EINDHOVEN UNIVERSITY OF TECHNOLOGY
Eindhoven University of Technology

MASTER

Combining Deep Learning and Simulated Annealing to Solve Vehicle Routing Problems with Time Windows

van Eekelen, P.H.A.

Award date:
2020

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Combining Deep Learning and Simulated Annealing to Solve Vehicle Routing Problems with Time Windows

Author
Paul van Eekelen Bsc, 0776673

Supervisor Technische Universiteit Eindhoven
Prof. dr. Frits Spieksma

Second supervisor Technische Universiteit Eindhoven
Dr. Christopher Hojny

Supervisor CQM
Dr. Frans de Ruiter

Date
November 15, 2020

Abstract

Artificial Intelligence Algorithms successfully solve many problems with a high intrinsic complexity. Routing problems are known to be notoriously hard to address by these algorithms. In this thesis we investigate whether the combined use of Deep Learning, Simulated Annealing and Time Dependent Simple Temporal Networks can be satisfactorily used to solve a class of routing problems in actual, daily practice. This case study aims to improve the daily routing of Valys, a well-known transport planning in The Netherlands. An improvement of two percent on the existing routing saves about 10000 kilometers a day. Numerical results show that Deep Learning can successfully estimate the feasibility of actions performed by the simulated annealing algorithm. The results of the developed Python prototype constitute a good first start to addressing the practical problems while directions are given to solve the remaining issues.

Contents

1 Introduction **1**

2 Problem Description **4**

2.1 Important considerations for the model 4

2.2 Integer Linear Programming 6

3 Valys Algorithm **12**

3.1 Simulated Annealing 12

3.2 Simulated Annealing for the Valys Algorithm 14

3.3 Checking Feasibility 16

4 Deep Learning **21**

4.1 Multilayer Perceptrons 21

4.2 Overfitting 25

4.3 Convolutional Neural Networks 25

4.4 Normalization layers 26

4.5 Type I and Type II errors 26

4.6 PyTorch 27

5 Data Generation **29**

5.1 Data Size 29

5.2 Data Sampling 30

5.3 Train, Validation and Test 31

5.4 Data processing 31

6 Models for the Valys problem **34**

6.1 Inputlayer 34

6.2 Multilayer Perceptron 34

6.3 Convolutional models 35

7 Results **36**

7.1 Instance Normalization 36

7.2 Batch Normalization 39

8 Discussion and Conclusion **42**

8.1 Discussion of the various models 42
8.2 Conclusion 43
8.3 Further research 43

Appendices **48**

A Speed test tables **48**

B Threshold Graphs **59**

1 Introduction

On a daily basis we encounter our own mini versions of the Traveling Salesman problem. Is it faster to go to the grocery store first or should we visit it after going to the butcher? What is the optimal route for a pizza courier to visit all her customers? The Traveling Salesman problem is the problem of finding the best order in which you should visit n places. The quality of a particular order is usually the duration of visiting those places in that order. There are a lot of variations on the Traveling Salesman problem where additional restrictions or possibilities are added. A possible addition is multiple available vehicles with capacities instead of a single vehicle called the Vehicle Routing Problem, VRP [19]. To solve this problem, Clark and Wright [13] published an algorithm called the Savings Algorithm. Another variation that occurs often is the Vehicle Routing Problem with Time Windows, so called VRPTW [5]. The VRPTW is a generalisation of the Traveling Salesman problem, where instead of one vehicle we can have more vehicles to visit all the places. See for example Figure 1. On the left-hand-side you see 16 blue dots and 1 black dot. Now imagine, all the blue dots are places that need to be connected to the black dot by a vehicle that drives by. On the right-hand-side you see a solution with 4 vehicles (in red, yellow, green and blue). The problem of finding an optimal solution adhering to these requirements is called the Vehicle Routing Problem. This also can be made more complicated by adding time windows, meaning that every blue dot has to be visited in one specific time window. Then it is called the vehicle routing problem with time windows and that is our problem of interest, which is described by Savelsbergh [18]. Many solution techniques have been proposed in the literature to solve this. Some of the most recent state-of-the-art algorithms can be found in -see papers on lee and lim benchmark website-.

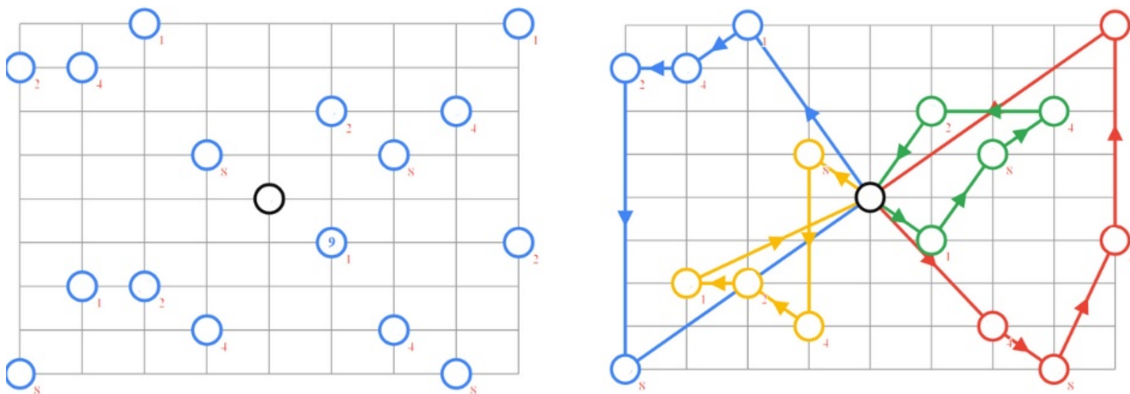

Figure 1: A solution to a vehicle routing problem. [1]

[1] Picture taken from [1]

Our variant of the VRPTW concerns the Valys taxi-contract, the biggest taxi-contract in the Netherlands. The contract facilitates elderly and disabled carpooling transport throughout the Netherlands. For this subsidy around 450,000 citizens are eligible and around 200,000 use their right. Every day about 5,000 taxi rides requests for the following day come in. With regular taxis each ride is serviced completely before starting with the next one. These rides however, can be combined such that people from different requests can end up in the same taxi at the same time. So this is a variant of carpooling with taxis. At night, an algorithm combines these rides as good as possible with respect to costs. This is a variant of the Vehicle Routing Problem with Time Windows mentioned above. The VRPTW is a NP-Hard problem, which means that no exact polynomial time algorithm for the VRPTW problem exists unless P = NP. However, an often made mistake is that all instances in practice are therefore hard to solve. In practice, very often fast algorithms exist for NP-Hard problems that can find good solutions in time reasonable for the problem at hand. The existing algorithm reduces on average the scheduled driving kilometers by 100,000 kilometers, when compared to the old decentralized scheduling, which was done by hand. If we could improve on the existing algorithm, we could potentially save 10,000 kilometers per day by allowing speeding up the algorithm, which allows us to do more local neighbourhood searches in the same time.. This existing algorithm uses a Simulated Annealing part to try and find new solutions and Time Dependent Simple Temporal Networks for checking feasibility of new solutions. Simulated Annealing is a heuristic method to find the best possible solution in a solution field [20]. A solution is feasible if it adheres to all the restrictions of the problem, such as respecting the time windows, maximum passenger time, vehicle capacities and complex driver resting times. Time Dependent Simple Temporal Networks are labelled graphs on which we try to find a shortest path to check for feasibility of a proposed route[17]. This last check is a time expensive check, especially since multiple possible breaks must be scheduled in between to comply with EU regulations on drivers resting times and shift lengths. Since we only have limited time, we would want to do this check less often to increase the number of iterations the simulated can perform in the same time. This is where this project comes in. Can the insertion of a deep learning model in an existing simulated annealing algorithm improve the solution of a VRPTW by quickly guessing feasibility of a new route, making expensive checks only for promising routes?

Deep learning is a method of artificial intelligence which uses neural networks to mimic mappings as good as possible [7]. In our case we want to mimic the feasibility mapping of potential solutions. This deep learning algorithm tries to classify solutions into feasible and non-feasible solutions as good as possible in a relative short time. We could also try to solve the VRPTW entirely with a deep learning model similarly as done by Nazari, Takac, Oroojlooy and Snyder [15], Lin, Ghaddar and Nathwani [12], Peng, Wang and Zhang [16] and Joe and Chuin Lau [9], but this has only proven to work in reasonable time for medium size, around 100 places that need to be visited, problem instances. These models take about 15 minutes for these 100 size problems. Since our

problem sizes can be around 10,000, two places for each of the 5,000 customers, this would probably result in computation times of at least days. We need to calculate our solution within a few hours, so this direct approach is out of the question.

In Chapter 2 we talk more about the problem and define it precisely. After that we explain the existing algorithm in Chapter 3. We address the relevant theory: the Simulated Annealing algorithm and Time Dependent Temporal Networks. Then, in Chapter 4, we continue with the relevant theory about Deep Learning. We talk about the different kinds of Neural Networks available to us and all the different kinds of layers in them. For these networks we use a Python library, Pytorch and we need data to train and test them. In Chapter 5 we explain how we extracted this data. After this step we made a few different models for testing. Which hyperparameters we choose and why is explained in Chapter 6. The performance of these models can be found in Chapter 7. This includes threshold tests. These results are discussed in Chapter 8 and what they mean for the future. Some future work possibilities are outlined there.

2 Problem Description

In this chapter we explain in further detail what the problem entails and give a formal definition. The problem at hand is a variation on the Vehicle Routing Problem with Time Windows[18]. On this problem there are some noteworthy organizational considerations and computational considerations. These can be found in the section 2.1. We transform the problem into an Integer Linear Program in section 2.2. There we state two different ways to state the model in a particular form suited for Integer Linear Programming.

2.1 Important considerations for the model

In our problem we get requests for taxi rides from customers to get from point A to point B within a certain time limit. After these rides the taxis have to return to their respective depots. The aim is to find the in total shortest combination of these requests with respect to time. In finding a solution for this problem, Valys has to take some considerations in mind. These considerations can be divided into organizational and computational considerations. We state these considerations below and after that go into further detail about them.

Organizational Considerations

- There is a maximum for the duration that a passenger can be in a car.
- Every passenger has a pick-up or drop-off time window.
- There are 2 different types of vehicles.
- There are various laws concerning drivers that need to be adhered.
- Drivers have to end their route at their depot.

Computational Considerations

- There are around 5000 rides that need to be scheduled every day, with up to 15000 during peak days such as Christmas.
- Travel times are dynamic.
- The capacity of the taxis is limited.
- There is a maximum for the computing time.

There is a maximum for the duration that a passenger can be in a car: The problem is a carpooling problem, so probably customers are not directly taken to their destination. However nobody wants to step into a car at 09:00h in Amsterdam and then dropped off in Haarlem at 21:00h. That is why a restriction is put in place where no ride of a passenger should take longer than 150% of the time it would take to drive the passenger directly from pick-up to drop-off.

Every passenger has a pick-up or drop-off deadline: Standard customers can assign a pick-up window of 30 minutes wherein they want to be picked up. Other customers who need to be somewhere at a specific time, for example funerals, do not assign a pick-up window. Instead, they give a latest arrival time which in turn implies a pick-up window. In the Integer Program formulation in section 2.2 we only use the (implied) pick-up windows.

There are two different types of vehicles and vehicle capacities: A taxi for four people besides the driver with no room for wheelchairs and a bus taxi for six people besides the driver with room for two wheelchairs. These also come with a different cost to operate.

There are various laws concerning drivers that need to be adhered: Drivers ofcourse have to adhere to the driving laws in the Netherlands. There are also restrictions for taxi drivers. For example, if their shift lasts longer than 5.5 hours, then they have to get at least one break of 15 minutes. Also their shift cannot exceed 10 hours in one day.

Drivers have to end at their depot: For our calculation purposes, each route for an individual taxi driver starts at their first customer, but does not end at the drop off location of their last customer. Instead it ends at their depot location, which can be different for every taxi company. Each taxi-route is assigned to the taxi-company with the same postal code as the first customer of that route. The depot of each taxi-company is in the same postal code as that company.

There are around 5,000 rides that need to be scheduled every day: Before every day people can make taxi requests where to be picked up and where to be brought to and they can include a pick-up time window. On a busy weekend day Valys gets about 5,000 requests, so the algorithm should be able to handle these numbers. On special days, such as Christmas, the number of requests could even reach 15,000. This, in combination with other restrictions and the type of the problem, makes it that we cannot find an optimum in adequate time. The optimum can only be approximated.

Travel times are dynamic: Taxis cannot always drive at the maximum allowed speed, for example during peak hours. So there is a system in place which predicts these times and adjust distances accordingly. For example, during peak hours, the predicted driving time will be multiplied with some factor $r > 1$. This predicted driving time is supplied by Google.

There is a maximum for the computing time: Up until midnight requests for the following day can be submitted. This leaves only the preceding night of every day as

the time period in which we can schedule all the rides. Since there are also other time consuming parts in the entire procedure, this will leave about 4 hours in computation time for our algorithm.

2.2 Integer Linear Programming

For optimizing problems with integer variables a perfect solution can be found if the number of variables and corresponding complexity of the problem is low enough. This can be done through Integer Linear Programming, ILP[23]. For this to work, we need to transfer the definition of the problem into the following form.

(1)	$\min$	$c^T x$	
(2)	subject to	$A_i x \leq b_i,$	$\forall i \in I$
(3)		$x \geq 0,$	
(4)	and	$x \in \mathbb{Z}^n,$	

where x, c and b_i are vectors, A_i are matrices and $i \in I$ are indices. The formula 1 expresses the cost function we need to minimize. The inequalities described in line 2 describe the various limitations and considerations for our model. The inequalities 3 and 4 are there to put the variable x, and other variables if they arise, into the right domain. In this part we will show by transforming our problem into this form, that there will be too many variables and thus the problem cannot be solved perfectly for all instances with standard techniques in adequate time. First we start with listing the variables used in the formulation:

- In our problem statement we consider two different types of taxis: taxis with six person capacity and two wheelchair capacity and taxis with four person capacity and no wheelchair capacity. We assume that these taxis always can achieve the same speed and thus take the same time to get from place A to place B. This means that we only need to consider if taxis do not exceed six person capacity and two wheelchair capacity in our ILP formulation. If in a solution only at most four persons and no wheelchairs are assigned to a taxi we can change that taxi to a four person taxi instead of a six person one. We define our taxis k to be in the set K.

- To get a integer program we discretize the time $t \in T$. We only consider minutes and thus round off all seconds to the upper minute.

- Let Z be the set of requests.

- Let P be the set of all the places that the taxis visit. Then let $P_d \subset P$ be the subset of the places P in which all places belong that are depots. Let $p_k \in P_d$ be the depot

location of taxi driver k. $p_0 \in P$ is the auxiliary starting place of all routes, which has distance 0 to all other places. Furthermore, we assume that all drop-off and pick-up locations are unique. Let $p_z, d_z \in P$ be the pick-up and drop-off locations respectively of request z with $z \in Z$.

- For each pair $(i, j) \in P^2$, taxi $k \in K$ and timeunit $t \in T$ we define the binary variable $x_{i,j,k,t}$ as an indicator variable whether the route from place i to place j is started by taxi k at time t. This route is completed without interruption.
- $c_{i,j,k,t}$ is the cost of the route from place i to place j by taxi k on starttime t.
- $w_{k,t}$ is the indicator variable for whether taxi k takes a break on timestep t. Let b be the size of these breaks in minutes. For now this is 15 minutes, but this could change.
- The y_k are auxiliary variables used to activate different equations whether a taxi k needs breaks or not. M is another auxiliary variable. M should be a number such that equations 13, 14 and 15 are satisfied, when the factor multiplied with M is equal to 1. For our purposes $M = |X|$ is sufficient.
- Let f_1 and f_2 be the maximum time allowed for all taxi drivers in minutes per day with breaks or without breaks respectively. In the current model we have $f_1 = 600$ minutes and $f_2 = 330$ minutes, but this could change.
- $g_z \in G$ is the number of passengers that the request $z \in Z$ entails.
- $h_z \in H$ is the number of wheelchair passengers that the request $z \in Z$ entails.
- max_z is the maximum duration that the passengers from request $z \in Z$ can be in a taxi.
- PW_z is the middle of the pick-up window for request $z \in Z$. Let a be half the size of the time windows in minutes. In our model for now this is 15 minutes, but this can change.

Now we can define the constraints of our ILP:

$$\min \sum_{i,j \in P, k \in K, t \in T} c_{i,j,k,t} \cdot x_{i,j,k,t} \tag{5}$$

$$\sum_{j \in P, k \in K, t \in T} x_{i,j,k,t} = 1 \qquad \forall i \in P \backslash (P_d \cup \{p_0\}) \tag{6}$$

$$\sum_{i \in P, k \in K, t \in T} x_{i,j,k,t} = 1 \qquad \forall j \in P \backslash (P_d \cup \{p_0\}) \tag{7}$$

$$\sum_{i \in P, k \in K, t \in T} x_{i,p_0,k,t} = 0 \tag{8}$$

$$\sum_{j \in P, t \in T} x_{p_k,j,k,t} = 0 \qquad \forall k \in K \tag{9}$$

$$\sum_{j \in P, t \in T} x_{p_0,j,k,t} \leq 1 \qquad \forall k \in K \tag{10}$$

$$\sum_{i \in P, t \in T} x_{i,p_k,k,t} \leq 1 \qquad \forall k \in K \tag{11}$$

$$\sum_{i,j \in P, t \in T} c_{i,j,k,t} \cdot x_{i,j,k,t} \leq f_1 \qquad \forall k \in K \tag{12}$$

$$M(1 - y_k) + \sum_{t \in T} w_{k,t} \geq 1 \qquad \forall k \in K \tag{13}$$

$$-M(1 - y_k) + \sum_{t \in T} w_{k,t} \leq 1 \qquad \forall k \in K \tag{14}$$

$$\sum_{i,j \in P, t \in T} c_{i,j,k,t} \cdot x_{i,j,k,t} - M \cdot y_k \leq f_2 \qquad \forall k \in K \tag{15}$$

$$\sum_{j \in P, t \in T} (x_{p_z,j,k,t} - x_{d_z,j,k,t}) = 0 \qquad \forall k \in K, z \in Z \tag{16}$$

$$\sum_{j \in P, 0 \leq t \leq s} (x_{p_z,j,k,t} - x_{d_z,j,k,t}) \geq 0 \qquad \forall k \in K, z \in Z, s \in T \tag{17}$$

$$\sum_{j \in P, PW_z - a \leq t \leq PW_z + a, k \in K} x_{p_z,j,k,t} = 1 \qquad \forall z \in Z \tag{18}$$

$$(19) \quad \sum_{j\in P, z\in Z, 0\le t\le s} (g_z \cdot x_{p_z,j,k,t} - g_z \cdot x_{d_z,j,k,t}) \le 6 \qquad \forall k \in K, s \in T$$

$$(20) \quad \sum_{j\in P, z\in Z, 0\le t\le s} (h_z \cdot x_{p_z,j,k,t} - h_z \cdot x_{d_z,j,k,t}) \le 2 \qquad \forall k \in K, s \in T$$

$$(21) \quad \sum_{j\in P, z\in Z, 0\le t\le s} (g_z \cdot x_{p_z,j,k,t} - g_z \cdot x_{d_z,j,k,t}) + \sum_{j\in P, z\in Z, 0\le t\le s} (h_z \cdot x_{p_z,j,k,t} - h_z \cdot x_{d_z,j,k,t}) - M \cdot (1 - w_{k,s}) \le 0 \qquad \forall k \in K, s \in T$$

$$(22) \quad \sum_{j\in P, t\in T, k\in K} (t \cdot x_{d_z,j,k,t} - t \cdot x_{p_z,j,k,t}) \le max_z \quad \forall z \in Z$$

$$(23) \quad \sum_{j\in P} t \cdot x_{i,j,k,t} - \sum_{l\in P, s\le t} (x_{l,i,k,s} \cdot (c_{l,i,k,s} + s) + \sum_{s\le r\le t} w_{k,r} \cdot b) \ge 0 \qquad \forall i \in P, k \in K, t \in T$$

$$(24) \quad \sum_{i\in P, t\le s} (x_{i,j,k,t} - x_{j,i,k,t}) \ge 0 \qquad \forall j \in P, k \in K, s \in T$$

$$(25) \quad w_{k,t} \in \{0,1\} \quad \forall k \in K, t \in T$$

$$(26) \quad x_{i,j,k,t} \in \{0,1\} \quad \forall i \in P, j \in P, k \in K, t \in T$$

$$(27) \quad y_k \in \{0,1\} \quad \forall k \in K$$

- In formula 5 the optimization goal is stated which resembles the amount of time that is needed to service all customers.

- Equations 6 and 7 ensure that each destination and pick-up location is visited exactly once.

- Equations 8 and 9 and inequalities 10 and 11 ensure that the depots are incorporated correctly into the model. So each taxis ends their tour at their respective depot, each taxi starts from the auxiliary depot, no taxi returns to the auxiliary depot and no taxi leaves from a depot.

- Inequalities 12, 13, 14 and 15 are there for the maximum driving time and breaks restrictions for the taxi drivers. In equation 15 we have f_2 representing the maximum allowed time a taxi driver can drive without breaks. In equation 12 we see f_1 which represents the maximum allowed time a drive can drive with breaks. We subtracted the breaks durance, which is $2 \cdot 15 = 30$ minutes, since these breaks are already ensured to take place by equations 13 and 14.

- Equations 16 and inequalities 17 ensure that the taxi who picks up a passenger also drops off that passenger and in that order. Equation 16 checks whether a taxi k only picks up customers that it drops off. Equation 17 ensures that at each timestep s the taxi k has not visited the drop-off location of each customer z before their pick-up location.
- Equation 18 corresponds to the restriction that all passengers must be picked up in their pickup window.
- Inequalities 19 and 20 ensure that the capacity of the taxis is not exceeded.
- By adding inequalities 21 we ensure that breaks are only taken by taxi drivers when no passengers are in the taxi.
- Inequalities 22 is there to ensure that no passenger is longer in a taxi than their allotted time.
- Inequalities 23 is there to ensure that the pickup time assignments are reachable with respect to time from the previous location.
- Inequalities 24 is there to ensure that no place is left before it is reached.
- Expressions 25, 26 and 27 are there to confine the variables to their respective domains.

So we have variables in the order of $|P|^2 \cdot |K| \cdot |T| \approx 5000 \cdot 5000 \cdot K \cdot 24 \cdot 60 \approx 3.6 \cdot 10^{10}$ variables. We also have the same order of magnitude in equations. Solving all these constraints with this order of variables is infeasible within our time constraints. Since solving these equations is equal to finding an optimal solution to our problem, we can not find an optimal solution to our problem with this model using the standard techniques for this.

2.2.1 A different formulation

The formulation in equation 5 till equation 27 is not unique. In this subsection, we provide an alternative mathematical formulation based on an ordered set of routes that we refer to as a *route-plan*. A route-plan is an ordered set of routes which can be feasibly carried out by a taxi-driver and which satisfies a number of requests. A route-plan that serves request z, visits both p_z and d_z in that order. Let $A = \{1, ..., m\}$ be the set of all route-plans. We define b_i to be the binary indicator variable that determines whether we use route-plan i, $i = 1, ..., m$. Let d_i be the cost of routeplan i $(1 \leq i \leq m)$. We write $z \in i$ to indicate

that request $z \in Z$ is served by route-plan $i, i = 1, ..., m$. The model is then formulated in a compact way as follows:

$$\text{(28)} \qquad \min \qquad \sum_{i=1}^{m} d_i b_i$$

$$\text{(29)} \qquad \text{subject to} \qquad \sum_{i:z \in i} b_i = 1 \qquad \forall z \in Z$$

$$\text{(30)} \qquad b_i \in \{0, 1\} \qquad \forall i = 1, \ldots, m$$

Expression 28 is there to minimize the cost of the solution we want to find. Equation 29 ensures that each request z is in exactly one scheduled route. The expression 30 ensures that our binary decision variables b_i are indeed binary. The biggest difference with our former formulation is that we assume we can have access to the set of feasible routes. So we need a way to fabricate this set. Whereas in the first formulation we restrict ourselves only to feasible routes through equations 6 till 27.

3 Valys Algorithm

In this chapter we explain the existing solution to the problem: The Valys Algorithm. This algorithm is a combination of Simulated Annealing, some simple partial route feasibility checks from here on called fast checks and the Time-Dependent Simple Temporal Network. In section 3.1 we will explain Simulated Annealing in general and then show the implementation in the Valys Algorithm in section 3.2. In section 3.3.1 the fast checks are explained in more detail and we mention where our deep learning algorithm is inserted. The full feasibility check is explained in section 3.3.2.

3.1 Simulated Annealing

Simulated Annealing, as explained by van Laarhoven and Aarts [20], is a heuristic method for trying to approximate the optimum of a given function. A usual problem with methods that try to find a global optimum is that they get stuck in a local optimum. Simulated Annealing tries to tackle this by sometimes continuing with worse solutions to try to get out of local optima and to the global optimum.

Algorithm 1: Simulated Annealing Pseudo Code [20]

Data: Function f, Random Neighbour function RNB, epochs e, starting solution s_0, algorithm constant k
Result: Solution s

1. Let $s \leftarrow s_0$ be the initial solution;
2. $T \leftarrow e$;
3. **while** $T > 0$ **do**
4. $s^* \leftarrow RNB(s)$;
5.   **if** $f(s^*) \leq f(s)$ **then**
6. $s \leftarrow s_*$
7.   **else**
8.   **if** $e^{\frac{f(s)-f(s^*)}{k \cdot T}} \geq Random(0,1)$ **then**
9. $s \leftarrow s_*$
10. $T \leftarrow T - 1$;

In the algorithm 1 the initial solution s_0 can be chosen in various ways. Usually this a trivial solution, but it could also be a random chosen solution if such a thing can be constructed easily. The epochs e is the number of iterations you want the algorithm to run. This is usually dependant on the amount of time you have. The Random Neighbour function $RNB(s)$ is a function that selects a neighbour of s according to a distribution. This distribution is the uniform distribution, but can be different if desired. In most simulated annealing approaches, as is in ours, feasibility is mainted for each intermediate

solution within the algorithm. This requirement fits into the framework above by setting $f(s) = \infty$ for all infeasible s.

3.1.1 Temperature

In the Simulated Annealing algorithm 1 we use a variable T, called temperature. This variable starts out at a high number we called $epoch$ and decreases linearly with each iteration of the algorithm. The temperature is used to determine whether the considered solution is accepted or not. The considered solution is always accepted if it is the best so far, but if it is worse then it is accepted with a certain probability. This probability depends on the temperature and the difference in quality between the best and the considered solution. With a higher temperature the probability of acceptance will be higher. This makes it possible for the algorithm to accept low quality solutions in the beginning in order to explore the solution space. This is so that in the beginning of the algorithm we accept bad solutions to try to explore the solution space. In the later stages we care more about getting to the global minimum, or at least a local minimum, and so we accept worse solutions less often. This process is analogous to the annealing process in metallurgy. In that process, a metal that is heated past its melting point is slowly cooled, analogous to our changing temperature variable to prevent imperfections from occurring in the cooled metal. By slowly cooling, the atoms arrange themselves into states with the lowest possible energy. This lowest possible energy state is analogous to our quality of a solution.

3.1.2 Neighbourhood

First we will define what a neighbourhood is. A neighbourhood of a point is a set of other points with the constraint that when a point a is in a neighbourhood of point b, then point b is also in the neighbourhood of point a. In this case the points a and b are called neighbours. We will now define the neighbourhood:

$$Neighbourhood(s_n) = \{s_{n_0}, s_{n_1}, s_{n_2}, ..., s_{n_{m_n-1}}\} \tag{31}$$

$$Neighbour(s, s_0) = \text{True} \quad \text{iff } s_0 \in Neighbourhood(s) \tag{32}$$

$$Neighbour(s, s_0) = Neighbour(s_0, s) \tag{33}$$

Where n is the number of solutions and n_{m_n-1} is the number of neighbours of Sol_n. The Neighbour relation is symmetrical, as seen in equation 33, and irreflexive. A similar definition is stated by van Laarhoven and Aarts [20]. We call going from one solution to a neighbour solution a move. We also want total connectivity, i.e. we can get from any solution Sol_i to any other solution Sol_j through a finite number of moves. This is so we don't exclude an entire part of our solution space through our choice of the initial solution. For easy calculations we want that the computations for cost and feasibility

for a neighbour solution are small. So our neighbours should look similar and in such in way that computing the cost of our new solution is small. A simulated annealing algorithm starts with a random or constructed solution to the problem and calculates its cost. Then it goes to a random neighbour solution and calculates its cost. For example, when one customer is transferred to a different taxi. Now we need to know whether we want to continue with this neighbour or reject it and try another neighbour. The acceptance decision is explained in the next section.

3.1.3 Acceptance

After we select a neighbour solution s^* of our current solution s, we decide whether we want to continue with this solution s^*. For this we check what the score $f(s^*)$ for this new solution is and compare it to the score $f(s)$ of our current solution. If $f(s^*) \leq f(s)$, then we continue the algorithm with $s \leftarrow s^*$. If not, we only continue with s^* with a certain probability. This probability is [20]:

$$e^{\frac{f(s)-f(s^*)}{k \cdot T}} \tag{34}$$

where T is the temperature, e is Euler's number and k a constant specified for this algorithm. In equation 34 we can see that the probability decreases when s^* is bigger and when the T gets lower. So when the new solution s^* is worse or when we are further in the algorithm, we will accept s^* with a lower probability. After the acceptance or rejection of s^* we will decrease the temperature T, select a neighbour of our current s and continue with the algorithm from the start. If the temperature reaches 0 we will terminate the algorithm.

3.2 Simulated Annealing for the Valys Algorithm

After discussing the general form of the Simulated Annealing algorithm, we can now explain the details of the implementation in the Valys Algorithm.

We define a solution as an assignment of all the customer requests to the taxis and an order of pick up and delivery of their customers for every taxi. For example, let $0, 1, 2, 3, 4, 5, 6, 7, 8$ and 9 be customerID's and a, b, c, d, e and f be taxis. Then a solution is:
a : $2, 3, 3, 4, 2, 4$
b : $0, 1, 1, 0$
c :
d : $8, 7, 6, 6, 8, 7$
e :
f : $5, 5$
and

$a : 2, 3, 3, 4, 2, 4$
$b : 0, 1, 1, 0$
$c :$
$d : 8, 7, 6, 8, 7, 6$
$e :$
$f : 5, 5$
is a different solution, even though it has the same assignment, but it has a different ordering. The full Simulated Annealing pseudo code in the Valys algorithm is included in algorithm 2.
In our case this will be that a move constitutes that one client is serviced by a different taxi driver. All the other clients are serviced in the same order by the same taxi drivers.

Algorithm 2: Simulated Annealing for Valys Pseudo Code

```
   Data: Customer Requests, epochs
   Result: Route assignment
1  initialization;
2  Let s ← s_0 be the initial solution where every customer is assigned to a different
     taxi;
3  ' T ← epochs;
4  while Temperature T > 0 do
5      Take a customer c from a uniformly random selected driver x_i;
6      n ← 0;
7      while n < 20 do
8          n ← n + 1;
9          Take random other driver x_j and insert customer c in uniform random
             place in x_j's route;
10         if route passes fast checks then
11             if route passes full check then
12                 calculate route cost;
13             if route cost < best route cost then
14                 best route cost ← route cost;
15                 best route ← route;
16                 break;
17             else
18                 if e^((bestroutecost − routecost)/(k·T)) ≥ Random(0, 1) then
19                     best route cost ← route cost;
20                     best route ← route;
21                     break;
22     T ← T − 1;
```

3.3 Checking Feasibility

In this section we discuss the checking of feasibility of a proposal solution in the simulated annealing part of the Valys algorithm. In most Simulated Annealing algorithms this part is quite easy and does not take much time to compute. For example when Simulated Annealing is applied on a simple Vehicle Routing problem the feasibility check only consists of checking whether the capacity of a vehicle is not exceeded. This is matter of adding n elements and checking n times in between addition whether the capacity is exceeded. Here, n is the number of places this vehicle is scheduled to visit. In our problem however we need to check many more constraints. The feasibility check consists of two parts. The first part are the fast checks in Section 3.3.1. These checks are necessary, but not sufficient for feasibility of a particular route. The second part is the full check, which is described in section 3.3.2. For this full feasibility check we use a Time Dependent Simple Temporal Network as described in the paper by Pralet, Cédric and Verfaillie, Gérard [17].

3.3.1 Fast checks

Before the full feasibility check the algorithm does four incomplete feasibility checks. The passing of these checks is necessary for feasibilty, but it is not sufficient to prove that a particular route is feasible. These checks are computed relatively quickly. Since we make no smart guesses for the next proposed solution step, these checks are succesful in removing a lot of in the infeasible proposed solutions. These four checks are:

- checking whether the order of events is right with respect to the pickup windows.
- checking whether the pickup windows are reachable using the real unadjusted travel lengths.
- checking whether the time each person is in a taxi is at most 150% of their direct traveling time.
- checking whether the maximum time, 8 hours, is exceeded per taxidriver.

After these checks is the place were our deep learning models are inserted into the existing Valys algorithm. This will be followed by the full check described in section 3.3.2. The goal of these deep learning models is similar to that of the existing fast checks. Namely to eliminate as much of the remaining infeasible instances while still letting almost all feasible instances through to the full check. This check will not be a necessary check for feasibility; some instances that fail this check will actually be feasible. For more information on the deep learning models see chapter 6.

3.3.2 Full check

In this section we explain the full feasibility check of the existing Simulated Annealing algorithm. Here the algorithm needs to check whether all constraints that are explained in section 2 are adhered to. These can be written as constraints ct in the following form:

$$y - x \geq dmin(x, y) \tag{35}$$

where x and y represent the moment their related events X and Y take place and $dmin(x, y)$ represents the minimal distance these two events have to be apart. Events can be various things such as the picking up of passengers or a zero time moment of the day. When we add all these constraints together we get something called a Simple Temporal Network, STN. In a STN this $dmin(x, y)$ is independent on time. But in our case the distances between events can be dependent on time. To incorporate this we use Time Dependent Simple Temporal Networks, TDSTN. For more information on TDSTN's we refer to the paper by Pralet, Cédric and Verfaillie, Gérard [17]. A graph can be constructed from these constraints, where the events are represented as the nodes and time constraints by labeled directed edges. An example of such a graph can be found in figure 2.

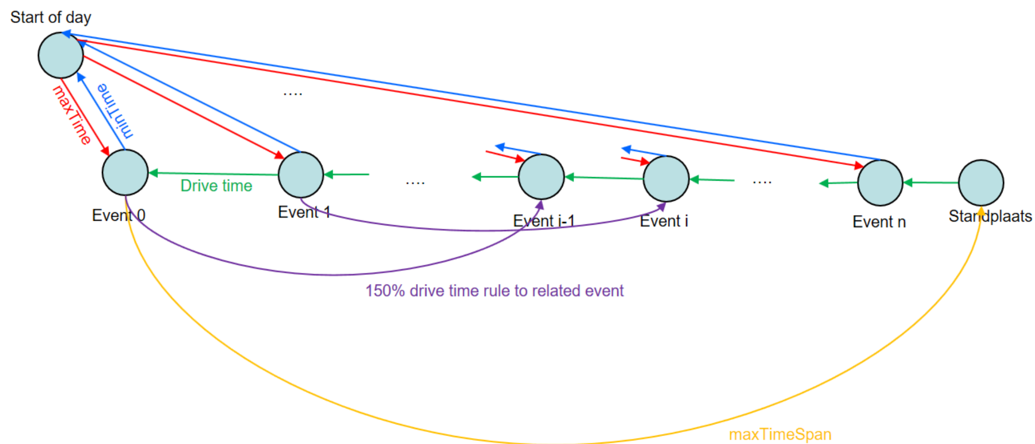

Figure 2: The graph belonging to the STN of the Transvision problem.

We solve these TDSTN using Floyd-Warshalls shortest path algorithm in combination with a smart ordering of edges. For more information on the Floyd-Warshalls algorithm see their paper[6]. For each event in the TDSTN we need a moment in time in which this event can occur. Using the constraints ct we can find time windows for these events. If for an event we ever find an empty time window, then that means there is no solution possible. So we start with very big time window and by applying these constraints we can prune these time windows until no change occurs or we find an empty time window for an event.

First we define the delay function as follows:

$$delay_{ct}(a,b) = a + dmin(a,b) - b \tag{36}$$

As you can see this definition directly comes from equation 35. If $delay_{ct}(a,b) \leq 0$ then the requirement is violated for this pair of (a,b).
A requirement ct is delay monotonic if it adheres to the following two requirements:

$$\forall a, a' \in \mathbf{d}(x), \forall b \in \mathbf{d}(y), (a \leq a') \rightarrow (delay_{ct}(a,b) \leq delay_{ct}(a',b)) \tag{37}$$

$$\forall a \in \mathbf{d}(x), \forall b, b' \in \mathbf{d}(y), (b \leq b') \rightarrow (delay_{ct}(a,b) \geq delay_{ct}(a,b')) \tag{38}$$

Here $\mathbf{d}(x)$ and $\mathbf{d}(y)$ are the domains of events x and y respectively. Informally in our situation rule 37 would imply that when you leave from a location later you cannot arrive earlier than you would before. Rule 38 implies that when you arrive earlier you cannot leave later than you would before. Since our restrictions are all concerned with how cars drive, these restrictions are all delay monotonic.
Now we define $earr_{ct}(a), ldep_{ct}(b), firstNeg(F, \mathbf{d}(x))$ and $lastNeg(F, \mathbf{d}(x))$. $earr_{ct}(a)$ is the earliest arrival time for constraint ct, when the departure is a. $ldep_{ct}(b)$ is the latest departure for constraint ct, when the arrival is a. $firstNeg(F, \mathbf{d}(x))$ is the smallest $a \in \mathbf{d}(x)$, such that $F(a) \leq 0$. $lastNeg(F, \mathbf{d}(x))$ is the largest $a \in \mathbf{d}(x)$ such that $F(a) \leq 0$. How these equations are related can be found in Equations 39 and 40.

$$\forall a \in \mathbf{d}(x), earr_{ct}(a) = firstNeg(delay_{ct}(a, .), (y)) \tag{39}$$

$$\forall b \in \mathbf{d}(y), ldep_{ct}(b) = lastNeg(delay_{ct}(., b), (x)) \tag{40}$$

To find $earr$ and $ldep$ we thus need to find values for $firstNeg$ and $lastNeg$. A possible algorithm for finding the value of $firstNeg$ is stated in Algorithm 3:

Algorithm 3: Possible way of computing $firstNeg(F, I)$, with $I = [a_1, a_2]$, maxIter a maximum number of iterations, and prec a desired precision. [17]

```
1  firstNeg(F, [a_1, a_2], maxIter, prec);
2  begin
3      f_1 ← F(a_1); if f_1 ≤ 0 then
4          return a_1;
5      f_2 ← F(a_2); if f_2 > 0 then
6          return ∞;
7      for i = 1 to maxIter do
8          a_3 = (f_1·a_2 − f_2·a_1) / (f_1 − f_2);
9          f_3 = F(a_3);
10         if f_3 < prec then
11             return a_3;
12         else
13             if f_3 > 0 then
14                 (a_1, f_1) ← (a_3, f_3)
15             else
16                 (a_2, f_2) ← (a_3, f_3)
17     return a_2;
```

Finding the value of $lastNeg$ happens in an analogue way. Using this definition we can now give a simple algorithm with which to update the domains of the time variables, the so called time windows:

Algorithm 4: Rules for pruning the time windows

```
1  d(y) ← d(y) ∩ [earr(min(d(x))), +inf[;
2  d(x) ← d(x)∩] − inf, ldep(max(d(y)))];
```

We can apply these rules indefinitely until no change occurs. A problem with this is that it is possible that the changes are incrementally smaller each time and never stop, but they do converge to an asymptote. But since our model only has integer values and we have bounded time windows, we can not have an infinite sequence of decreasing time windows. So applying the rules from algorithm 4 we eventually find whether the route has a solution. Lastly it is now important to have a smart order in which you prune the constraints. This is because not all constraints have the same direct or indirect influence on all the time windows. So a good ordering is vital to minimize the amount of revisions you have to do.

3.3.3 Acceptance

If our new solution is infeasible, then we reject it and go back to our original solution. If it is feasible we compare the neighbour score with our current best score. If the neighbour is better we continue with the neighbour. If the current score is better, then we continue with the neighbour with a chance. This chance is equal to the following formula [20]:

$$e^{\frac{S_0 - S_1}{T}} > RandUni(0, 1)$$

S_0 is the old score, S_1 is the new score, $RandUni(0, 1)$ is a uniform random number between 0 and 1. Note here that the score is the combined cost of all the taxirides. So a lower score is better. The temperature is a variable that decreases over time during the algorithm. This has as effect that the chance with which we accept worse scores becomes lower over time.

4 Deep Learning

In the previous chapter we discussed the existing Valys algorithm and the mathematical mechanism within that algorithm. In this chapter we go into detail about the mathematical techniques used in the algorithm which we investigated: Deep Learning methods. Deep Learning is a subclass of the machine learning methods as described in [7]. It uses artificial neural networks. Neural networks are nodes connected with each other that models neurons from your brain. These nodes are ordered in layers. Each layer consists of a number of nodes. The nodes in each layer are connected with weighted connections with all nodes in the previous and next layer. There are different kinds of neural networks: supervised, unsupervised or anything in between. For our purpose we use *supervised* Deep Learning models. This means that the model is given guidance into what the output of the model should be. Guidance means that the correct output is known for the given input. In our case, there is enough data available with the correct labels so that is why we use supervised learning.

4.1 Multilayer Perceptrons

Multilayer Perceptrons are a class of Deep Learning models where layers are fully connected with the layers before and after the current layer. This means that every node in a hidden layer has a connection with every node in the next layer. Each node is dependent on every node in the last layer. They are universal function approximaters which is shown by Leshno, Lin, Pinkus and Schocken [11]. This means that in theory you could approximate real valued continuous functions to any degree that you want with Multilayer Perceptrons. This makes them very suitable for situations where a direct formula is unknown, non-existent or expensive to calculate and only an as good as possible answer is needed. In our case we have a time-expensive feasibility function we want to approximate.

4.1.1 Layers, Nodes, Weights and Biases

We differentiate between three layers: input, hidden and output layers. The input layer, which we usually only have one of, is the layer where values from an instance comes into the model. A hidden layer is a layer in the black box part of the model. There, values are stored for further processing and put through to other layers. The output layer is where the output of the model is stored. Every layer in a neural network consists of a number of nodes. Every node receives input from nodes from the previous layer. A linear transformation is applied on the input data of the following form,

$$z_{i,k} = \sum_j w_{i,j,k} \cdot y^{out}_{j,k-1} + b_{i,k}, \tag{41}$$

with $z_{i,k}$ the weighted value of node i in layer k, $w_{i,j,k}$ the so-called weight corresponding to the connection from node j in layer $k-1$ to node i in layer k, $b_{i,k}$ is the so-called bias added to the node $z_{i,k}$ and $y^{out}_{j,k-1}$ the output of the node from layer $k-1$. Now an activation function f can be applied resulting in the output of node i in layer k of the form,

$$y^{out}_{i,k} = f(z_{i,k}). \tag{42}$$

These models need to store their parameters for further use of the model. The models we use store their parameters in tensors. Tensors are three or more dimensional matrices containing numbers. In figure 3 we can see an example of a model with one hidden layer, where every node is fully connected.

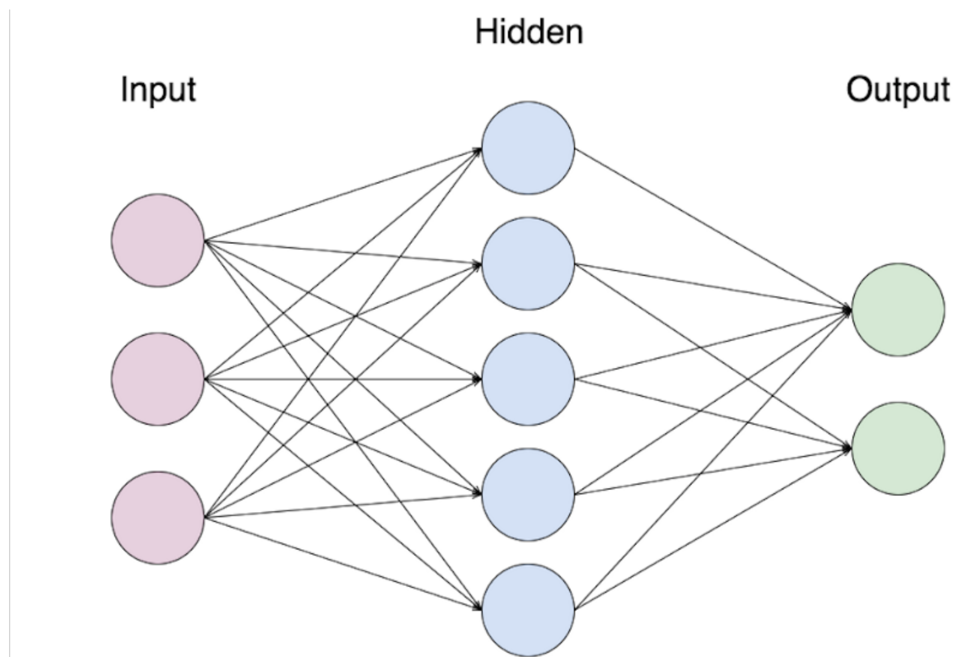

Figure 3: A representation of a Neural Network[2]

4.1.2 Activation Functions

Activation functions are functions that determine if a node passes a signal on and if so, how big the signal is. These functions need to be differentiable everywhere for the back-propagation to work. We explain this backpropagation later in section 4.1.4. Because of

[2]This figure can be found in [2]

this reason we can not use the most natural activation function. This activation function is the following:

$$\text{Binair}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases} \tag{43}$$

The derivative of this function is 0 everywhere and undefined in 0. This means that the backpropagation never changes the weights, since the change is a constant times this derivative which is 0. There are different kinds of activation functions which do derivatives with which we can work. The most used activation functions are Sigmoid and ReLU functions. ReLU(x) is called the rectified linear unit. The function is zero for $x < 0$ and x for $x \geq 0$. So

$$\text{ReLU}(x) = \max(x, 0). \tag{44}$$

This function is differentiable everywhere except $x = 0$. In practice the value for $\frac{d}{dx}\text{ReLU}(x) = 0$ or 1. The Sigmoid function is defined as follows [8]:

$$S(x) = \frac{1}{1 + e^{-x}}. \tag{45}$$

This function is differentiable everywhere, stays between 0 and 1 and has the following nice property for the derivative [8]:

$$S'(x) = S(x)(1 - S(x)). \tag{46}$$

This is a nice property, because it makes it easy to compute the derivative. We already have the value $S(x)$, so computating the derivative computating using equation 46 is relatively easy.

4.1.3 Forward Pass

In a forward pass an instance is fed to the neural network. This instance gets multiplied with all the weights while it passes through the layers. At the end you have some values in the output nodes. In our case this is the prediction that an instance is a feasible route. If the model is in training mode it can be helpful to calculate the gradient of your tensors, because you need this gradient for the backpropagation.

4.1.4 Backpropagation

After the forward pass we want to adjust our model according to how far off we were. We compare our predicted value with the actual label using the loss function that we chose. A loss function is a function that maps its variables to a real number, in our case bounded

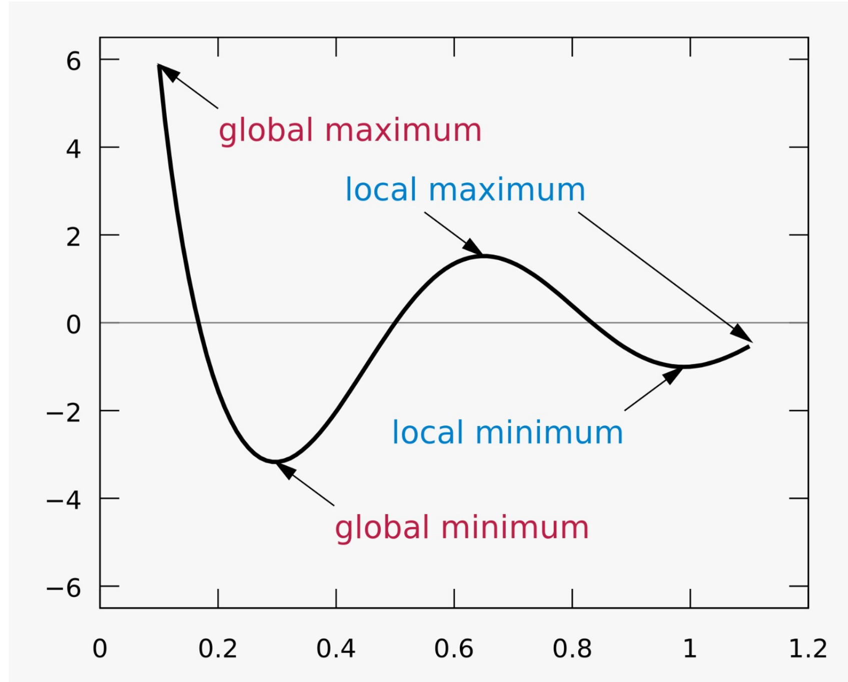

Figure 4: Gradient Descent fins a local minimum and possibly the global minimum[3].

below by 0. This number represents a loss or a cost that you generally want to minimize. A simple loss function is the Mean Squared Error. The MSE has the following formula [22]:

$$MSE(\mathbf{y}, \mathbf{y}^*) = \frac{1}{|\mathbf{y}|} \sum_{i \in [|\mathbf{y}|]} |y_i - y_i^*|^2. \tag{47}$$

Where $\mathbf{y}$ and $\mathbf{y}^*$ are the output vector of the model and label vector respectively. Another loss function which is often used for binary classification is Binary Cross Entropy [14]:

$$BCE(\mathbf{y}, \mathbf{y}^*) = -\frac{1}{|\mathbf{y}|} \sum_{i \in [|\mathbf{y}|]} (y_i \cdot \log(y_i^*) + (1 - y_i) \cdot \log(1 - y_i^*)). \tag{48}$$

From [4] we know that for a binary classification problem the BCE is an appropriate loss function for our models. After that we change the weights and biases in response to faulty label predictions. Gradient Descent is a usefull method for this. Gradient descent calculates the gradients of the loss function in respect to the weights and biases all individually and then change those weights and biases into the opposite direction of the gradient with an appropriate amount. This moves the state of the learnable parameters

towards a local minimum. For this to work, we need to have gradients and thus all the functions in the model need to be differentiable. When the weights and biases reaches the local minimum the gradient is zero, so it stops changing. In Figure 4 a function with a global minimum and local minimum can be seen. With gradient descent we could either end up in the local minimum or the global minimum depending on where we start relative to the local maximum. To cross this local maximum we could use something called momentum. Momentum uses the gradient of the last batch to keep going more steadily in the same direction. This could push the state of the parameters over maxima and into a better minimum.

4.2 Overfitting

Overfitting is when you use too many parameters to try to fit data. This gives you a result that perfectly fits those datapoints, but does not find the underlying connection. This is a problem if you want to use your model to predict something on data it has never seen, so called out-of-sample performance. This is usually the case with Deep Learning models. To rectify this we could use Dropout or Dropconnect. Dropout is a layer you place after normal layers. It puts the activation value of some nodes in that layer to 0 with a preset chance. Dropconnect is a layer you put before a normal layer. It sets some weights of the layer to 0 with a preset probability. To some extent, this ensures that the model does not become too reliant on single nodes for the predicted label.

To check whether you are overfitting we separate our data into 3 groups: train, validation and test data. We use the train data to train our model on. We use the validation data to check when we need to stop with training. If the model stops improving on the validation data we stop as well. When we continue beyond this point, we improve the performance on the training set, but potentially worsen out-of-sample performance as overfitting occurs. We need to use data for this on which the model was not trained, because the model learns specific elements of the train data. This means that the loss on the train data decreases long after the model has stopped improving. Lastly we use test data to give a definite score to the model. We needs this last set, because we implicitly shape our model to the validation data. So this test data is a set of data which the model has completely not seen before and thus suitable for as a score giver.

4.3 Convolutional Neural Networks

A convolutional network is especially useful in situations where you want to detect the same pattern in different parts of your input. For example, you want to detect if a dog is present in a picture. You want to get a positive result whether the dog is in the center, top right or anywhere else in the picture. Convolutional layers use the same weights variables for different nodes in the same layer. Thus, they are also learned together. We could use

this in our project because errors later in a route could also occur earlier in the route. For this approach to work, the input types have to be the same everywhere where we apply this weight-matrix also know as kernel. Otherwise the same weights would not make sense.

4.3.1 Pooling Layers

Pooling layers combine the outputs of subsets of neurons. They act the same as convolutional layers, except that their weights and biases are not learnable. The two pooling layers we use, are maximum pooling layers and average pooling layers. Maximum pooling layers take the maximum of their input nodes. Average pooling layers take the average of their input nodes.

4.4 Normalization layers

To further improve our model we use normalization layers. A normalization layer ensures that the layer after that specific normalization layer gets their input from a certain interval. So it scales the input. This makes learning faster, because bigger learning rates can be used. Without normalization layers this could cause problems with nodes that get high or low activation values, which would slow down or halt training. It also has some small regularization benefits. We used two different normalization layers: Instance normalization and batch normalization. Batch normalization normalizes over the batches and instance normalization normalizes over each input node.

4.5 Type I and Type II errors

An error is when the prediction of an instance by the model is not equal to the label corresponding to that particular instance. For our problem you can divide these errors into two kinds: feasible instances that are predicted infeasible and infeasible instances that are predicted feasible. We call these the ***Type I*** and ***Type II*** errors respectively. Usually you try to optimize your model to reduce all errors, but sometimes Type I errors are more important than Type II errors or vice versa.

The goal of our model is to know if a route-proposal is feasible or infeasible. However, after applying the deep learning models on our input, this does not result in the binary classified output feasible and infeasible. The model outputs a number between zero and one, where the model has trained to try to match their output as closely to the associated labels ”0” and ”1”. A ”0” constitutes an infeasible route-proposal and a ”1” constitutes a feasible route-proposal. Now, a post-processing procedure has to be applied to categorize all the output in feasible and infeasible route-proposals. A natural way to categorize is to say that every output below a threshold $t = 0.5$ means infeasible and

every output equal to or above $t = 0.5$ is feasible. However, you can take any threshold t between 0 and 1, it does not have to be 0.5. For example, take 0.1. Then, every number above 0.1 is feasible, and any number below 0.1 is infeasible. There are multiple reasons why you would want to do this:

1. There can be a situation where you only need the feasible-instances and not the infeasible-instances, and it is important that you have almost all feasible instances. Then, when setting your threshold at 0.5, you possibly miss a lot of feasible - instances, because you discarded everything below 0.5. When you set your threshold at a lower number, you still discard a lot of infeasible instances, but much less feasible instances.

2. The goal is to maximise the accuracy and the maximum accuracy is not achieved at $t = 0.5$.

3. The environment in which your model is deployed has a skewed data distribution and your model has a skewed accuracy table.

In our situation we want to reduce the number of infeasible instances we need to fully check, but certainly not discard some feasible instances. In this project the situation as described in the first item is applicable.

4.6 PyTorch

Deep learning is around for some time, so people have made various libraries in python and other coding languages to make the process of training and deploying models easier. We use PyTorch as our library in python. The library maintains weights, biases and gradients. It also has methods for applying the model and the accompanying backpropagation. Every model consists of different layers applied one after another. These layers could be any of the mentioned layers above or any other implemented layerstructure. Furthermore, there are various optimizers, loss functions and datastructures implemented [10].

5 Data Generation

In Chapter 4 we explained the mathematics behind the deep learning model. For these models to train we need data. In this chapter we explain how we extract our data and how it is going to look. In section 5.1 we explain how much data we extract. How we extract this data and how this data is sampled is explaind in section 5.2. We need to divide this data into three parts. Why and how we do this is explained in section 5.3. In the last section of this chapter we go into detail about how we shape the instances we feed to the models.
However first we need a place to extract our data from. Fortunately enough we can run the existing algorithm on 60 runs of separate days and extract data from those runs. The big question is now: what data do we want? First and foremost we want route instances that passed the fast checks of the existing algorithm 3.3.1. This only leaves us with those that would normally go through the expensive full check. So it would only encounter instances that passed the fast checks. Then we could simply extract all the available information on routes that pass the fast checks and use that for the models. This gives us about 2.7 billion instances where each instance consist of about 500 numbers. This is way too much to store and too much to put through the algorithm. Thus we need to make a selection with enough instances to get a good first training session, but not too few for competent training. Another problem is the differing size of the routes. Models are usually built for one input size. Not all our instances contain the same number of customer requests. More requests per route means more information. So we need a way to solve this problem. We call the number of customer requests per route the route cardinality.

5.1 Data Size

The first idea is to get 10,000 instances per route cardinality, feasibility and day combination. Since we have six cardinalities, two feasibility options and 60 days, this results in a total of $7.2 \cdot 10^6$ instances. For each instance we would store the Route ID's in the order in which they are picked up and dropped of together with whether it was feasible or not. For the actual input to the models we add some information, but we can extract that from this in due time. For example for an route cardinality 2:

$$1, 0776293, 0776293, 0662732, 0662732 \tag{49}$$

where the 1 indicates that this instance is feasible and 0776293 and 0662732 are Route ID's. The example means that the customer 0776293 is first picked up and dropped off, after which customer 0662732 is picked up and then dropped off. This gives approximately 3.5 GB of data, which is big enough in and in itself. We take the same number of instances per feasibility bit, since it is usefull to have a balanced dataset. If you would have more instances which are infeasible, then the model gets pulled more towards giving

an infeasible rating. At the extreme if a model only gets infeasible instances it would just have 0 as output regardless of the input. Even with some feasible instances it would probably always output 0 or something very close to it. When your data is skewed and not numerous, there is a method which uses all the data and mitigates this pulling effect. This method simply exists of taking equal shares from the feasible and infeasible data per training run. The smallest class you take completely and for the bigger class you sample a set with the same size as the smallest class. This way you can use all your data, but you do not train with skewed results.

5.2 Data Sampling

These 10000 instances are taken uniformly over all the instances that the algorithm encounters. We do this uniformly, since we do not have a reason to prioritize different parts of the data stream. In some situations it could be useful to take more from the end of the algorithm, since there, improvement is hard so we really need to test all feasible solutions. Or it might be better to get more instances from the beginning, since there we need to spread out our search for the global minimum as much as possible. In the aforementioned described cases we would assume that the routes tested from the beginning are structurally different from routes tested at the end. Instead we assume the opposite: That the routes from the beginning of the data stream are structurally the same as the routes from the end. We sample this as if we get the data in an infinite stream. For this we use Reservoir Sampling by Vitter [21]. This version has a time complexity of $O(10000(1 + log(n/10000))$, but we use a somewhat simpler version which has time complexity $O(n)$. The pseudo code for the $O(n)$ variant can be found in algorithm 5. We choose to use the $O(n)$ variant, since the time complexity of the rest of the algorithm is $O(n^2)$ or worse. So our algorithm runs with about the same running time with this insertion.

Algorithm 5: Reservoir Sampling Pseudo Code [21]

Data: Datastream d, Reservoir Size n
Result: Sampled set r

```
1 i=0;
2 while True do
3     i++;
4     k=RandomInteger[0,i) if k < n then
5         r[k]=d[i];
6     if End of stream then
7         break;
```

5.3 Train, Validation and Test

When we get our data we need to split our data into 3 sets as mentioned in section 4.2, namely: Train data, Validation data and Test data. The question is how we want to split this data. Since eventually we train our model on data from some days and use it for other days with, in theory, independent routes, we are going to assign whole days of data to each of these 3 sets.
Because of this we do not want to do cross-validation. Cross validation would mean that we would join the train and validation data and choose a different part of this set to use as validation data each epoch. This would mix the day sets and as such is only advisable if you do not have a lot of data. As stated in section 5.1 we use 60 days of data, which gives approximately 1.2 million instances for our models. 60% of this is enough to train our networks on.

5.4 Data processing

When we get the necessary data we need to process before we can feed it into the models. The model can only use linearly scaleable data. For example a route duration of 20 means that it is two times as long as a route duration of 10. However, a route ID of 20 does not mean that it is two times as big as a route ID of 10. It is just a different ID. So these ID's must be transformed into dummy variables. This means that if we have n different ID's we transform each ID into a serie of n booleans. The first ID we encounter, we correspond with $1000\cdots000$. The second with $0100\cdots000$ and so on. These booleans are linearly scale able. This is because a 1 corresponds with the occurrence of a particular ID and a 0 with the absence. So a 0 corresponds with $0 \cdot 1$. We also need some more information about the routes, which we extract using the Route ID's. We need the pickup window for each customer in the route and the maximal length that each customer can be in the car. We use the coordinates of the pickup and dropoff location of every customer. This transformation makes the instance in table 49 into:

Table 1: Processed data of the example data found in equation 49

1						
0	1	43	36900	52.294924	4.9799599	0
0	1	43	36900	52.202983	5.2963768	0
1	0	130	43200	52.241340	5.1651836	0
1	0	130	43200	51.443094	5.6277974	0

This is formatted in a two dimensional array for readability. In the model it is in a one dimensional array. The label of the instance is the only entry in the first row. The

first two columns, except for the first row, correspond to the dummy representation of the tripID's. The third column is the maximum time allotted to corresponding customers. The fourth column is the pickup time for the customer. If the request corresponding to the customer is an arrival guarantee request, then this column is zero at that position. Column five and six correspond to the coordinates of the pick up or drop off location. The two requests were not arrival guarantee request, so the last column is all zeros. After this transformation we separate the label in front from the rest of the data. Now we can train the models on the data.

6 Models for the Valys problem

In this Chapter, we explain the architecture of the models for the Valys problem we trained and tested. For these models we performed hyper-parameter optimization on a defined set of parameters which we explain further in this chapter. This produced 425 models in total for the Route-cardinalities two untill seven combined in the first run. We can divide these models into two types of models: Multi layer perceptron models and Convolutional models. For the first run of these models we used one instance normalization layer.

For the second round we only considered models with a layer depth of one, switched the normalization layer to batch normalization, changed the number of kernels per layer and used only the halving layer width method for the MLP models. This led to 30 models.

6.1 Inputlayer

From the data generation we get instances of cardinality k in the following form:

$$RouteID_0, RouteID_1, \cdots, RouteID_{2k-2}, RouteID_{2k-1} \tag{50}$$

where $RouteID_i \in [k]$ and every element in $[k]$ is visited twice. After every $RouteID$ we add the following five variables: Geo coordinates, Max Route Duration, Pick up window and Arrival guarantee deadline. From the last two exactly one is nonzero, because in the case of an arrival guarantee no pick up window is given and vice versa. After this the $RouteID$'s are transformed into dummy variables. For the case of $k = 2$ there is an example in table 1. In total this has $2 * k * (k + 5)$ variables. For more details on form of the data view chapter 5.

6.1.1 Loss function

As a loss function we use Binary Cross Entropy which we defined in equation 48. This is a good loss function for a binary classification problem as stated in [4].

6.1.2 Optimizer

As optimizer we chose RMSProp. The learning rate is 0.01 with a scheduler that reduces the rate with a factor 0.5 when the loss does not decrease for 6 epochs. We chose these hyper-parameters as such through trial and error.

6.2 Multilayer Perceptron

For the multilayer perceptron models we tried variations in the number of hidden layers and the number of nodes in these layers. The output layer always contain one node, which

is the feasibility prediction of the model. The widths of the other layers we handled in three different ways. We choose to always decrease the widths of the layers to make the execution of the models faster. We decreased it linearly, multiplicatively and divided it by two with a last jump to one. So lets say that n is the number of nodes in the input layer and we have m hidden layers. Then the number of nodes in the linearly decreasing schedule is:

$$n, \left\lfloor \frac{n \cdot m}{m+1} \right\rfloor, \left\lfloor \frac{n \cdot (m-1)}{m+1} \right\rfloor, \cdots, \left\lfloor \frac{n \cdot 1}{m+1} \right\rfloor, 1. \tag{51}$$

For multiplicatively the width schedule would look like this:

$$n, \lfloor n^{\frac{m}{m+1}} \rfloor, \lfloor n^{\frac{m-1}{m+1}} \rfloor, \cdots, \lfloor n^{\frac{2}{m+1}} \rfloor, \lfloor n^{\frac{1}{m+1}} \rfloor, 1 \tag{52}$$

For divide by 2 the width schedule would look like this:

$$n, \left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2^2} \right\rfloor, \cdots, \left\lfloor \frac{n}{2^{m-1}} \right\rfloor, \left\lfloor \frac{n}{2^m} \right\rfloor, 1 \tag{53}$$

For this last one we assumed $n \geq 2^m$, which with our $m \in [4]$ and $n \geq 28$ was always satisfied. Some of these coincided for specific numbers, so there are not $3*4*(7-2+1) = 72$ MLP models, but rather there are 65 models. In the second run we only used the halving method and one hidden layer per Route cardinality. So there are six models in the second run.

6.3 Convolutional models

The convolutional models have more hyperparameters in which they differ. They have the following differing hyperparameters: depth, width, kernel size and average or maximum pooling layers. The depth is here the number of layers. The widths is number of different kernels per layer. The kernel size concerns the size of the of kernels used for the hidden layers after the first hidden layer. For the first hidden layer we use a kernel width dimensions of 2RouteCardinality by 1. If the kernel size is equal to r then the kernel is a square matrix with dimensions $r \times r$. The model uses either average or maximum pooling layers.

The depth ranges from one to four. The width is in the set $\{2, 4, 8\}$ as the starting width. The kernel size is in the set $\{3, 5, 7\}$. In the second set of convolutional models we used the starting width set $\{16, 32\}$. The kernel set becomes irrelevant, because the first hidden layer has a different layer structure than the rest of the layers and is set. We still tried maximum and average pooling layers.

7 Results

In this chapter we state the results of the training of the various models as described in chapter 6. There are 3 different metrics on which we judge these models in chapter 8: Accuracy, execution speed and threshold performance. We subdivided these results into the results for the models with the instance normalization layers in section 7.1 and the results for the models with batch normalization in section 7.2.

7.1 Instance Normalization

7.1.1 Accuracy

In the table 2 we can see that the accuracy is slightly worse for models with higher route cardinality. This is not unexpected, since larger route cardinality instances are complexer and thus more difficult to predict. We also see that the accuracy only increases with the number of hidden layers for cardinality 2. For higher cardinalities it stays about the same or even decreases. This could be because a model with more variables is harder to train.
In table 3 we see slightly better results than in their MLP counterparts in table 2. And here the model's performance does not decrease when the number of layers is increased.

Table 2: Best accuracy percentages for the MLP models.

Route Cardinality	2	3	4	5	6	7
1 Hidden Layer	89.03	88.84	88.51	87.97	87.30	86.25
2 Hidden Layers	90.55	88.75	88.95	87.77	86.95	84.97
3 Hidden Layers	90.56	89.07	87.78	87.46	85.61	85.38
4 Hidden Layers	90.17	88.38	88.59	88.15	84.88	82.69

Table 3: Best accuracy percentages for the Convolutional models.

Route Cardinality	2	3	4	5	6	7
1 Hidden Layer	91.13	89.37	88.63	88.28	87.41	86.54
2 Hidden Layers	91.27	89.38	89.63	88.73	88.27	87.41
3 Hidden Layers	91.25	89.63	89.48	89.15	88.57	87.43
4 Hidden Layers	91.27	89.43	89.82	89.50	88.66	87.66

7.1.2 Execution speed

Each execution speed test is performed on 240000 instances. So the results you see in for example table 4 are in seconds per 240000 instances. For the MLP models you can see in

for example table 4 the execution speed is approximately the same for the different widths. When the number of hidden layers is increased the execution duration does increase. When the Route Cardinality is increased, such as in table 6, you can see that the execution speed also does not increase. This means that when comparing different models on their accuracy only a consideration needs to be taken concerning the number of hidden layers. For the Convolutional models we have the same pattern as with the MLP models as you can see in table 5. What is important to note is that the convolutional layers in general are slower than their MLP counterparts. They are about 20 to 35 percent slower. A note has to be taken that these tests have been done in Python, while the model is deployed in C++, which is a lower level code and therefore expected to evaluate faster. The rest of the execution speed test tables can be found in the Appendix 8.3.

Table 4: Speed test results for the MLP models with route cardinality 2

Route Cardinality 2	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	58.92	58.96	58.97
2 Hidden Layers	76.49	76.18	76.30
3 Hidden Layers	93.04	93.17	92.75
4 Hidden Layers	109.22	105.78	105.27

Table 5: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 7.

Route Cardinality = 7, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.41	71.23	71.31
2 Layers	95.12	95.75	95.26
3 Layers	118.27	118.42	118.07
4 Layers	146.71	147.17	148.00

Table 6: Speed test results for the MLP models with route cardinality 7

Route Cardinality 7	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	57.62	56.80	56.88
2 Hidden Layers	73.53	73.59	73.22
3 Hidden Layers	89.20	89.09	89.14
4 Hidden Layers	104.91	105.21	104.74

7.1.3 Threshold

In figure 5, figure 6, figure 7 and figure 8 we can see the accuracy values for different thresholds. The Accuracy lines represent the accuracy values over all in the instances. The Infeasible Accuracy and Feasible Accuracy lines represent the accuracy values over the infeasible and feasible instances respectively. We are interested in the accuracy values in the deployed environment, so where the ratio between feasible and infeasible instances is different. These values are represented by the Realised Accuracy lines. We tested these models for threshold values in the range of $[0, 1]$ with increments of 0.005. So in total we have 200 data-points per graph. In these graphs we can see that we can achieve a very high accuracy, $\geq 99.50\%$, for the feasible instances, while still achieving a reasonable accuracy, around 65% for the infeasible instances. All graphs for the instancenorm models can be found in the Appendix 8.3.

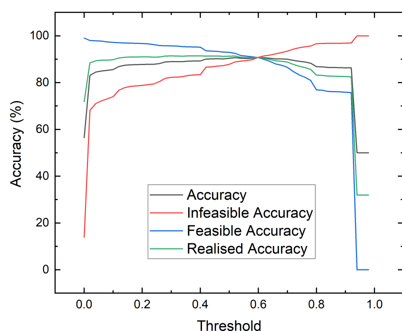

Figure 5: Accuracy for different thresholds where inputsize$= 2$

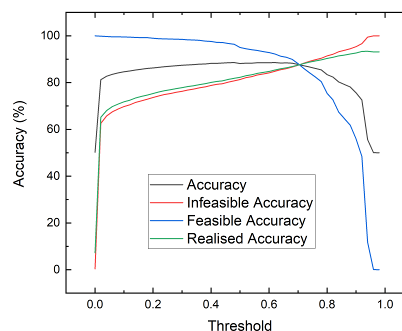

Figure 6: Accuracy for different thresholds where inputsize$= 7$

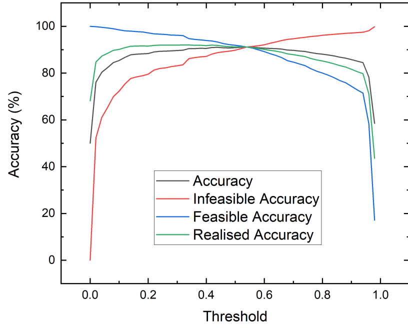

Figure 7: Accuracy for different thresholds where inputsize= 2 and convolutional

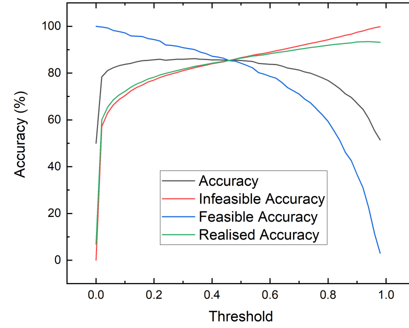

Figure 8: Accuracy for different thresholds where inputsize= 7 and convolutional

7.2 Batch Normalization

7.2.1 Accuracy

The accuracy is much higher with batch normalization versus instance normalization. The increase is around 4 percentage points. Convolutional layer models are oddly enough worse than their MLP counterparts. Except for the case of routecardinality 2. Even though the convolutional models were better than the MLP models in the instance normalization case.

Table 7: Best accuracy percentages for the batch normalization models.

Route Cardinality	2	3	4	5	6	7
MLP model	95.06	94.25	94.29	93.99	93.34	93.98
Conv model	95.15	92.30	91.70	90.54	91.38	92.97

7.2.2 Execution Speed

These speed tests on these models are done in the same way as described in section 7.1.2. The speed overall is about 10 percent slower in comparison to the Instancenorm counterparts in section 7.1.2 as can be seen in the tables below. The speed difference when increasing the number of layers is the same as with the instance norm case. The difference when switching to the convolutional models is also similair to that of the instance norm.

Table 8: Best speed percentages for the batch normalization models.

Batch normalization	MLP	Max Pool16	Avg Pool16	Max Pool32	Avg Pool32
Route Cardinality 2	57.95	74.51	71.36	73.64	71.13
Route Cardinality 3	57.76	73.40	71.38	73.58	71.42
Route Cardinality 4	58.26	74.12	71.27	74.02	71.03
Route Cardinality 5	58.05	74.67	70.88	74.13	70.87
Route Cardinality 6	57.76	74.43	70.90	74.89	70.85
Route Cardinality 7	58.14	74.46	71.12	74.86	70.97

7.2.3 Sensitivity Analysis

In figure 9 and figure 10 we see the results of the best batch normalization models for inputsize two and seven respectively with different threshold values. In section 7.1.3 we explained what the different lines represent. Threshold results for lower numbers are very promising. For a route cardinality of 2 at 0.005 we have 99.97% accuracy for feasible instances and still 72.74% accuracy for infeasible instances. What the preferred values for the threshold is, can not be directly stated from this data. The different threshold values should be tested in the entire route finding algorithm to find the best one. This is because the effect of discarding feasible solutions is hard or even impossible to quantify.

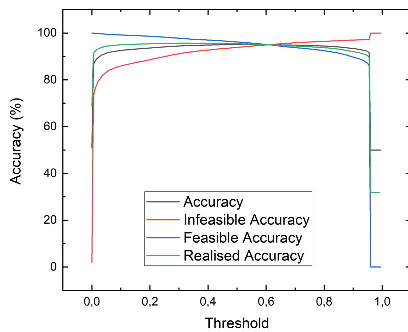

Figure 9: Accuracy for different thresholds where inputsize= 2

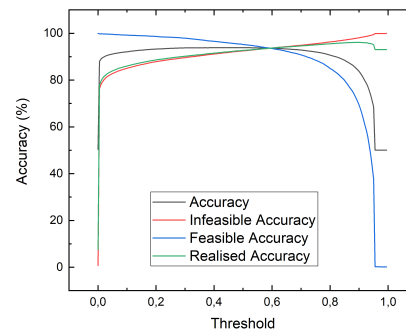

Figure 10: Accuracy for different thresholds where inputsize= 7

8 Discussion and Conclusion

In this chapter we discuss the results from Chapter 7 in section 8.1. Afterwards we give a conclusion in section 8.2. We end this chapter with a list of possible further research in section 8.3.

8.1 Discussion of the various models

Here we briefly remark the findings in chapter 7 and point out some interesting findings.

8.1.1 Accuracy

In section 7.1.1 we can see that for the instance normalization models we can reach around $87 - 91\%$ accuracy. For the batch normalization models in section 7.2.1 we see that we can achieve substantially higher accuracy around $93 - 95\%$. This means that so long the execution speed of the batch normalization models is not substantially lower than that of the instance normalization models we should use the batch normalization models. Oddly the convolution models are almost all worse than their MLP counterparts with the same number of hidden layers when using the batch normalization models. One thing to note is that we used coordinates of the addresses instead of the direct distances. Were we to use the direct distances, still without the dynamic adjustments, we could improve on the models a lot. We did not use these, since we thought accessing this data would take too much time. From the models with input cardinality 2, which should be a really easy problem, we can see that we have a lot of room for improvement. If we used the direct distances we could probably get very close to the 100 percent.

8.1.2 Execution Speed

From the results in sections 7.1.2 and 7.2.2 we can find that the convolutional models are substantially slower than their MLP counterparts. Since in section 8.1.1 we stated that their accuracy was also lacking we can state that the batch normalization MLP models are the ones we want to use. Sadly we also saw in 8 that all models are too slow for 300.000.000 iterations. The fastest model would execute in about 17 hours, but the entire algorithm can not take longer than 4 hours. Note that this speed testing has been done in Python with a Pytorch package, with is expected to be much faster in C++. To fully understand whether the deep learning model is fast enough, the model evaluation should be benchmarked versus the current full checks.

8.1.3 Sensitivity Analysis

For the batch normalization MLP models we saw in section 7.2.3 that the results for low threshold values were still good. At a threshold value of 0.05 we could still filter out $72 - 78\%$ of the infeasible solutions, while maintaining $99.79 - 99.93\%$ of the feasible solutions. So if these models prove to be fast enough in a C++ implementation, we could improve the entire Valys Algorithm.

8.2 Conclusion

The batch normalization and full layer variant seem superior to their instance normalization or convolutional layer counterparts. For the precise value of the threshold it should be looked at the implementation what the result is on the actual outcome. Especially considering the speed of the models. The results are also very promising for higher route cardinalities than 7. So, there could be some gains to train models for higher route cardinalities than 7. We set out to find a fast accurate deep learning model to help the existing simulated annealing model find better solutions to a variant of the VRPTW. The models we found are fairly accurate, but in Python they are too slow. It could be the case that is is only an issue in Python, but not if we would implement it in C++.

8.3 Further research

For further research there are a few avenues left open. We mention and discuss these briefly below.

- **Testing this framework in different simulated annealing settings.** CQM also applies simulated annealing for pickup and delivery problems around gasoline stations, distribution centers, among others. The same principles of expensive neighbourhoods could apply there and the deep learning framework is flexible enough to cover other situations as well.
- **Removing the Fast checks and retrain the deep learning models.** The Fast checks are there just to swiftly remove a number of infeasible route solutions. They fulfill in that sense the same role as the deep learning model. If we remove the fast checks and retrain the deep learning model for the new input range, we could do more iterations. This could help solve the issue of the lacking speed of the deep learning model. Careful consideration should be taken for the choice for the input data. For when naively we would take the same approach as in 5.2, it is possible we overtrain on the instances which would not pass the fast checks. This is because a lot more instances get rejected by the fast checks than that are passed by them.

- **Use the static distance between visiting places.** At the moment, the deep learning models use the coordinates of the places that are to be visited, since accessing that data is faster than retrieving the static distances. However, this leads to some information loss. If we were to use the static distances, we could maybe significantly improve the models with only a relative small loss in execution speed.

- **Interpret the output of the deep learning model as a probability of feasibility.** The models compare the output of the deep learning models against a preset threshold value and assign a prediction of infeasible or feasible according to whether the output is bigger or smaller. Since the deep learning models are deterministic we always get the same prediction for the same instances. This means that some feasible instances are excluded from being predicted feasible. This could lead to the exclusion of certain important parts of the solution field. If we would instead interpret the outcome of the model as an percentage chance that an instance is feasible, we would exclude fewer parts of the solution field.

- **Converting the models into C++ and merging the models in the existing model.** To truly know whether the deep learning models could be an improvement we would need to implement them into the existing algorithm. For this to work the models also need to be implemented in C++.

- **Removing more layers for increase in speed.** The models could be lacking in execution speed, so a possible solution for that could be removing more layers from the model. This would result in a loss of accuracy, but the upside of increased speed could be worth it. The most promising layers to be removed are: ReLU activation functions and the dropout layers.

- **Implement the deep learning models without packages.** It could be that the models would be faster if instead of using packages, the models were implemented directly.

- **Further hyperparameter optimization.** Not all possibilities for hyperparameter optimization have been exhausted. One could look for example further into the shapes of the layers and into Long Short-term memory networks.

- **Threshold testing in the implemented model.** We have done testing for different threshold values. Which threshold is best is hard to say, without knowing what kind of effect it has on the complete model. Testing this would give a better idea of what the optimal threshold value would be.

- **Making models for higher route cardinalities.** Since the calculation of the feasibility is more time consuming for routes of higher cardinalities, it would be interesting to train models for cardinalities higher than 7. The data as we generated it

had a lot of duplicates and was low on the number of feasible instances for these cardinalities. So this would have to be solved for this to work.

- **Adjusting the Simulated Annealing algorithm.** The models are slow when we put instance one by one through the models. But it can handle a lot of instances in parallel. If we could alter the existing algorithm such that it would feed the deep learning models multiple instances at the same time, that could help in solving the execution speed issue.

References

[1] How to solve vehicle routing problems: Route optimization software and their apis. *https://www.altexsoft.com/blog/business/how-to-solve-vehicle-routing-problems-route-optimization-software-and-their-apis/0*, 2020. [Online; accessed 26-May-2020].

[2] M. Chan. Classical neural network: What really are nodes and layers? *https://towardsdatascience.com/classical-neural-network-what-really-are-nodes-and-layers-ec51c6122e09 [Accessed 10 Augustus 2020]*, 2020.

[3] Y. Chauvin and D. E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology press, 1995.

[4] F. Chollet. *Deep Learning with Python Video Edition*. Manning Publications, 2017.

[5] M. Desrochers, J. Desrosiers, and M. M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342–354, 1992.

[6] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[8] J. Han and C. Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*, pages 195–201. Springer, 1995.

[9] W. Joe and H. C. Lau. Deep reinforcement learning approach to solve dynamic vehicle routing problem with stochastic customers. In J. C. Beck, O. Buffet, J. Hoffmann, E. Karpas, and S. Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 394–402. AAAI Press, 2020.

[10] N. Ketkar. Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer, 2017.

[11] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993.

[12] B. Lin, B. Ghaddar, and J. Nathwani. Deep reinforcement learning for electric vehicle routing problem with time windows. *CoRR*, abs/2010.02068, 2020.

[13] J. Lysgaard. Clarke & wright's savings algorithm. *Department of Management Science and Logistics, The Aarhus School of Business*, 44, 1997.

[14] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[15] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 9861–9871, 2018.

[16] B. Peng, J. Wang, and Z. Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. *CoRR*, abs/2002.03282, 2020.

[17] C. Pralet and G. Verfaillie. Time-dependent simple temporal networks. In *International Conference on Principles and Practice of Constraint Programming*, pages 608–623. Springer, 2012.

[18] M. W. Savelsbergh. The vehicle routing problem with time windows: Minimizing route duration. *ORSA journal on computing*, 4(2):146–154, 1992.

[19] P. Toth and D. Vigo. The vehicle routing problem (siam, philadelphia). *Monographs on Discrete Mathematics and Applications, Philadelphia*, 2002.

[20] P. J. Van Laarhoven and E. H. Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[21] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, Mar. 1985.

[22] Z. Wang and A. C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117, 2009.

[23] L. A. Wolsey. *Integer programming*, volume 52. John Wiley & Sons, 1998.

Appendices

A Speed test tables

Table 9: Speed test results for the MLP models with route cardinality 2

Route Cardinality 2	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	58.92	58.96	58.97
2 Hidden Layers	76.49	76.18	76.30
3 Hidden Layers	93.04	93.17	92.75
4 Hidden Layers	109.22	105.78	105.27

Table 10: Speed test results for the MLP models with route cardinality 3

Route Cardinality 3	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	56.93	57.16	57.10
2 Hidden Layers	73.49	73.87	73.88
3 Hidden Layers	91.12	89.90	89.55
4 Hidden Layers	105.17	106.10	105.68

Table 11: Speed test results for the MLP models with route cardinality 4

Route Cardinality 4	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	56.99	56.84	57.22
2 Hidden Layers	73.55	73.66	73.73
3 Hidden Layers	89.79	89.60	90.47
4 Hidden Layers	105.07	104.60	105.31

Table 12: Speed test results for the MLP models with route cardinality 5

Route Cardinality 5	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	56.75	56.68	57.04
2 Hidden Layers	73.27	73.39	73.30
3 Hidden Layers	89.52	89.40	89.53
4 Hidden Layers	105.09	104.79	105.00

Table 13: Speed test results for the MLP models with route cardinality 6

Route Cardinality 6	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	56.79	56.78	56.59
2 Hidden Layers	73.27	73.39	73.26
3 Hidden Layers	89.18	89.06	89.07
4 Hidden Layers	105.05	104.45	104.64

Table 14: Speed test results for the MLP models with route cardinality 7

Route Cardinality 7	Width Halving	Width Linear Decrease	Width Power Decrease
1 Hidden Layer	57.62	56.80	56.88
2 Hidden Layers	73.53	73.59	73.22
3 Hidden Layers	89.20	89.09	89.14
4 Hidden Layers	104.91	105.21	104.74

Table 15: Speed test results for the convolutional models with maximum pooling layers, kernel size 3 and route cardinality 2.

Route Cardinality = 2, max pool and kernel = 3	# kernel = 2	# kernel = 4	# kernel = 8
1 Layer	73.87	74.32	74.47
2 Layers	94.94	95.05	94.60
3 Layers	114.64	116.05	115.00
4 Layers	147.36	147.46	146.73

Table 16: Speed test results for the convolutional models with maximum pooling layers, kernel size 3 and route cardinality 3.

Route Cardinality = 3, max pool and kernel = 3	# kernel = 2	# kernel = 4	# kernel = 8
1 Layer	73.39	73.71	73.39
2 Layers	95.32	94.79	94.46
3 Layers	115.00	114.23	114.71
4 Layers	146.83	146.47	146.58

Table 17: Speed test results for the convolutional models with maximum pooling layers, kernel size 3 and route cardinality 4.

Route Cardinality = 4, max pool and kernel = 3	# kernel = 2	# kernel = 4	# kernel = 8
1 Layer	73.59	73.71	74.71
2 Layers	94.30	94.53	94.40
3 Layers	114.49	114.11	114.20
4 Layers	142.41	142.76	143.42

Table 18: Speed test results for the convolutional models with maximum pooling layers, kernel size 3 and route cardinality 5.

Route Cardinality = 5, max pool and kernel = 3	# kernel = 2	# kernel = 4	# kernel = 8
1 Layer	73.60	73.65	73.56
2 Layers	96.32	94.87	94.81
3 Layers	114.55	114.39	116.75
4 Layers	142.60	142.73	143.66

Table 19: Speed test results for the convolutional models with maximum pooling layers, kernel size 3 and route cardinality 6.

Route Cardinality = 6, max pool and kernel = 3	# kernel = 2	# kernel = 4	# kernel = 8
1 Layer	73.98	73.70	73.61
2 Layers	94.38	94.63	94.25
3 Layers	114.53	114.08	114.32
4 Layers	142.13	142.53	143.05

Table 20: Speed test results for the convolutional models with maximum pooling layers, kernel size 3 and route cardinality 7.

Route Cardinality = 7, max pool and kernel = 3	# kernel = 2	# kernel = 4	# kernel = 8
1 Layer	74.34	73.83	73.93
2 Layers	95.19	95.51	94.46
3 Layers	114.86	114.83	114.68
4 Layers	142.57	143.00	142.96

Table 21: Speed test results for the convolutional models with average pooling layers, kernel size 3 and route cardinality 2.

Route Cardinality = 2, avg pool and kernel = 3	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.11	71.35	71.78
2 Layers	92.28	92.69	92.41
3 Layers	112.18	112.36	113.16
4 Layers	141.96	142.13	142.13

Table 22: Speed test results for the convolutional models with average pooling layers, kernel size 3 and route cardinality 3.

Route Cardinality = 3, avg pool and kernel = 3	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.00	71.17	71.23
2 Layers	92.96	92.36	93.14
3 Layers	112.44	112.00	112.48
4 Layers	142.56	141.95	141.78

Table 23: Speed test results for the convolutional models with average pooling layers, kernel size 3 and route cardinality 4.

Route Cardinality = 4, avg pool and kernel = 3	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.31	71.21	72.08
2 Layers	92.36	92.11	92.20
3 Layers	113.68	111.83	111.94
4 Layers	137.49	137.55	138.19

Table 24: Speed test results for the convolutional models with average pooling layers, kernel size 3 and route cardinality 5.

Route Cardinality = 5, avg pool and kernel = 3	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.17	71.24	71.15
2 Layers	92.11	92.52	92.51
3 Layers	112.45	112.12	112.22
4 Layers	137.90	137.70	137.90

Table 25: Speed test results for the convolutional models with average pooling layers, kernel size 3 and route cardinality 6.

Route Cardinality = 6, avg pool and kernel = 3	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.16	71.22	71.42
2 Layers	93.39	92.03	92.46
3 Layers	111.98	111.54	111.91
4 Layers	137.48	137.61	138.19

Table 26: Speed test results for the convolutional models with average pooling layers, kernel size 3 and route cardinality 7.

Route Cardinality = 7, avg pool and kernel = 3	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.30	71.27	71.40
2 Layers	92.15	93.49	91.92
3 Layers	112.54	112.59	112.61
4 Layers	137.60	137.48	137.61

Table 27: Speed test results for the convolutional models with maximum pooling layers, kernel size 5 and route cardinality 2.

Route Cardinality = 2, max pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	74.32	74.00	73.74
2 Layers	97.49	97.75	98.70
3 Layers	121.17	120.30	120.44
4 Layers	150.67	151.44	152.12

Table 28: Speed test results for the convolutional models with maximum pooling layers, kernel size 5 and route cardinality 3.

Route Cardinality = 3, max pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.56	73.54	73.96
2 Layers	97.27	97.60	97.42
3 Layers	120.01	121.27	121.43
4 Layers	150.66	151.68	152.20

Table 29: Speed test results for the convolutional models with maximum pooling layers, kernel size 5 and route cardinality 4.

Route Cardinality = 4, max pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.69	73.53	73.50
2 Layers	97.11	97.33	97.79
3 Layers	119.41	120.17	120.23
4 Layers	150.34	151.30	151.63

Table 30: Speed test results for the convolutional models with maximum pooling layers, kernel size 5 and route cardinality 5.

Route Cardinality = 5, max pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	74.13	73.55	73.77
2 Layers	97.45	97.70	97.63
3 Layers	119.66	120.24	121.30
4 Layers	150.68	151.74	152.33

Table 31: Speed test results for the convolutional models with maximum pooling layers, kernel size 5 and route cardinality 6.

Route Cardinality = 6, max pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.64	73.75	73.56
2 Layers	97.27	97.65	97.52
3 Layers	119.97	120.53	120.53
4 Layers	150.79	151.19	151.58

Table 32: Speed test results for the convolutional models with maximum pooling layers, kernel size 5 and route cardinality 7.

Route Cardinality = 7, max pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.81	74.05	73.95
2 Layers	97.19	97.27	97.70
3 Layers	120.31	120.39	120.59
4 Layers	150.94	151.91	152.08

Table 33: Speed test results for the convolutional models with average pooling layers, kernel size 5 and route cardinality 2.

Route Cardinality = 2, avg pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.33	71.56	71.68
2 Layers	94.89	96.02	95.11
3 Layers	117.20	118.06	118.59
4 Layers	146.25	146.82	146.95

Table 34: Speed test results for the convolutional models with average pooling layers, kernel size 5 and route cardinality 3.

Route Cardinality = 3, avg pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	70.99	71.20	71.66
2 Layers	95.12	95.13	94.80
3 Layers	117.83	118.27	118.12
4 Layers	146.43	147.04	147.22

Table 35: Speed test results for the convolutional models with average pooling layers, kernel size 5 and route cardinality 4.

Route Cardinality = 4, avg pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.43	71.36	72.39
2 Layers	94.49	95.03	94.95
3 Layers	117.44	117.88	117.65
4 Layers	145.65	146.57	147.03

Table 36: Speed test results for the convolutional models with average pooling layers, kernel size 5 and route cardinality 5.

Route Cardinality = 5, avg pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.13	71.54	71.25
2 Layers	94.86	95.99	95.66
3 Layers	117.51	118.06	118.16
4 Layers	145.86	146.90	147.39

Table 37: Speed test results for the convolutional models with average pooling layers, kernel size 5 and route cardinality 6.

Route Cardinality = 6, avg pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.34	71.39	71.41
2 Layers	94.64	95.08	95.17
3 Layers	117.41	118.13	118.80
4 Layers	145.96	146.80	147.06

Table 38: Speed test results for the convolutional models with average pooling layers, kernel size 5 and route cardinality 7.

Route Cardinality = 7, avg pool and kernel = 5	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.24	71.32	71.34
2 Layers	94.88	94.98	95.39
3 Layers	118.19	118.66	118.31
4 Layers	146.84	147.34	147.56

Table 39: Speed test results for the convolutional models with maximum pooling layers, kernel size 7 and route cardinality 2.

Route Cardinality = 2, max pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.90	74.20	73.76
2 Layers	97.76	97.44	97.72
3 Layers	119.42	120.40	120.98
4 Layers	151.02	151.38	152.33

Table 40: Speed test results for the convolutional models with maximum pooling layers, kernel size 7 and route cardinality 3.

Route Cardinality = 3, max pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.79	74.08	74.08
2 Layers	97.24	97.40	97.59
3 Layers	119.56	120.39	121.40
4 Layers	150.43	151.51	152.30

Table 41: Speed test results for the convolutional models with maximum pooling layers, kernel size 7 and route cardinality 4.

Route Cardinality = 4, max pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.63	73.54	73.46
2 Layers	97.19	97.27	97.49
3 Layers	119.78	120.01	121.42
4 Layers	150.08	152.93	152.03

Table 42: Speed test results for the convolutional models with maximum pooling layers, kernel size 7 and route cardinality 5.

Route Cardinality = 5, max pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.70	73.90	74.22
2 Layers	98.27	99.56	97.89
3 Layers	121.67	120.27	120.53
4 Layers	150.88	152.84	153.56

Table 43: Speed test results for the convolutional models with maximum pooling layers, kernel size 7 and route cardinality 6.

Route Cardinality = 6, max pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.78	73.66	73.96
2 Layers	97.47	98.38	97.74
3 Layers	121.40	120.89	120.16
4 Layers	150.78	152.09	152.51

Table 44: Speed test results for the convolutional models with maximum pooling layers, kernel size 7 and route cardinality 7.

Route Cardinality = 7, max pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	73.90	74.17	73.83
2 Layers	97.34	98.39	97.77
3 Layers	120.04	120.81	120.20
4 Layers	151.84	152.38	152.61

Table 45: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 2.

Route Cardinality = 2, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.59	71.64	71.33
2 Layers	95.19	95.54	95.00
3 Layers	118.20	117.74	118.33
4 Layers	146.31	146.77	146.74

Table 46: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 3.

Route Cardinality = 3, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.86	71.14	71.35
2 Layers	94.76	95.07	94.93
3 Layers	119.64	117.95	118.19
4 Layers	145.82	146.70	147.23

Table 47: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 4.

Route Cardinality = 4, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.16	71.18	71.29
2 Layers	94.64	95.86	95.57
3 Layers	118.71	117.79	118.11
4 Layers	146.12	146.99	147.38

Table 48: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 5.

Route Cardinality = 5, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	72.68	71.36	71.26
2 Layers	95.31	95.75	95.68
3 Layers	119.21	118.67	118.88
4 Layers	146.26	147.03	147.62

Table 49: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 6.

Route Cardinality = 6, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.36	71.26	71.16
2 Layers	94.82	95.27	94.78
3 Layers	117.92	117.45	117.46
4 Layers	146.54	147.09	147.81

Table 50: Speed test results for the convolutional models with average pooling layers, kernel size 7 and route cardinality 7.

Route Cardinality = 7, avg pool and kernel = 7	start# kernel = 2	start# kernel = 4	start# kernel = 8
1 Layer	71.41	71.23	71.31
2 Layers	95.12	95.75	95.26
3 Layers	118.27	118.42	118.07
4 Layers	146.71	147.17	148.00

B Threshold Graphs

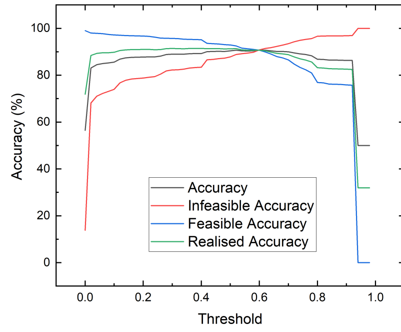

Figure 11: Accuracy for different thresholds where Route Cardinality$= 2$

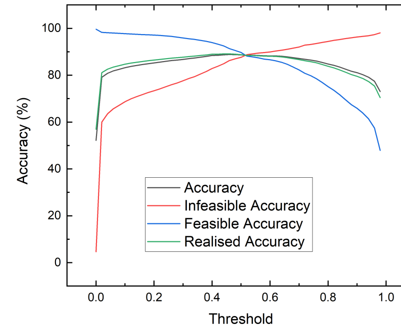

Figure 12: Accuracy for different thresholds where Route Cardinality$= 3$

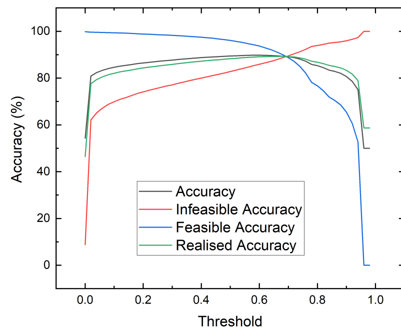

Figure 13: Accuracy for different thresholds where Route Cardinality$= 4$

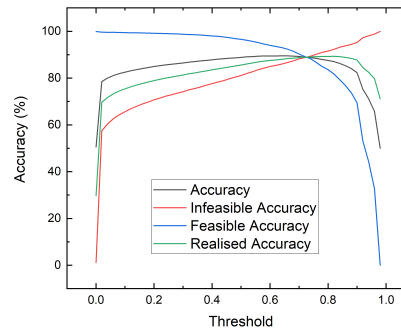

Figure 14: Accuracy for different thresholds where Route Cardinality$= 5$

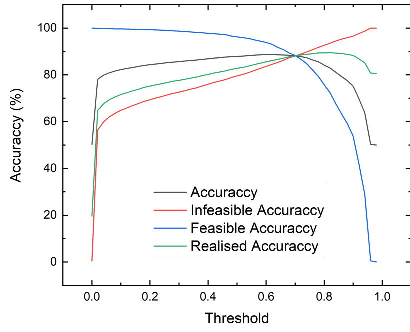

Figure 15: Accuracy for different thresholds where Route Cardinality$= 6$

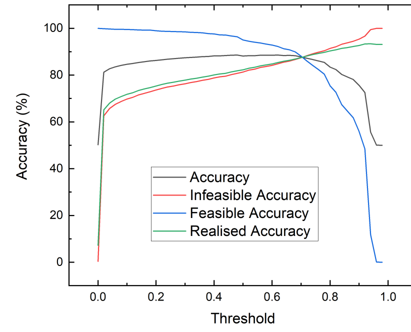

Figure 16: Accuracy for different thresholds where Route Cardinality$= 7$

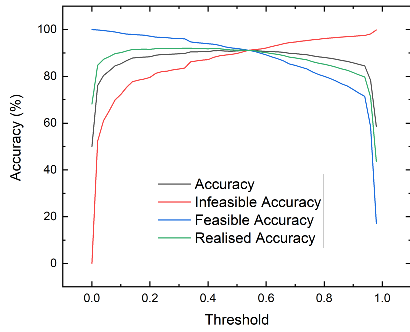

Figure 17: Accuracy for different thresholds where Route Cardinality$= 2$ and convolutional

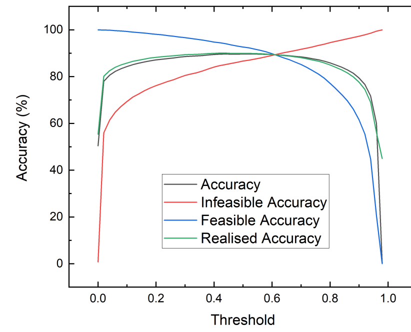

Figure 18: Accuracy for different thresholds where Route Cardinality$= 3$ and convolutional

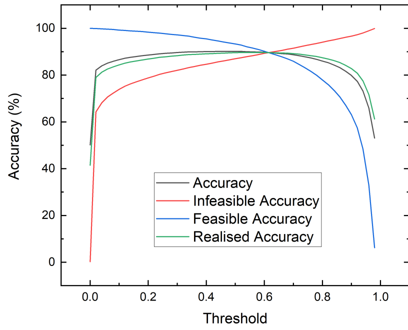

Figure 19: Accuracy for different thresholds where Route Cardinality= 4 and convolutional

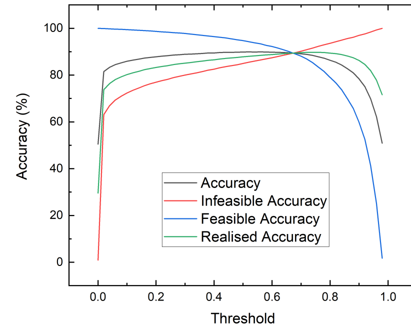

Figure 20: Accuracy for different thresholds where Route Cardinality= 5 and convolutional

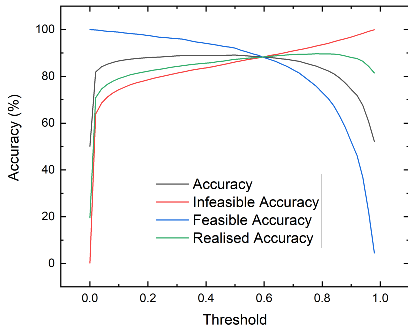

Figure 21: Accuracy for different thresholds where Route Cardinality= 6 and convolutional

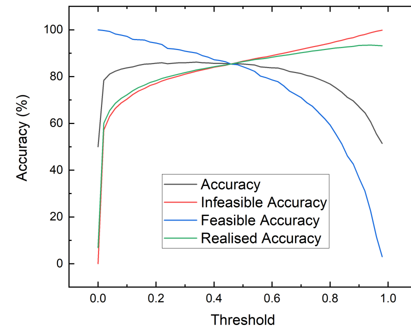

Figure 22: Accuracy for different thresholds where Route Cardinality= 7 and convolutional

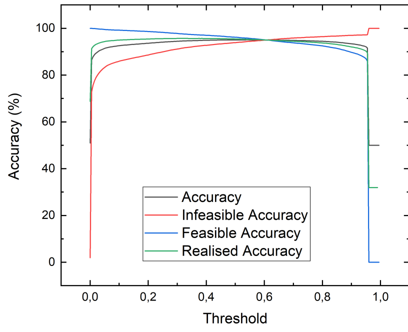

Figure 23: Accuracy for different thresholds where Route Cardinality$= 2$

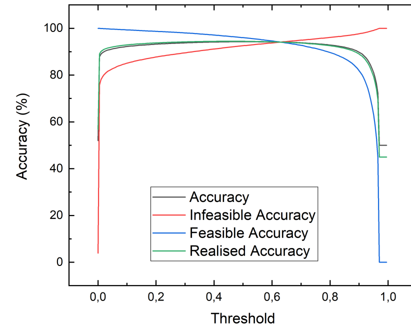

Figure 24: Accuracy for different thresholds where Route Cardinality$= 3$

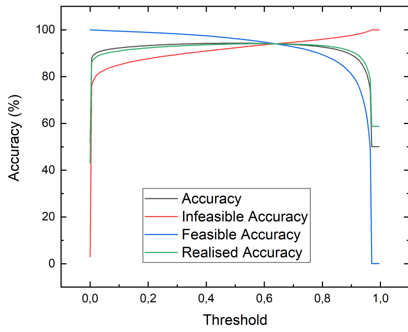

Figure 25: Accuracy for different thresholds where Route Cardinality$= 4$

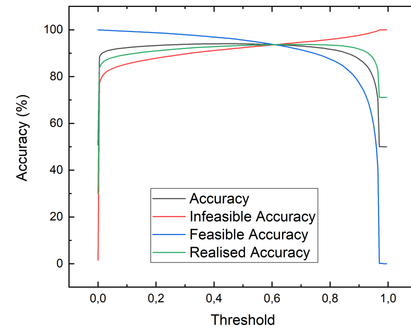

Figure 26: Accuracy for different thresholds where Route Cardinality$= 5$

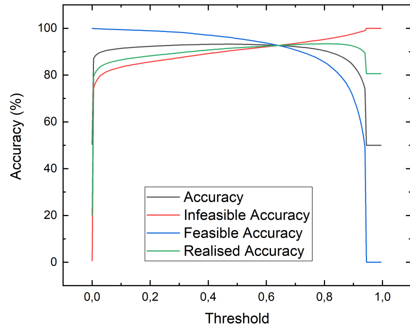

Figure 27: Accuracy for different thresholds where Route Cardinality$= 6$

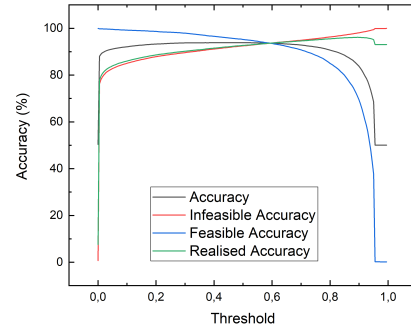

Figure 28: Accuracy for different thresholds where Route Cardinality$= 7$