmax}(\theta^{prob}(C)), \tag{4.4}$$

where θ^{prob} represents the parameters of the feed-forward network that maps the concatenated features to a probability distribution, $P \in \mathbb{R}^k$. The softmax function ensures that the probabilities sum up to 1, allowing us to interpret them as the likelihood of selecting each vehicle.

During the training phase, the vehicle is selected based on a multinomial distribution derived from the computed probabilities, that is

$$v_{l+1} \sim \text{Multinomial}(P_1, P_2, ..., P_k), \tag{4.5}$$

where $P_1, P_2, ..., P_k$ are the probabilities assigned to each vehicle and v_{l+1} is the vehicle chosen in step $l + 1$.

After a vehicle has been selected, the next node is selected. The node selection employs a multi-head attention mechanism to compute the probabilities of selecting the next node in the route. The function operates on fixed and state-dependent inputs: node embeddings, context node projections, and vehicle-specific information.

First, fixed graph embeddings are computed to be re-used in every node selection for an instance. The graph embedding is computed with a linear layer θ_{graph} applied to the mean of the node embeddings E:

$$C_{fixed} = \theta_{graph}\left(\frac{1}{N}\sum_{i=1}^{N} E_i\right). \tag{4.6}$$

From the node embeddings, three different representations for each node are also computed: the glimpse key and value, as well as the logit key. The vectors are projections of the node embeddings with distinct weights.

$$G_{key} = \theta_{G_k}(E) \tag{4.7}$$
$$G_{value} = \theta_{G_v}(E) \tag{4.8}$$
$$L_{key} = \theta_{L_k}(E), \tag{4.9}$$
$$\tag{4.10}$$

where θ_{G_k}, θ_{G_v} and θ_{L_k} correspond to the parameters of distinct linear layers.

The node selector then computes a query vector, which is a mapping of the chosen vehicle's current position embedding (node embedding of the last visited node) and the remaining capacity. A feed-forward network, θ_{query}, maps the concatenation of the embedding and remaining capacity and the fixed graph embedding is added such that:

$$q = C_{fixed} + \theta_{query}([E_{T_v^{l+1}}; Q_{remaining}], \tag{4.11}$$

where $E_{T_{l+1}^v}$ is the embedding of the node for vehicle v current position. A multi-head attention layer, MHA, is then applied to the query, glimpse key and glimpse value such that an attention-weighted context for the current node is computed by

$$G = MHA(q, G_{key}, G_{value}) = softmax\left(\frac{qG_{key}^T}{\sqrt{d_q}}\right) G_{value}, \tag{4.12}$$

where d_q is the dimension of the query. Equation (4.12) omits to reshape the glimpse key, value and query to multiple heads. The glimpse is then projected back with a linear layer, θ_{proj}, to the original embedding dimension to update the context node embedding

$$G = \theta_{proj}(G). \tag{4.13}$$

The node selection operation then computes unnormalized log probabilities (logits) of selecting each node as the next node in the route. The logits are computed by a scaled dot product of the updated context node embedding (glimpse) and the logit keys:

$$L = \left(\frac{GL_{key}^T}{\sqrt{d_k}}\right), \tag{4.14}$$

where d_k is embedding size such that $G \in \mathbb{R}^{N \times d_k}$ for graph size N. The logit values corresponding to nodes that have already been visited or which have too high demand for the selected vehicles are set to $-\infty$ for the probability of selecting them to be zero. The node is, during training, selected according to the probability distribution of the normalized logits.

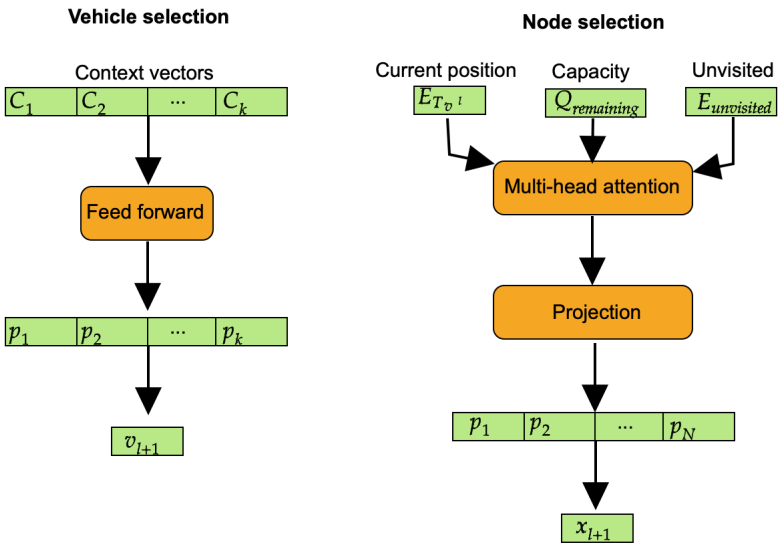

Figure 4.2: Figure of vehicle and node selection. The vehicle selection uses the accumulated route features and the last node location as input. For node selection, the current position of the chosen vehicle v_{l+1} for step $l + 1$, remaining vehicle capacity and the remaining unvisited nodes' features are used as input.

The node selection function thus combines the information from the graph, the current state of the algorithm, and the vehicle-specific information to compute the probabilities of selecting the next node in the route. A higher-level scheme of the vehicle and node selection can be seen in Figure 4.2.

4.2.3 Reinforcement learning model

Our base model uses a similar learning algorithm as Li et al. [3], which is based on the policy gradient method, incorporating a baseline network. The training algorithm is defined by two networks: the policy network and the baseline network. The policy network, π_θ, with θ denoting the parameters of the network, selects an action and generates probability vectors for both vehicles and nodes based on this action at each decoding step. The baseline network, v_Φ, has the same architecture as the policy network but uses greedy decoding of vehicles and nodes.

A Monte Carlo method is used to update the parameters, and a fixed dataset is generated for each training epoch, and routes are computed for each by both the policy and baseline network using a greedy decoding (GreedyRollout). The expected reward of the baseline network is a greedy execution of the policy. If the latest policy network significantly outperforms the baseline network, as determined by a paired t-test over the fixed dataset, the parameters of the baseline network are replaced with those of the policy network. This iterative process of updating the two networks makes the policy π_θ progressively improve to find higher-quality solutions. The pseudo-code for the training algorithm, implemented by Li et al. [3], is shown in Algorithm 3.

Algorithm 3: Training algorithm - Policy gradient method with baseline network

Result: Optimized parameters θ and ϕ

Input: Initial parameters θ for policy network π_θ; initial parameters ϕ for baseline network v_ϕ; number of iterations I; iteration size N; number of batches M; maximum training steps T (number of nodes); significance α.

for $iter = 1, 2, \ldots, I$ **do**
  Sample N problem instances randomly;
  for $i = 1, 2, \ldots, M$ **do**
    Retrieve batch $b = N_i$;
    for $t = 0, 1, \ldots, T$ **do**
      Pick an action $a_{t,b} \sim \pi_\theta(a_{t,b}|s_{t,b})$;
      Observe reward $r_{t,b}$ and next state $s_{t+1,b}$;
    end
    $R_b =$
     $-\max($GreedyRollout with baseline v_ϕ and compute its reward $R_{BL}; B_b)$;
    $d\theta \leftarrow \frac{1}{B_b}(R_b - R_{BL})\nabla_\theta \log \pi_\theta(s_{T,b}|s_{0,b})$;
    $\theta \leftarrow \text{Adam}(\theta, d\theta)$;
  end
  if $ONESIDEDPAIREDTTEST(\pi_\theta, v_\phi) < \alpha$ **then**
    $\phi \leftarrow \theta$;
  end
end

4.3 Data simulation for CVRP

The simulation approach for CVRP instances is based on the code provided by Li et al. [3] at `https://github.com/Demon0312/HCVRP_DRL`. We generate training data by sampling nodes uniformly from the unit square. The depot node is also randomly sampled, and the demands are integers in the $[1, 10]$ range. The edge weights are the Euclidean distances between each of the nodes. We use the same vehicle capacities and speeds as Li et al. [3] for comparability.

4.4 Iterative route decoding

The base model used greedy decoding of routes for inference, which means that the following action with the highest output probability is chosen. We implemented a method for conditioning on previously obtained vehicle and node selection solutions. We assume that the Markov property holds when designing the iterative method. Specifically, we assume that all information on the previous vehicle route is contained within the final routes. We implemented a method for using the previous routes to improve routes iteratively. The core idea behind iterative route decoding is that we *plan* the routes for a specific graph *multiple* times with context from the *previous*

routes. The modified structure of the route decoding can be seen in Algorithm 4.

Algorithm 4: Iterative route decoder algorithm with I iterations and previous route conditioning

Result: Routes
Input: node embeddings, vehicle initial position and capacity, number of iterations I
Initialize: route for each vehicle, previous routes
for *i=1 to I* **do**
  while *unvisited nodes* **do**
    v = select vehicle(routes, embeddings, previous routes);
    x = select node(routes, v, unvisited node embeddings, previous routes);
    update previous routes;
  end
end
return *routes*

The following sections describe how we condition on the previously planned routes.

4.4.1 Iterative vehicle selection

The vehicle selection process is enhanced by incorporating information from previous routes, which is achieved by encoding the previous routes into the model. Similarly, as in the vehicle selection described in Section 4.2.2, we compute a mean tour with the previous routes to encode them as a context. We also compute the cost of each node decision in the previous tours, and each tour cost is concatenated with the corresponding node cost and the vehicle's current location. The node costs measure how much the decision to visit each node contributed to the total cost of the tour. For a particular node T_i^j indexed by j in tour T_i corresponding to vehicle i the node cost is given by:

$$c_{T_i^j} = |P_{T_i^{j+1}} - P_{T_i^j}|_2 / S_i, \tag{4.15}$$

where $||_2$ refers to the euclidean distance and S_i is the speed of vehicle i. The previous tour embeddings are concatenated with the node costs and current positions of the corresponding vehicle,

$$\begin{aligned}\tilde{M}_i^{prev} =& [E_{T_i^1}\ c_{T_i^1}\ v_i^x\ V_i^y];\\ & [E_{T_i^2}\ c_{T_i^2}\ V_i^x\ V_i^y];\\ & .\\ & .\\ & .\\ & [E_{T_i^l}\ c_{T_i^l}\ V_i^x\ V_i^y].\end{aligned}$$

The resulting tensor for each vehicle is then encoded with a small attention network, θ_{prev}. The attention encoder takes as input the concatenated embeddings of the

nodes in the previous tours and outputs a single vector that represents the attention over the nodes in the tours. The encoder consists of two linear layers. The first layer transforms the input embeddings into a higher-dimensional space, and the second layer computes a score for each node in the tour. A softmax function is then applied to the scores, which normalizes them into a probability distribution. This distribution represents the attention that the model pays to each node in the tour when computing the mean tour. The attention scores are then used to weigh the node embeddings. Each embedding is multiplied by its corresponding attention score, which gives more importance to nodes with higher scores. The weighted embeddings are then summed to produce a single vector that represents the previous mean tour:

$$M_i^{prev} = \theta_{prev}(\tilde{M_i}^{prev}), \tag{4.16}$$

and concatenated together with the mean tour of the current iteration

$$M_i^{all} = [M_i^{prev}; M_i]. \tag{4.17}$$

The remaining vehicle selection process is then the same as described in Section 4.2.2. In summary, the encoder, θ_{prev}, use an attention mechanism to weigh the importance of each node in the previous tours when computing the mean tour. This allows the model to focus on the most relevant nodes, potentially improving the efficiency and effectiveness of the vehicle selection process.

4.4.2 Iterative node selection

We also use the computed previous mean tour for the node selection and pass it through two feedforward layers that project the mean tours to a lower dimension. The previous tour information is then concatenated in the node selection step.

The node selection is designed to handle two scenarios: when there are no previous routes and when there are previous routes. In the absence of previous routes, the computation of logits over the nodes is computed as in Equation (4.14).

When previous routes are available, the computation of logits is slightly modified. The previous routes, represented by the vector $\mathbf{C^{prev}}$ (tour context), are concatenated with the glimpse G and the logit key L_{key} along the last dimension. This incorporates the information from previous routes into the computation of logits after the computation of glimpse, G in Equation (4.13). The modified computation is as follows:

$$\begin{aligned} \tilde{G} &= [G; C^{prev}] \\ \mathbf{C^{prev}_{expanded}} &= \text{repeat}(C^{prev}, N) \\ \tilde{L_{key}} &= [L_{key}; \mathbf{C^{prev}_{expanded}}] \end{aligned} \tag{4.18}$$

Here, the semicolon denotes the concatenation operation. The repeat function replicates the tour context vector to match the third dimension of the key matrix. The logits are then computed in the same way as in the case without previous routes; see Equation (4.14).

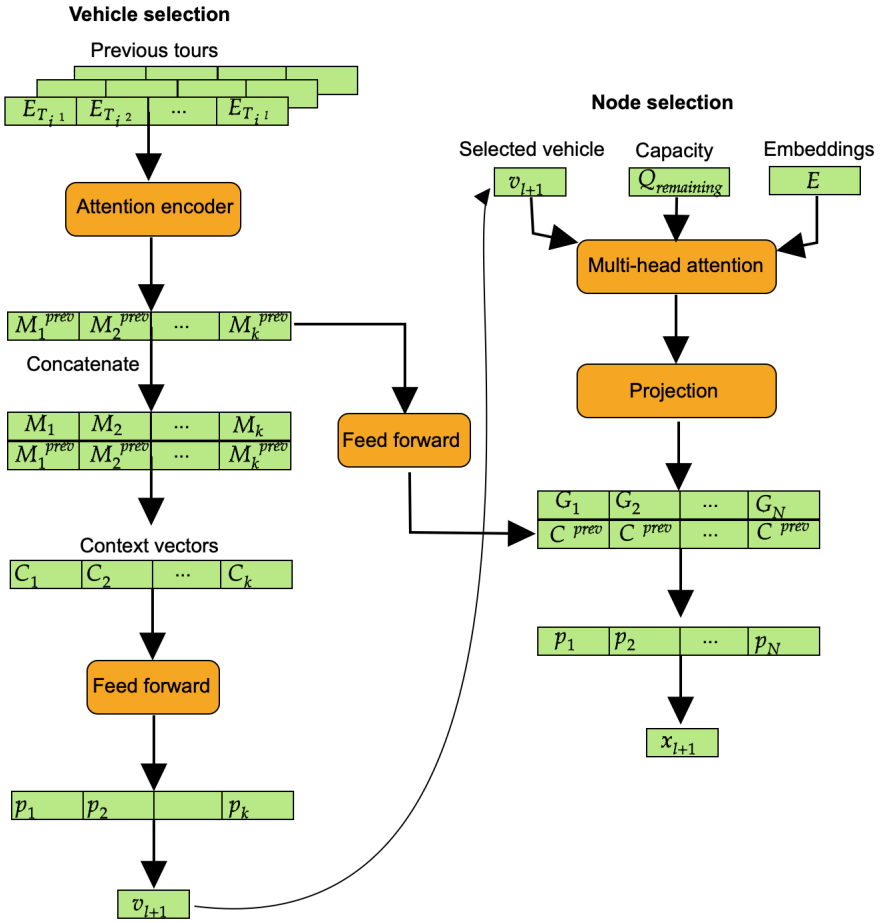

Figure 4.3: Figure of vehicle and node selection with context from previous routes added. The vehicle selection uses the accumulated route features and previous routes as input. For node selection, the current position of the chosen vehicle v_{l+1} for step $l+1$, remaining vehicle capacity and the remaining unvisited nodes' features are used as input. A previous context is also concatenated with the node context in the computation of the logits $p_1, ..p_N$.

The effect of this implementation is that the model can leverage historical route information when available to make more informed decisions when selecting a node. This could lead to more efficient routes and improved performance of the VRP solution. The method for adding context is further visualized in Figure 4.3.

4.4.3 Baseline iteration method for route decoding

The process began with the development of a preliminary solution, which served as a baseline for the subsequent stages of the study. This initial solution was designed to provide a rudimentary model for iterative route planning.

The baseline solution was then subjected to a series of tests to evaluate its performance. These tests were not intended to produce definitive results but rather to provide a benchmark against the base model. The results of these tests were used to identify areas of potential improvement and to guide the development of the improved solution.

The base solution was very similar to the iterative vehicle and node selection but did not use an attention operation for the previous route and did not take into account the node costs or current position when processing the previous tours. Instead, for vehicle selection, the previous mean tour was computed with max-pooling and concatenated with the mean tour, and for node selection, respectively, the previous mean tour was concatenated with the queries and keys similar to Equation (4.18).

4.5 Iterative route decoding with gradient updates

In this approach, we leveraged transfer learning by starting from a pre-trained baseline model. The idea is to fine-tune the model on a single instance, allowing it to adapt to the specific characteristics of the instance. This method can be particularly effective when the instance has unique features that were not adequately represented in the original training data. The two key differences from the iterative route decoding described in Section 4.4 are 1) this method is for optimizing a single instance, and 2) gradient updates are performed between iterative route decoding steps.

The process involves performing gradient updates on the pre-trained model using a single instance. This fine-tuning process allows the model to adjust its parameters to fit the instance better, potentially leading to improved performance. The updated training algorithm which incorporates this idea can be seen in Algorithm 5.

Algorithm 5: Training algorithm - Policy gradient method with baseline network for single instance training

Result: Optimized parameters θ and ϕ
Input: pre-trained parameters θ for policy network π_θ; pre-trained parameters ϕ for baseline network v_ϕ; number of iterations I; maximum training steps T;
for $iter = 1, 2, \ldots, I$ **do**
  Retrieve instance b;
  for $t = 0, 1, \ldots, T$ **do**
    Pick an action $a_{t,b} \sim \pi_\theta(a_{t,b}|s_{t,b})$;
    Observe reward $r_{t,b}$ and next state $s_{t+1,b}$;
  end
  $R_b =$
  $-\max($GreedyRollout with baseline v_ϕ and compute its reward $R_{BL}; B_b)$;
  $d\theta \leftarrow \frac{1}{B_b}(R_b - R_{BL})\nabla_\theta \log \pi_\theta(s_{T,b}|s_{0,b})$;
  $\theta \leftarrow \text{Adam}(\theta, d\theta)$;
  if $\pi_\theta < v_\phi$ **then**
    $\phi \leftarrow \theta$;
  end
end

We extended Algorithm 5 further to include conditioning on previous routes such that the model not only improves from the weights updates but also uses explicit information from the previous results. When the method for vehicle and node selection described in Section 4.4 are further with the gradient updates, the resulting reinforcement learning algorithm is slightly modified as seen in Algorithm 6.

Algorithm 6: Training algorithm - Policy gradient method with baseline network for single instance training

Result: Optimized parameters θ and ϕ

Input: pre-trained parameters θ for iterative policy network π_θ; pre-trained parameters ϕ for iterative baseline network v_ϕ; number of iterations I; maximum training steps T; Initialize state S_0 = None;

for $i = 1, 2, \ldots, I$ **do**
  Retrieve instance b;
  for $t = 0, 1, \ldots, T$ **do**
    Pick an action $a_{t,b} \sim \pi_\theta(a_{t,b}|s_{t,b}, S_{i-1})$;
    Observe reward $r_{t,b}$ and next state $s_{t+1,b}$;
  end
  Save state $S_i = s_{T,b}$;
  $R_b =$
  $-\max($GreedyRollout with baseline v_ϕ and compute its reward $R_{BL}; B_b)$;
  $d\theta \leftarrow \frac{1}{B_b}(R_b - R_{BL})\nabla_\theta \log \pi_\theta(s_{T,b}|s_{0,b})$;
  $\theta \leftarrow \text{Adam}(\theta, d\theta)$;
  if $\pi_\theta < v_\phi$ **then**
    $\phi \leftarrow \theta$;
  end
end

4.6 MuZero implementation

We implemented MuZero for the task of solving CVRP with a game-based approach. Due to training and inference costs, we realized that this would not be a feasible approach for large-scale route planning, and the model was not used in our final results. For future reference, we describe our modeling approach here to give insight into the challenges this model posed.

4.6.1 Modeling CVRP as a game

According to the framework for reinforcement learning introduced in Section 3.4.1 we model the CVRP problem as a single-player game; that is, the goal of the game is to maximize the expected total reward obtained from the environment. At times t, the state, S_t of the game is the set of nodes V as well as the nodes $\{v_1, ..., v_{t-1}\}$ divided into K different vehicle routes $\{V_1, .., V_K\}$ such that $v_i \in V_k$ for all $i \in \{1, ..., t-1\}$ and $k \in \{1, ..., K\}$. Each V_k signifies a vehicle with a specified capacity C_k. Each node v_i has a demand d_i and coordinates (x_{v_i}, y_{v_i}) in a 2D-space.

The cumulative reward of the game is defined as the sum of all Euclidean distances between consecutive nodes in each route. We define the reward function r_t as follows:

$$r_t = \begin{cases} -\sum_{k=1}^{K} \sum_{i=1}^{|V_k|-1} |(x_{v_i}, y_{v_i}) - (x_{v_{i+1}}, y_{v_{i+1}})|_2, & \text{if } t = |V| \\ 0 \text{ else} \end{cases}$$

where $|\cdot|_2$ denotes the Euclidean distance between two points in 2D-space. In reinforcement learning, it is general practice to maximize the expected reward, but CVRP aims to minimize the total distance. As a mathematical trick, we„ add a negative sign in the reward to turn the minimization problem into a maximization problem.

The action space $\mathcal{A}_t$ at time t consists of adding a new node to an existing route. For each vehicle k, the available actions at time t are to add any node v_t to a route V_k for which $C_k \geq d_t$ and for v_t which has not already been visited.

The state transition function f takes the current state S_t and the chosen action a_t as inputs and returns the new state S_{t+1}. If the chosen action is to add node v_t to route V_k, the new state S_{t+1} is obtained by adding v_t to V_k and updating the capacity and total length of V_k accordingly. The end of an episode is defined by all nodes being visited, $t = |V|$ and each node can only be visited if its demand can be satisfied. In practice, we implement a penalty in case the action space is empty in any time step before $t = |V|$.

The CVRP can be formulated as a Markov Decision Process (MDP) where the objective is to maximize the cumulative reward over a finite time horizon. In this case, the horizon corresponds to the number of nodes in the input graph. The MuZero algorithm can then be used to learn a policy and value function for the CVRP MDP, which can be used to generate approximate solutions to the problem.

4.6.2 Graph attention network for heterogenous graphs

The basic building block of the graph network we implemented was a Heterogeneous Graph Attention Network (HeteroGAT), which is an extension of the Graph Attention Network (GAT) [8] designed to handle heterogeneous graphs for routing problems. In the context of the Capacitated Vehicle Routing Problem (CVRP), the nodes represent customers and vehicles, and the edges represent the connections between them.

The HeteroGAT architecture consists of two main components: the GAT layers and the forward function. The GAT layers are responsible for learning the node embeddings, while the forward function is responsible for applying these layers to the graph data.

The GAT layers are multi-head attention mechanisms that compute the node embeddings based on their neighborhood information. The number of heads in each layer is a hyperparameter that can be tuned for different tasks. In the HeteroGAT architecture, there are two GAT layers for each edge type: one for the connections between customers and one for the connections between vehicles and customers. Each layer takes as input the node features and the edge indices and outputs the updated node features.

The forward function applies the GAT layers to the graph data. It first retrieves the node features and edge indices for each edge type. Then, it applies the corresponding GAT layer to each node type and edge type. The output node features are then

concatenated and returned.

Pseudo-code for the heterogeneous graph attention network can be seen in Algorithm 7.

Algorithm 7: Pseudo-code for implementation of heterogenous GAT for route decoding.

Result: Updated node features
Input: Graph data
nodes, edges ← retrieveFeatures(GraphData);
CustomerNodeFeatures ← GATLayer(CustomerNodeFeatures,
customer-customer edges);
allNodeFeatures ← Concatenate CustomerNodeFeatures and
VehicleNodeFeatures;
allNodeFeatures ← GATLayer(allNodeFeatures, customer-vehicle edges);
customerVehicleFeatures, vehicleFeatures ← split(allNodeFeatures);
customerNodeFeatures ← customerNodeFeatures + customerVehicleFeatures;
return *updatedNodeFeatures*;

4.6.3 MuZero architecture

The implementation of the MuZero algorithm for vehicle routing problems is based on a Graph Neural Network (GNN) architecture. The GNN is composed of three main components: the representation network, the dynamics network, and the prediction network.

The representation network is responsible for encoding the raw observations into a latent state representation. It uses a Graph Attention Network (GAT) to process the input graph, which is composed of vehicle and customer nodes. The representation network can optionally downsample the input graph using another GAT before processing it if multiple previous observations are used during training or inference. The output of the representation network is a latent state representation of the input graph, which is normalized to the range [0, 1] for stability during training.

The dynamics network predicts the next latent state and the immediate reward given the current latent state and an action. The action is a pair of indices indicating the vehicle and the customer node. The action is one-hot encoded and concatenated to the features of the corresponding nodes in the input graph. The updated graph is then processed by the dynamics networks consisting of two HeteroGAT layers to produce the next latent state. The immediate reward is predicted by a multi-layer perceptron (MLP) that takes the mean features of the vehicle and customer nodes as input. The output of the dynamics network is the next latent state, which is also normalized to the range [0, 1] and the immediate reward.

The prediction network predicts the policy and value functions given the current latent state. The policy function is a distribution over the possible actions, that is, the combination of nodes and vehicles, and the value function estimates the expected

return from the current state. The policy is predicted by an MLP that takes the concatenated features of each vehicle-customer pair as input. The value is predicted by another MLP that takes the maximum features of all vehicle-customer pairs as input.

These networks are wrapped in a MuZero network, providing the initial and recurrent inferences interface. The initial inference takes a raw observation as input and returns the initial latent state, policy, value, and a dummy reward. The dummy reward is a placeholder, as there is no reward associated with the initial observation. The recurrent inference takes a latent state and an action as input and returns the next latent state, reward, policy, and value.

4.6.4 Training algorithm

The training algorithm used in this implementation is the same as the one described in the original MuZero paper [30]. The training process involves two main steps: self-play data generation and network training.

In the self-play step, the current network is used to play games against itself and in the case of a single-player game, this is a process of generating episodes with the agent informing what actions to take. Each game starts with an initial state, the depot node, and proceeds by repeatedly selecting the next node to visit using Monte Carlo Tree Search (MCTS), which uses the current network to guide its search. The MCTS returns a policy that represents the probability of selecting each action from the current state, and this policy is used to select the next action. The game continues until a terminal state is reached; in our case, this is when all nodes have been visited. The rewards during the game follow the definition in Section 4.6.1. The sequence of states, actions, policies, and rewards encountered during the game is stored as a trajectory.

In the network training step, the stored trajectories are used to update the network parameters. Each trajectory is processed in reverse order, starting from the terminal state and moving backwards to the initial state. For each state in the trajectory, the target for the value function is computed as the sum of the discounted rewards from the current state to the terminal state. The targets for the policy function are the policies returned by the MCTS. The network parameters are updated by minimizing a loss function that measures the difference between the predicted and target values and policies. The training process alternates between these two steps. For the training step, we use the open implementation of self-play and multi-GPU training for MuZero by Duvaud et al. [34].

4.7 Evaluation

Both traditional heuristic methods and deep learning methods can be baselines for our research. In Li et al. [3] the heuristic method, SISR, is referred to as a baseline method for CVRP with regard to the optimality gap. For solution speed, on the other hand, deep learning methods are dominant. These benchmarks are important

to investigate the model's applicability to real-world problems. Further, we notice that there is a lack of overlap in the test instances in previous research papers, and therefore, we choose to evaluate on the same test sets as is used in Li et al. [3] and extend the evaluation by including selected benchmarks from CVRP-lib.

We evaluate the model on the instances in these benchmarks and report results on the value of the solution as well as the optimality gap in the case where optimal solutions exist. We also report inference speed on different problem sizes. The evaluations are run on a single RTX3090 and the running time is averaged over several runs.

5

Results

5.1 Training settings

The training settings for the machine learning model used in this study are detailed in this section. The initial learning rate was set at 0.0001 with a learning rate decay of 0.995 every epoch. The Adam optimizer was used for the training process. The batch size was set to 512. During training, the gradients were clipped to a maximum gradient norm for clipping was set to 3.0 to stabilize training. The model was designed with 3 graph encoder layers. and the node embedding size was set to 128. This is the size of the vectors that the model uses to represent each node. These settings were determined used to mirror the settings of the base model to be able to compare results with pre-trained models.

All our models are trained on graphs of 40 nodes with one depot node and 3 available vehicles with different capacities and speeds. In each training epoch, we generate a dataset of 1280000 new instances according to the specification in Section 4.3.

5.2 Evaluation on generated test datasets

We base our model on Li et. al. and use similar settings for data generation and further test on the same test sets that are generated from the same distribution as the training to show in-distribution results as well as our results with gradient iterations.

5.3 Base solution for iterative decoding with greedy decoding

To evaluate the baseline iteration method, described in Section 4.4.3, We train three models from scratch on simulated data with the training settings in Section 5.1. The first model is the base model, then we train a model with added vehicle selection iterations, and lastly a model with both vehicle and node selection information. The iterative models were trained with 3 iterations and in total for 20 epochs. We then evaluate the models on a test data set with 40 nodes and 1280 instances using greedy decoding. The results are presented in Table 5.1.

	V3-C40	
Model	Value	Serial duration (batch size 1024)
Baseline	59.37	0.733
+ Vehicle selection (3 iterations)	59.26	0.745
+ Node selection (3 iterations)	59.15	0.751

Table 5.1: Results from training base model, base model, and vehicle selection iterations and node selection iterations with three iterations. The results are with greedy decoding and one iteration for all models on test data with the distribution as the training data with 40 nodes.

5.3.1 Evaluation of iterative method

In this section, we evaluate the performance of iterative route decoding compared to traditional methods and our base model. The model we trained is denoted "our model" and is initialized with the same graph encoder as the base model and then trained for 20 epochs. In Table 5.2 and 5.3, we report the result of our model trained for five iterations on data with 40 nodes. The base model is trained on similar data.

The test data is the same test data that Li et. al. [3] reported their results on and contains instances for the heterogenous CVRP with 40-120 nodes. Each instance has a fixed capacity for each vehicle of [20, 25, 30] and we assign the speeds of the corresponding vehicles to [1/4, 1/5, 1/6]. The objective of the model is to minimize the sum of the time traveled by all vehicles.

	V3-C40			V3-C60			V3-C80		
Model	Value	Opt. gap	Time	Value	Opt. gap	Time	Value	Opt. gap	Time
Exact-solver	55.43*	0%	(74s)	78.47*	0%	(283s)	102.42*	0%	(1047s)
SISR	55.79	0.65%	(335s)	79.12	0.83%	(631s)	103.41	0.97%	(763s)
VNS	57.54	3.81%	(145s)	81.44	3.78%	(384s)	106.18	3.67%	(722s)
ACO	60.11	8.44%	(265s)	86.05	9.66%	(398s)	113.75	11.06%	(782s)
FA	59.94	8.14%	(216s)	85.36	8.78%	(359s)	112.81	10.14%	(512s)
AM (Greedy)	66.54	20.04%	(0.64s)	91.19	16.21%	(1.10s)	117.22	14.45%	(1.33s)
Baseline (Greedy)	58.99	6.42%	0.81s	83.25	6.09%	0.90s	109.19	6.66%	0.99s
Our model (5 iterations)	58.38	5.32%	1.87s	82.70	5.39%	2.82s	108.78	6.21%	4.04s

Table 5.2: Results from the evaluation of our model trained with 5 iterations on test datasets during route decoding see Section 4.4. Both the base model and our model have the same graph encoder and are trained with similar settings for comparability. The reported times in parenthesis are estimated from times computed on another machine. The traditional methods we compare with are described more in detail in Section 2.5.2

	V3-C100			V3-C120		
Model	Value	Opt. gap	Time	Value	Opt. gap	Time
Exact-solver	124.61*	0%	(3315s)	-	-	-
SISR	126.19	1.27%	(1504s)	149.10	0%	(2200s)
VNS	129.32	3.78%	(1092s)	152.56	2.32%	(1606s)
ACO	140.61	12.84%	(1133s)	166.50	11.67%	(1589s)
FA	138.92	11.48%	(683s)	164.53	10.35%	(862s)
AM(Greedy)	141.14	13.27%	(1.62s)	164.57	10.38%	(1.86s)
Baseline (Greedy)	133.26	6.94%	1.13s	157.14	5.39%	1.28s
Our model (5 iterations)	133.05	6.77%	5.51s	157.33	5.51%	7.01s

Table 5.3: Results from the evaluation of our model trained with 5 iterations on test datasets during route decoding see Section 4.4. Both the base model and our model have the same graph encoder and are trained with similar settings for comparability. The reported times in parenthesis are estimated from times computed on another machine. The traditional methods we compare with are described more in detail in Section 2.5.2

5.3.2 Evaluation of gradient iterations

We further evaluate the method of gradient iterations for route decoding on in-distribution test data to assess if it can be used to improve the performance of a model on a specific instance over a pre-trained model's performance. For this, we take a sample of 100 test instances from the test dataset with 40 nodes. We train our model on each instance for 20 gradient updates and compare the average costs to our model before fine-tuning. In Table 5.4 we report the result for greedy decoding for both models on the test data.

Model	Avg. cost
Our model	58.98
Our model fine-tuned on benchmarks	56.97

Table 5.4: Results from the test set of 100 instances that were generated as the training data.

We choose to use 20 gradient updates because we see a trend of diminishing returns in many cases when increasing the number of gradient updates, as is shown in Figure 5.1. During training we also use a trick where we use parallelization to train with a batch size of 128 for the instance. A Wilcoxon Signed-Rank Test for paired data shows that the observed differences are statistically significant with $p < 0.001$.

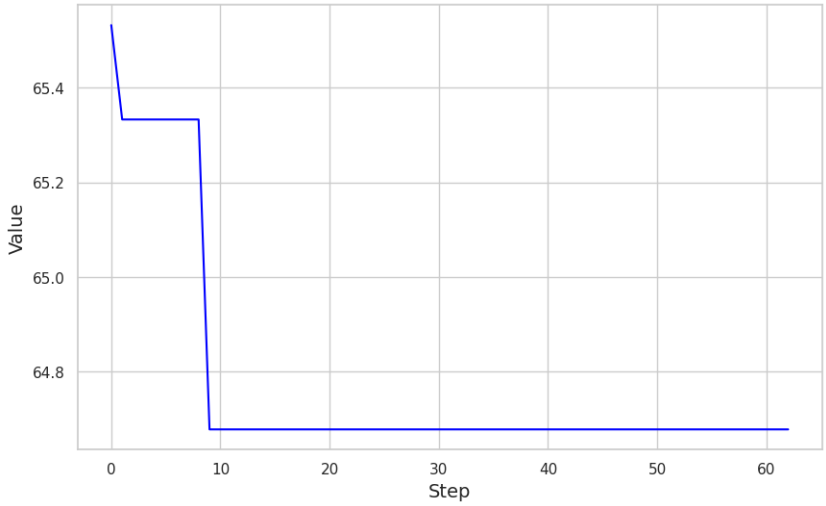

Figure 5.1: Graph of the best validation value observed as a function of gradient step for a specific instance.

5.4 Evaluation on CVRP-lib benchmarks

We further report results on out-of-distribution benchmarks represented in CVRP-lib that give insight into the generalization ability of our model. Specifically in Table 5.5, we present average results over the problem set P in CVRP-lib. The baseline model and our model are both trained on data with 40 nodes and the results reported are for the average optimality gap of instances with 15-100 nodes.

Method	Avg. opt. gap
Baseline	0.659
Our model	0.483
Baseline fine-tuned on benchmarks	0.4049
Our model fine-tuned on benchmarks	0.3636

Table 5.5: Results from fine-tuning on benchmarks for 20 epochs for both baseline model and our model. The benchmarks are from the problem set P from CVRP-lib with graphs of sizes from 15-100 nodes with the number of required routes from 2-15 routes.

In Appendix A we report the value and optimality gaps on specific benchmarks from these models.

6

Discussion

6.1 Baseline model

We choose to use a baseline model for our experiments since our experiments with iterative route decoding and gradient updates are not dependent on the model used for route decoding. Our ideas can be incorporated into any deep learning model used for route planning.

We utilized a baseline primarily to establish a solid foundation for our experiments. It provides a stable and reliable platform that has been previously tested and validated, thereby reducing the potential for errors and uncertainties that could arise from using an untested model. By employing a model recognized in the field, we can ensure that our findings are directly comparable to those of other researchers.

The baseline also serves as a control in our experiments, allowing us to isolate the effects of our modifications and interventions. By comparing the base model's performance with that of the modified models, we can determine the effectiveness of our proposed methods.

We do not extensively discuss the limitations and potential improvements of the baseline model with regard to the model architecture, as our project scope is mainly concerned with testing two specific methods to improve route decoding.

6.2 Iterative route decoding

The iterative route decoding approach, as implemented in this study, has demonstrated a marginal improvement over the baseline as seen in Table 5.1, 5.2 and 5.3. Although the results are not overwhelmingly significant, they indicate a promising direction for further research. The observed improvement, albeit modest, underscores the potential of iterative solutions in enhancing the performance of route decoding.

A key observation from our experiments is that the choice of iteration method seemed impactful. During development, we experimented with different methods of conditioning on the previous solution before converging to the one described in Section 4.4. Many of these methods had a worsening or no effect in preliminary experiments. We, therefore, believe further development in methods for encoding previous route context could lead to further improvement between iterations.

During the training phase, we implemented five iterations and generally observed an improvement in the model's performance with each iteration. This observation suggests that the model's learning benefits from iterative refinement. One potential avenue for further exploration could be introducing an iteration loss that encourages the model to seek improvements in every iteration.

Moreover, our findings suggest that encoding the previous route in the graph encoder could be beneficial. This could be achieved by introducing a graph network with edges representing the chosen route. Such an approach could provide a more comprehensive representation of the problem space, thereby potentially improving the model's ability to identify optimal routes.

However, it is important to note that these are preliminary findings, and further research is needed to validate these hypotheses. Future studies could explore strategies for iterative route decoding, such as varying the number of iterations or introducing different types of iteration loss. Additionally, the potential benefits of encoding the previous route in the graph encoder could be explored in more depth, for instance, by experimenting with different types of graph networks or investigating how the chosen route can be most effectively represented in the graph network.

6.3 Improved route planning with gradient updates

The results show the potential of improving route decoding with gradient updates, particularly for out-of-distribution data, as seen in Table 5.5. The method, which involves fine-tuning a pre-trained model on a single instance, has been shown to be able to yield significant improvements over the baseline for out-of-distribution data. This suggests that the model can adapt to unique features of the instance that were not adequately represented in the original training data.

Interestingly, while the improvements on in-distribution data were not as pronounced, see Table 5.4, the significant gains on out-of-distribution data underscore the practical utility of this approach. This is particularly relevant in real-world scenarios where the data may not always follow the same distribution as the training data. The ability of the model to adapt and perform well on such data is a valuable attribute, making it a more robust solution for route decoding tasks.

Potential use cases of this method are setting a time limit on route planning and running iterations until no change has been observed for a certain number of iterations, which can be an effective strategy. Most of the improvements were also observed early in the process, with minor enhancements occurring later. This suggests that the model adapts to the instance relatively quickly, making it a time-efficient solution.

Furthermore, the model demonstrated effectiveness both with and without conditioning on previous routes. This flexibility adds to the practicality of the approach, as it can be adapted to different models. It should be noted thought that our experiments are performed on relatively few test examples and to verify the effectiveness

and the uncertainty of the method further, a larger and more varied benchmark dataset could be required.

Lastly, the importance of a trained baseline model must be considered. The baseline architecture can, at weight initialization, not solve the route planning task for a single instance, even given a long training time with reinforcement learning. This highlights the value of transfer learning even in this context, where a pre-trained model can leverage its existing knowledge and adapt it to the specific characteristics of an instance.

6.4 Out of distribution results

We first note that the base model we build upon does show some generalization problems, as can be seen in the results on the CVRP-lib problems in Table A.1. Two potential causes for this are that the training data is relatively limited in the types of instances it covers. All node demands are uniformly distributed between 1-10, and the capacities for the vehicles are kept constant during training. We observe that the iterative method for route decoding, even without gradient updates, shows greater generalization potential than the base model, as seen in Table 5.5. This suggests that for out-of-distribution data, re-thinking (iterating) previous solutions can be of greater importance. When gradient updates are applied to both models, the results are comparable with our model, with iterative decoding having a slightly lower average cost over the benchmarks. The results suggest that when the test data is sufficiently different from the training data, the introduced iteration methods can be a method to bridge the performance gap between training and test data in practical applications without re-training.

6.5 Answering research questions

6.5.1 Conditioning on feasible solution

In our method for iterative route decoding, we let our model generate the first feasible solution and then iterate from this starting point. We conclude that the effect of this type of iteration is not major without gradient updates. This suggests that the model's built-in restrictions towards infeasible solutions provide a sufficient baseline for feasibility. For future research, the method of iteration, therefore, seems of higher importance.

6.5.2 Iteratively updating solutions

We introduced two methods for iterative route decoding: iterative decoding and fine-tuning on instances with gradient updates. Improvements are observed in both the simulated test data as well as the benchmarks from CVRP-lib. Therefore, the findings from our research suggest that the quality of solutions can be augmented by iteratively updating the solution. Although the improvement was not overwhelmingly significant, it does indicate a promising direction for further research.

Using gradient updates alongside iterative route decoding also showed potential for solution quality improvement. For out-of-distribution data, this method yielded significant improvements over the baseline model.

6.5.3 Using MuZero for route planning

Our approach to implementing MuZero for solving CVRP was less efficient than we had hypothesized. The primary issues we identified were slow training speed, convergence issues, and the time-consuming nature of the Monte Carlo Tree Search (MCTS) process.

The training speed of the MuZero model was significantly slower than anticipated. This was a major setback, as it increased the time required to reach a satisfactory level of performance. The slow training speed can be attributed to the complexity of the model and the computational resources required for its operation. This issue was further compounded by the convergence problems we encountered. The model struggled to converge to an optimal solution, a critical aspect of machine learning models. The convergence issue could be due to the complexity of the CVRP, which made it difficult for the model to find an optimal solution within a reasonable time frame.

Another significant challenge was the time taken by the MCTS process. The MCTS is an integral part of the MuZero model, used for planning and decision-making. However, in our case, it proved to be a bottleneck due to its time-consuming nature. The MCTS process involves many simulations, which can be computationally expensive and time-consuming when training from scratch.

Furthermore, we found it unnecessary to re-run the representation network every time, given that the graph representing the CVRP does not change. The representation network is used in MuZero to extract useful features from the environment, which are then used for planning and decision-making. However, in our case, the CVRP graph was relatively static, with minimal changes over time.

In conclusion, while the MuZero algorithm has shown promise in various domains, its application to route planning and the CVRP presented several challenges. These issues, particularly the slow training speed, convergence problems, and the time-consuming nature of the MCTS process, hindered the successful implementation of the model. Future research could explore ways to optimize the MuZero model for such applications by simplifying the training process and focusing most of the computation on the search for the next action.

6.5.4 Graph attention model for routing problems

The research question posed in this study was whether the graph-attention layer can be enhanced for routing problems. We introduced the Heterogeneous Graph Attention Network (HeteroGAT), an extension of the Graph Attention Network (GAT) specifically designed to handle heterogeneous graphs for routing problems. The HeteroGAT architecture was integrated into a MuZero architecture. However,

the model did not obtain satisfactory convergence, and thus, the efficiency of the architecture could not be fully evaluated.

Despite the lack of convergence, the theoretical underpinnings of the HeteroGAT architecture suggest the potential for enhancement of the graph-attention layer for routing problems. The HeteroGAT architecture was designed to handle the complexity of routing problems by considering the heterogeneity of the graph, i.e., the different types of nodes (customers and vehicles) and edges (connections between customers and vehicles). This is a significant enhancement over traditional GATs, which do not distinguish between different types of nodes and edges.

However, the lack of convergence in the MuZero architecture suggests there may be issues with the implementation or the compatibility of the HeteroGAT and MuZero architectures. It is also possible that the hyperparameters were not optimally tuned for the task. Further investigation is needed to identify the cause of the non-convergence and to evaluate the efficiency of the HeteroGAT architecture.

In conclusion, while the implemented HeteroGAT architecture did not provide empirical evidence to answer the research question due to the lack of convergence, the theoretical design of the architecture suggests the potential for enhancing the graph-attention layer for routing problems.

6.6 Future work

The findings of this study have opened up several directions for future research. Despite only marginal improvements over the baseline, the iterative route decoding approach has demonstrated potential for further exploration, especially on data different from the training data. Future studies could investigate different strategies for iterative route decoding, such as encoding the route with the graph's node embeddings instead of during route decoding.

The results of this study also suggest that the method of improving route decoding with gradient updates, particularly for out-of-distribution data, holds promise. Future research could explore the practical applications of this method, such as setting a time limit on route planning and running iterations until no change has been observed for a certain number of iterations. Another avenue of research would be to improve the efficiency of this method by improving the method of updating the gradients of the model, for example, maintaining an exponential average of weights.

The use of MuZero for route planning, while less efficient than hypothesized, could be further optimized in future studies. The primary challenges were identified as the slow training speed, convergence issues, and the time-consuming nature of the MCTS process. Future research could explore ways to optimize the MuZero model for such applications.

7

Conclusion

Our thesis has explored the potential of iterative route decoding and gradient updates in improving the performance of route decoding. The study highlighted the potential of the model to adapt to unique features of the instance that need to be adequately represented in the original training data, particularly for out-of-distribution data. Our results suggest that the method is a robust solution for route decoding tasks, capable of adapting to real-world scenarios where the data may sometimes follow a different distribution than the training data.

Bibliography

[1] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," en, *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://www.nature.com/articles/s41586-019-1724-z (visited on 11/30/2022).

[2] Y. Bengio, A. Lodi, and A. Prouvost, *Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon*, arXiv:1811.06128 [cs, stat], Mar. 2020. DOI: 10.48550/arXiv.1811.06128. [Online]. Available: http://arxiv.org/abs/1811.06128 (visited on 11/30/2022).

[3] J. Li, Y. Ma, R. Gao, *et al.*, "Deep Reinforcement Learning for Solving the Heterogeneous Capacitated Vehicle Routing Problem," *IEEE Transactions on Cybernetics*, vol. 52, no. 12, pp. 13 572–13 585, Dec. 2022, arXiv:2110.02629 [cs, math], ISSN: 2168-2267, 2168-2275. DOI: 10.1109/TCYB.2021.3111082. [Online]. Available: http://arxiv.org/abs/2110.02629 (visited on 11/30/2022).

[4] K. Lei, P. Guo, Y. Wang, X. Wu, and W. Zhao, *Solve routing problems with a residual edge-graph attention neural network*, arXiv:2105.02730 [cs], May 2021. DOI: 10.48550/arXiv.2105.02730. [Online]. Available: http://arxiv.org/abs/2105.02730 (visited on 11/30/2022).

[5] W. Kool, H. van Hoof, J. Gromicho, and M. Welling, *Deep Policy Dynamic Programming for Vehicle Routing Problems*, arXiv:2102.11756 [cs, stat], Dec. 2021. DOI: 10.48550/arXiv.2102.11756. [Online]. Available: http://arxiv.org/abs/2102.11756 (visited on 11/30/2022).

[6] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2, Montreal, Que., Canada: IEEE, 2005, pp. 729–734, ISBN: 9780780390485. DOI: 10.1109/IJCNN.2005.1555942. [Online]. Available: http://ieeexplore.ieee.org/document/1555942/ (visited on 12/09/2022).

[7] T. N. Kipf and M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, arXiv:1609.02907 [cs, stat], Feb. 2017. DOI: 10.48550/arXiv.1609.02907. [Online]. Available: http://arxiv.org/abs/1609.02907 (visited on 11/27/2022).

[8] P. Velikovi, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph Attention Networks*, arXiv:1710.10903 [cs, stat], Feb. 2018. DOI: 10.48550/arXiv.1710.10903. [Online]. Available: http://arxiv.org/abs/1710.10903 (visited on 12/09/2022).

[9] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Velikovi, *Combinatorial optimization and reasoning with graph neural networks*, arXiv:2102.09544 [cs, math, stat], Sep. 2022. DOI: 10.48550/arXiv.2102.09544. [Online]. Available: http://arxiv.org/abs/2102.09544 (visited on 11/30/2022).

[10] D. A. Spielman and S.-H. Teng, *Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time*, arXiv:cs/0111050, Oct. 2003. DOI: 10.48550/arXiv.cs/0111050. [Online]. Available: http://arxiv.org/abs/cs/0111050 (visited on 12/06/2022).

[11] J. V. Bernhard Korte, *Combinatorial Optimization. Theory and Algorithms* (Algorithms and Combinatorics), 2nd. Springer, 2002.

[12] B. Kallehauge, J. Larsen, O. B. Madsen, and M. M. Solomon, "Vehicle Routing Problem with Time Windows," en, in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds., Boston, MA: Springer US, 2005, pp. 67–98, ISBN: 9780387254869. DOI: 10.1007/0-387-25486-2_3. [Online]. Available: https://doi.org/10.1007/0-387-25486-2_3 (visited on 01/25/2023).

[13] A. Toriello, W. B. Haskell, and M. Poremba, "A Dynamic Traveling Salesman Problem with Stochastic Arc Costs," en, *Operations Research*, vol. 62, no. 5, pp. 1107–1125, Oct. 2014, ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.2014.1301. [Online]. Available: http://pubsonline.informs.org/doi/10.1287/opre.2014.1301 (visited on 11/30/2022).

[14] M. A. H. Akhand, Zahrul Jannat Peya, T. Sultana, and Al-Mahmud, "Solving Capacitated Vehicle Routing Problem with route optimization using Swarm Intelligence," in *2015 2nd International Conference on Electrical Information and Communication Technologies (EICT)*, Khulna, Bangladesh: IEEE, Dec. 2015, pp. 112–117, ISBN: 9781467392563 9781467392570. DOI: 10.1109/EICT.2015.7391932. [Online]. Available: http://ieeexplore.ieee.org/document/7391932/ (visited on 11/30/2022).

[15] J. Christiaens and G. Vanden Berghe, "Slack Induction by String Removals for Vehicle Routing Problems," en, *Transportation Science*, vol. 54, no. 2, pp. 417–433, Mar. 2020, ISSN: 0041-1655, 1526-5447. DOI: 10.1287/trsc.2019.0914. [Online]. Available: http://pubsonline.informs.org/doi/10.1287/trsc.2019.0914 (visited on 03/27/2023).

[16] Z. Xu and Y. Cai, "Variable neighborhood search for consistent vehicle routing problem," en, *Expert Systems with Applications*, vol. 113, pp. 66–76, Dec. 2018, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2018.07.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418304238 (visited on 08/09/2023).

[17] A. Palma-Blanco, E. R. González, and C. D. Paternina-Arboleda, "A Two-Pheromone Trail Ant Colony System Approach for the Heterogeneous Vehicle Routing Problem with Time Windows, Multiple Products and Product Incompatibility," en, in *Computational Logistics*, C. Paternina-Arboleda and S. VoSS, Eds., vol. 11756, Cham: Springer International Publishing, 2019, pp. 248–264, ISBN: 9783030311391 9783030311407. DOI: 10.1007/978-3-030-31140-7_16. [Online]. Available: http://link.springer.com/10.1007/978-3-030-31140-7_16 (visited on 08/09/2023).

[18] T. M. Mitchell, *Machine Learning*, en. McGraw-Hill, 1997, Google-Books-ID: EoYBngEACAAJ, ISBN: 9780071154673.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: https://www.deeplearningbook.org/ (visited on 03/15/2023).

[20] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention Is All You Need*, arXiv:1706.03762 [cs], Dec. 2017. DOI: 10.48550/arXiv.1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762 (visited on 03/25/2023).

[21] OpenAI, *GPT-4 Technical Report*, arXiv:2303.08774 [cs], Mar. 2023. DOI: 10.48550/arXiv.2303.08774. [Online]. Available: http://arxiv.org/abs/2303.08774 (visited on 03/25/2023).

[22] L. Ouyang, J. Wu, X. Jiang, *et al.*, *Training language models to follow instructions with human feedback*, arXiv:2203.02155 [cs], Mar. 2022. DOI: 10.48550/arXiv.2203.02155. [Online]. Available: http://arxiv.org/abs/2203.02155 (visited on 03/25/2023).

[23] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, *End-to-End Object Detection with Transformers*, arXiv:2005.12872 [cs], May 2020. DOI: 10.48550/arXiv.2005.12872. [Online]. Available: http://arxiv.org/abs/2005.12872 (visited on 03/25/2023).

[24] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, arXiv:2010.11929 [cs], Jun. 2021. DOI: 10.48550/arXiv.2010.11929. [Online]. Available: http://arxiv.org/abs/2010.11929 (visited on 03/25/2023).

[25] I. Beltagy, M. E. Peters, and A. Cohan, *Longformer: The Long-Document Transformer*, arXiv:2004.05150 [cs], Dec. 2020. DOI: 10.48550/arXiv.2004.05150. [Online]. Available: http://arxiv.org/abs/2004.05150 (visited on 03/25/2023).

[26] R. Child, S. Gray, A. Radford, and I. Sutskever, *Generating Long Sequences with Sparse Transformers*, arXiv:1904.10509 [cs, stat] version: 1, Apr. 2019. DOI: 10.48550/arXiv.1904.10509. [Online]. Available: http://arxiv.org/abs/1904.10509 (visited on 03/25/2023).

[27] N. Kitaev,. Kaiser, and A. Levskaya, *Reformer: The Efficient Transformer*, arXiv:2001.04451 [cs, stat], Feb. 2020. DOI: 10.48550/arXiv.2001.04451. [Online]. Available: http://arxiv.org/abs/2001.04451 (visited on 03/25/2023).

[28] R. S. Sutton and A. G. Barto, *Reinforcement Learning, second edition: An Introduction*, en. MIT Press, Nov. 2018, Google-Books-ID: sWV0DwAAQBAJ, ISBN: 9780262039246.

[29] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, arXiv:1712.01815 [cs], Dec. 2017. DOI: 10.48550/arXiv.1712.01815. [Online]. Available: http://arxiv.org/abs/1712.01815 (visited on 03/26/2023).

[30] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model," *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 2020, arXiv:1911.08265 [cs, stat], ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03051-4. [Online]. Available: http://arxiv.org/abs/1911.08265 (visited on 01/29/2023).

[31] A. Fawzi, M. Balog, A. Huang, *et al.*, "Discovering faster matrix multiplication algorithms with reinforcement learning," en, *Nature*, vol. 610, no. 7930, pp. 47–53, Oct. 2022, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-022-05172-4. [Online]. Available: https://www.nature.com/articles/s41586-022-05172-4 (visited on 12/05/2022).

[32] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Advances in Neural Information Processing Systems*, vol. 12, MIT Press, 1999. [Online]. Available: https://papers.nips.cc/paper_files/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html (visited on 08/08/2023).

[33] V. Mnih, A. P. Badia, M. Mirza, *et al.*, *Asynchronous Methods for Deep Reinforcement Learning*, arXiv:1602.01783 [cs], Jun. 2016. DOI: 10.48550/arXiv.1602.01783. [Online]. Available: http://arxiv.org/abs/1602.01783 (visited on 08/08/2023).

[34] W. Duvaud and A. Hainaut, "MuZero General: Open Reimplementation of MuZero," *GitHub*, 2019. [Online]. Available: https://github.com/werner-duvaud/muzero-general (visited on 03/13/2023).

A

Benchmark results

A.1 Baseline model

Benchmark	Value	Optimal solution	Optimality gap
P-n45-k5	1173.62	510.0	1.3
P-n76-k5	1649.61	627.0	1.63
P-n40-k5	1060.38	458.0	1.32
P-n60-k15	1290.04	968.0	0.33
P-n23-k8	579.53	529.0	0.1
P-n22-k2	265.01	216.0	0.23
P-n70-k10	1347.05	827.0	0.63
P-n55-k7	1006.07	568.0	0.77
P-n60-k10	1066.21	744.0	0.43
P-n51-k10	1225.97	741.0	0.65
P-n55-k8	1012.59	588.0	0.72
P-n22-k8	1076.6	603.0	0.79
P-n76-k4	1516.22	593.0	1.56
P-n50-k10	941.68	696.0	0.35
P-n16-k8	495.73	450.0	0.1
P-n21-k2	233.59	211.0	0.11
P-n50-k7	1014.75	554.0	0.83
P-n20-k2	231.4	216.0	0.07
P-n55-k10	933.89	694.0	0.35
P-n101-k4	2081.15	681.0	2.06
P-n50-k8	956.73	631.0	0.52
P-n55-k15	1245.87	989.0	0.26
P-n65-k10	1237.8	792.0	0.56
P-n19-k2	245.74	212.0	0.16

Table A.1: Results from baseline on problem set P from CVRP-lib.

A.2 Baseline model fine-tuned on benchmarks

Benchmark	Value	Optimal solution	Optimality gap
P-n45-k5	772.8	510.0	0.52
P-n76-k5	1323.05	627.0	1.11
P-n40-k5	683.45	458.0	0.49
P-n60-k15	1159.46	968.0	0.2
P-n23-k8	546.77	529.0	0.03
P-n22-k2	260.27	216.0	0.2
P-n70-k10	1289.2	827.0	0.56
P-n55-k7	822.41	568.0	0.45
P-n60-k10	1040.37	744.0	0.4
P-n51-k10	878.98	741.0	0.19
P-n55-k8	781.48	588.0	0.33
P-n22-k8	747.23	603.0	0.24
P-n76-k4	1120.86	593.0	0.89
P-n50-k10	893.81	696.0	0.28
P-n16-k8	451.95	450.0	0.0
P-n21-k2	292.93	211.0	0.39
P-n50-k7	845.37	554.0	0.53
P-n20-k2	255.1	216.0	0.18
P-n55-k10	1119.79	694.0	0.61
P-n101-k4	1486.71	681.0	1.18
P-n50-k8	778.58	631.0	0.23
P-n55-k15	1155.41	989.0	0.17
P-n65-k10	1089.62	792.0	0.38
P-n19-k2	245.32	212.0	0.16

Table A.2: Results from baseline model on problem set P from CVRP-lib fine-tuned on benchmarks.

A.3 Our model

Benchmark	Value	Optimal solution	Optimality gap
P-n45-k5	1199.18	510.0	1.35
P-n76-k5	1708.94	627.0	1.73
P-n40-k5	1085.27	458.0	1.37
P-n60-k15	1387.85	968.0	0.43
P-n23-k8	562.83	529.0	0.06
P-n22-k2	351.6	216.0	0.63
P-n70-k10	1577.68	827.0	0.91
P-n55-k7	1261.63	568.0	1.22
P-n60-k10	1235.67	744.0	0.66
P-n51-k10	1171.44	741.0	0.58
P-n55-k8	1242.49	588.0	1.11
P-n22-k8	870.9	603.0	0.44
P-n76-k4	1736.06	593.0	1.93
P-n50-k10	997.33	696.0	0.43
P-n16-k8	487.12	450.0	0.08
P-n21-k2	295.33	211.0	0.4
P-n50-k7	1219.76	554.0	1.2
P-n20-k2	267.88	216.0	0.24
P-n55-k10	1214.3	694.0	0.75
P-n101-k4	2234.18	681.0	2.28
P-n50-k8	995.97	631.0	0.58
P-n55-k15	1164.37	989.0	0.18
P-n65-k10	1355.37	792.0	0.71
P-n19-k2	272.64	212.0	0.29

Table A.3: Results from our model on problem set P from CVRP-lib.

A.4 Our model fine-tuned on benchmarks

Benchmark	Value	Optimal solution	Optimality gap
P-n45-k5	764.21	510.0	0.5
P-n76-k5	1095.08	627.0	0.75
P-n40-k5	629.16	458.0	0.37
P-n60-k15	1134.89	968.0	0.17
P-n23-k8	549.48	529.0	0.04
P-n22-k2	261.71	216.0	0.21
P-n70-k10	1137.73	827.0	0.38
P-n55-k7	846.89	568.0	0.49
P-n60-k10	985.4	744.0	0.32
P-n51-k10	941.88	741.0	0.27
P-n55-k8	827.11	588.0	0.41
P-n22-k8	764.15	603.0	0.27
P-n76-k4	1074.26	593.0	0.81
P-n50-k10	890.96	696.0	0.28
P-n16-k8	485.01	450.0	0.08
P-n21-k2	233.59	211.0	0.11
P-n50-k7	813.43	554.0	0.47
P-n20-k2	241.68	216.0	0.12
P-n55-k10	928.02	694.0	0.34
P-n101-k4	1445.08	681.0	1.12
P-n50-k8	891.8	631.0	0.41
P-n55-k15	1105.52	989.0	0.12
P-n65-k10	1097.43	792.0	0.39
P-n19-k2	277.57	212.0	0.31

Table A.4: Results from our model on problem set P from CVRP-lib fine-tuned on benchmarks.