Finding Solutions to the Vehicle Routing Problem using a Graph Neural Network

Fredrik Hagström

School of Science

Bachelor's thesis
Espoo 21.01.2022

Supervisor and advisor

Prof. Fabricio Oliveira

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the bachelor's thesis

Author Fredrik Hagström

Title Finding Solutions to the Vehicle Routing Problem using a Graph Neural Network

Degree programme Engineering Physics and Mathematics

Major Mathematics and Systems Sciences **Code of major** SCI3029

Teacher in charge and advisor Prof. Fabricio Oliveira

Date 21.01.2022 **Number of pages** 23+3 **Language** English

Abstract

The Vehicle Routing Problem is a famous problem in combinatorial optimization that aims to find an optimal delivery route for a set of customers. It is an NP-hard problem, and so finding exact solutions is computationally demanding and frequently intractable. Classically, many hand-tailored approximate algorithms have been developed to find near-optimal solutions in reasonable time. However, it is challenging to develop new improved algorithms since it requires expert domain knowledge. With recent development in deep learning models, it is enticing to incorporate these in the field of combinatorial optimization to automatically learn improved algorithms. Another attractive aspect of deep learning models are their fast computation times, potentially enabling application to even greater problem sizes than current solvers can handle.

In this thesis, we present a supervised deep learning framework for obtaining approximate solutions to the Vehicle Routing Problem on 2D Euclidean graphs. We implement a Graph Neural Network that learns a probabilistic representation of the solution space to the problem. This representation is converted into a valid solution using a beam search decoder. The beam search procedure is parallelized, allowing for fast search over the solution space. The performance of our model is evaluated on three different problem sizes. The network is trained on sets of near-optimal solutions obtained using the OR-tools routing solver.

The model manages to produce decent results on small problem sizes with 20 nodes, finding solutions under 5 % from the target on average. Applying the model to larger problem sizes proves challenging, as it struggles to find good solutions for problems with 50 nodes, being over 100 % worse than the target on average. Another weakness of the model is its inability to generalize to problem sizes not seen during training. This restricts our framework to trivially small problem sizes efficiently solvable by standard solvers.

Keywords Vehicle Routing Problem, deep learning, Recurrent Relational Network, beam search, average optimality gap

Aalto University
School of Science

Aalto-universitetet, PB 11000, 00076 AALTO
www.aalto.fi
Sammandrag av kandidatarbetet

Författare Fredrik Hagström

Titel Lös ruttplaneringsproblemet med hjälp av ett grafneuronnät

Utbildningsprogram Teknisk fysik och matematik

Huvudämne Matematik och systemvetenskaper **Huvudämnets kod** SCI3029

Ansvarslärare Prof. Fabricio Oliveira

Datum 21.01.2022 **Sidantal** 23+3 **Språk** Engelska

Sammandrag

Ruttplaneringsproblemet (Vehicle Routing Problem), är ett välkänt problem inom kombinatorisk optimering, vilket ämnar hitta en optimal leveransrutt för en given mängd kunder. Problemet är NP-hårt, vilket innebär att det inte finns en känd algoritm för att lösa problemet i polynomiell tid. Följaktligen är det ogörligt att finna en exakt lösning för stora probleminstanser. Traditionellt har flera approximativa algoritmer utvecklats för att hitta lösningar nära optimum inom rimlig tid. Detta är emellertid väldigt utmanande, då det krävs expertkunskap inom området för att utveckla nya, förbättrade algoritmer. Den häftiga utvecklingen av djupinlärningsmetoder under de senaste åren gör det väldigt frestande att inkorporera dessa i kombinatoriska optimeringsproblem för automatisk inlärning av förbättrade algoritmer. En annan attraktiv aspekt av djupinlärningsmetoder är deras snabba beräkningstider, vilket potentiellt kan medföra tillämpningar till större probleminstanser än modern lösningsmjukvara kan hantera.

I denna avhandling presenterar vi en vägledd djupinlärningsmetod för att erhålla approximativa lösningar till ruttplaneringsproblemet på tvådimensionella grafer. Vi implementerar ett artificiellt grafneuronnät vilket lär sig en probabilistisk representation av sökrymden, alltså mängden av kandidatlösningar. Representationen konverteras till en giltig lösning av problemet med hjälp av en sökalgoritm kallad strålsökning (beam search). Implementeringen av strålsökning är parallelliserad, vilket leder till snabb utforskning av sökrymden. Grafneuronnätet utvärderas på tre olika storlekars probleminstanser. Nätet tränas på näroptimala lösningar erhållna av Googles lösningsmjukvara OR-tools.

Vår metod producerar nöjaktiga resultat på små probleminstanser med 20 noder i grafen, med lösningar i genomsnitt under 5 % från mållösningen. Tillämpning till större probleminstanser visar sig utmanande för vår metod, då lösningar till probleminstanser med 50 noder i genomsnitt är över 100 % från mållösningen. En annan svaghet hos vår metod är dess oförmåga att generalisera till problemstorlekar den inte påträffat under träningstid. Dessa resultat begränsar tillämpningen av vår metod till trivialt små probleminstanser, vilka kan effektivt lösas av standard lösningsmjukvara.

Nyckelord ruttplaneringsproblemet, djupinlärning, återkopplade relationella neuronnät, strålsökning, genomsnittlig optimalitetsavvikelse

Contents

1 Introduction

Combinatorial optimization is a subfield of discrete optimization that sees application in areas such as transportation, supply chain management and scheduling. Many important combinatorial optimization tasks belong to the class of NP-hard problems, meaning that there are no known algorithms for solving the problem optimally in polynomial time. Because of this, exact methods that guarantee optimal solutions become intractable at large scales. To achieve manageable computation times for these problems, many heuristics have been developed (Gendreau et al., 2010) that approximate the solution. Developing better heuristics, however, requires expert knowledge, and is challenging and time consuming. In recent years there has been a growing interest in incorporating machine learning in the field of combinatorial optimization (Bengio et al., 2021). This allows for automatic learning of better heuristics to improve search algorithms and can also decrease computation times. In this thesis we explore a deep learning model for finding a probability distribution over the solution space to guide a search algorithm.

We focus on the Vehicle Routing Problem (VRP). It is one of the most studied combinatorial optimization problems and has many important applications in transportation and distribution. According to Moghdani et al. (2021), the application of computerized procedures for distribution process planning can produce savings of up to 20 % in global transportation costs. There are multiple variations of the VRP, but we follow a simple variation where the goal is to complete all deliveries as soon as possible, which has the following problem statement: "Given n customer nodes and a fleet of v vehicles, find the optimal set of sub-routes such that every customer gets visited and the length of the longest sub-route is minimized". No further constraints, e.g., vehicle capacity, are added.

Graph Neural Networks (GNN) are a class of deep learning models that operate on graph input data. The VRP, and many other combinatorial optimization problems, are graph-based problems. Hence, it is natural to look to GNNs for a suitable model, and GNNs have seen many applications in deep learning for combinatorial optimization [Nowak et al. (2017); Dai et al. (2017); Selsam et al. (2018); Kool et al. (2018); Li et al. (2018); Joshi et al. (2019); Gao et al. (2020)].

Recently proposed deep learning models utilize autoregressive decoding to build the solution to the VRP step-by-step [Nazari et al. (2018); Kool et al. (2018); Hottung and Tierney (2019); Gao et al. (2020); Sheng et al. (2020)]. They use reinforcement learning (RL) to train a policy for selecting the next node in the solution based on a reward function computed at each step. We explore a non-autoregressive model trained in a supervised manner, similarly to the framework of Joshi et al. (2019) for solving the Travelling Salesperson Problem (TSP), a special case of the VRP where only one vehicle is available. Supervised learning (SL) has a few advantages over RL by being less computationally expensive, as well as generally outperforming in solution quality given sufficient training data.

Our proposed deep learning framework combines a GNN with beam search to produce approximate solutions to the VRP. The network, VRPNet, is based on the Recurrent Relational Network architecture introduced in Palm et al. (2017). The

network attempts to infer which sub-route each node in the input graph belongs to by iteratively sending messages over the graph, encoding the current hidden states and positions of each node, and updating their hidden states based on the received messages. The hidden states of each node pair is used to determine the probability of the edge connecting them being active, i.e. lying on the solved route. The output of the network is an adjacency matrix denoting the probability of each edge being active. This adjacency matrix describes a probabilistic heat-map over the graph, on which beam search is performed to construct a valid route. Beam search is a greedy algorithm that explores a limited set of candidate routes by iteratively expanding them with the most likely node. We use a standard supervised learning procedure to train the network by minimizing the loss between pairs of problem instances and (near-)optimal solutions generated with Google's OR-tools solver.

The model manages to produce valid solutions and finds solutions close to optimal (under 5 % from optimal) for small problem sizes. As the problem size grows however, the model struggles, failing to produce good solutions already for a VRP with 50 nodes. The model is also unable to generalize to unseen problem sizes, limiting the application of the model to trivially small problem sizes that can easily be solved by other readily available algorithms.

2 Related work

Neural Networks (NN) have seen application to combinatorial optimization problems since the 1980s, when Hopfield and Tank (1985) developed the Hopfield network to solve the travelling salesperson problem (TSP). Most of the early research in this field focused on the TSP, and the two main approaches used were Hopfield networks and self-organizing feature maps (Smith, 1999). In recent years, with the dramatic progress achieved with deep learning, there has been a resurgence in research applying NNs to the field of combinatorial optimization.

The Pointer Network (Ptr-Net) was introduced by Vinyals et al. (2015). The Ptr-Net is a sequence-to-sequence based model, which uses a Recurrent Neural Network (RNN) encoder and decoder with an attention mechanism to output permutations of the input sequence. The model is trained in a supervised manner and applied to the convex hull, Delaunay triangulation and the TSP. They use a beam search procedure to obtain valid solutions to the TSP at test time. Bello et al. (2016) improve on the Ptr-Net by using reinforcement learning to train the network in an unsupervised manner. Their Actor-Critic framework is trained via a policy gradient method, where the REINFORCE algorithm (Williams, 1992) is used to estimate the policy gradient based on solutions obtained using Monte-Carlo sampling. They mask out previously visited nodes at each decoding step to ensure valid solutions. Similarly, Nazari et al. (2018) also train a Ptr-Net in an actor-critic framework. They argue that the order of the input is not meaningful, hence they omit the RNN encoder, instead opting to use high-dimensional embeddings of the input. They apply the network to the capacitated VRP and VRP with split deliveries, outperforming classical heuristics like the Clarke-Wright savings (Clarke and Wright, 1964) and Sweep (Wren and

Holliday, 1972) heuristics, as well as OR-tools on medium-sized problems (up to $n = 100$). Sheng et al. (2020) propose a Ptr-Net for solving the VRP with Task Priority and Limited Resources. They use a strategy gradient method, estimating the gradient based on the calculated benefits of two consecutive batches of training data. They demonstrate better solutions at faster computation times than the Genetic Algorithm (Reeves, 2010) at problem sizes up to $n = 150$.

Dai et al. (2017) move away from Ptr-Nets in favor of GNNs. Their proposed model, structure2vec, is a graph embedding network better suited for capturing the combinatorial structure of problems like the TSP than sequence-to-sequence based models. Using a 1-step DQN (Mnih et al., 2015) training procedure, they train a greedy policy that incrementally constructs the solution tour based on the current embedding. They add a helper function to find the best insertion point of each node. Deudon et al. (2018) and Kool et al. (2018) replace the structure2vec network with Graph Attention Networks (GAT) (Veličković et al., 2017). Deudon et al. (2018) train an attention-based encoder-decoder to autoregressively build a solution to the TSP using REINFORCE. The output of the network itself is comparable to high-performing heuristics, e.g., OR-tools, but a hybrid approach combining the network solution with the 2-opt heuristic (Croes, 1958) provides improved results. Kool et al. (2018) propose a different, more powerful decoder and improve training by using REINFORCE with a greedy rollout baseline, creating a non-hybrid GAT model that outperforms the hybrid approach of Deudon et al. (2018). The model is also applied to capacitated VRP and VRP with split deliveries, outperforming standard heuristics like OR-tools, and the Ptr-Net of Nazari et al. (2018).

Hottung and Tierney (2019) and Gao et al. (2020) tackle the VRP by combining NNs with the Large Neighborhood Search (LNS) heuristic (Shaw, 1998). The approach of Hottung and Tierney (2019) uses two simple pre-defined destroy procedures to remove nodes from a partial solution and learns the repair operator using an attention-based embedding network without encoding and decoding, while Gao et al. (2020) use a GAT with integrated node and edge embeddings to learn both the destroy and repair operators. Both models are trained in an actor-critic framework, with Hottung and Tierney (2019) using REINFORCE, and Gao et al. (2020) using the more novel Proximal Policy Optimization method (Schulman et al., 2017). The framework of Gao et al. (2020) is shown to outperform both powerful heuristics, such as Adaptive Large Neighborhood Search (Ropke and Pisinger, 2006) and Slack Reduction by String Removals (Christiaens and Vanden Berghe, 2020), and the prominent GAT model of Kool et al. (2018) for medium-sized problems. It can even handle problems as large as $n = 400$. Hottung and Tierney (2019) also outperform the model of Kool et al. (2018), and demonstrate results comparable to current state-of-the-art heuristic solvers such as LKH3 (Helsgaun, 2017). No direct comparison between these two approaches is made.

While the field of neural combinatorial optimization is dominated by autoregressive approaches trained using RL, a few non-autoregressive models have been proposed. Nowak et al. (2017) propose an SL framework, where a GNN based on Scarselli et al. (2008) learns to output the solution of the TSP as an adjacency matrix, ensuring feasibility of the solution via beam search. They only report results for $n = 20$

with an unimpressive performance compared to many other available approaches. Joshi et al. (2019) follow suit with a similar framework, replacing the GNN with a Graph Convolutional Network (GCN) (Bresson and Laurent, 2017). The GCN proves effective, achieving better scores than other NNs, including the GAT of Kool et al. (2018), on the TSP. However, Joshi et al. (2019) acknowledge the limitations of using an SL framework. Since their model fails to generalize to unseen problem sizes, its application gets restricted to small problem sizes for which sufficient training data is attainable. Despite this, their results show potential for SL frameworks in neural combinatorial optimization, assuming a model with good generalization can be developed.

3 Data

The data used for training and testing are randomly generated instances of the VRP solved using OR-tools. OR-tools is a Google Suite that provides powerful solvers for important optimization tasks. The solvers for routing problems are approximative, so the solutions are not guaranteed to be global optima. To generate an instance of the VRP, we choose n nodes randomly on the coordinate grid $S = \{x_i\}_{i=1}^{n}$, where every $x_i \in [0, 1000]^2$. We compute the distances between each pair of nodes and store them in the $n \times n$ distance matrix $D = \{d_{ij}\}_{i,j=1}^{n}$, where $d_{ij} = ||x_i - x_j||_2$ is the Euclidean distance. We also need to define the number of available vehicles v and the depot node. The number of available vehicles is the same for every instance in the data set and the depot is always chosen as the 0^{th} node. We choose the local search metaheuristic used by the solver to be Guided Local Search (GLS), an efficient local search algorithm that penalizes the objective function to help escape local minima (Voudouris et al., 2010). The solution returned by the solver is stored as an edge adjacency matrix $\mathbf{y} = \{y_{ij}\}_{i,j=1}^{n}$, which will be the target of VRPNet.

To examine the effect of varying problem sizes on model performance, we generate three different data sets. The model is trained separately on each data set. The first set, VRP20, contains 10 000 instances of a 20 node VRP with five available vehicles. The second, TSP20, contains 10 000 instances of a 20 node VRP with one available vehicle, making it equivalent to a TSP. The final set, VRP50, includes 1000 instances of a 50 node VRP with five available vehicles. In the OR-tools solver, we need to set the stopping criterion for the local search. We use a time limit as the stopping criterion. For the smaller problem sizes, OR-tools tends to find the best solution in under 0.5 seconds, so we set the time limit to 1 second. For the VRP50, we need to increase the time limit to consistently get the best solution, so we set it to 10 seconds. Because of the increased solution time, we only generate and solve 1000 instances.

VRPNet receives as input the coordinates of each node in the graph and the distances between each pair of nodes. We also add a special token q_i to each node to signal to the network if the node is a depot or not. The input is normalized, since we found it to slightly improve learning. The coordinates are normalized to the unit grid $S_{unit} = \{\hat{x}_i\}_{i=1}^{n}$, where every $\hat{x}_i \in [0, 1]^2$. The pair-wise distances d_{ij} are also

normalized using the standard score:

$$\hat{d}_{ij} = \frac{d_{ij} - \mu}{\sigma}, \tag{1}$$

where μ is the mean distance and σ the standard deviation of the distances. The data is divided into training and test data using a random 80/20 split.

Table 1: Table summarizing the sizes of and number of vehicles used for the different datasets.

Dataset	Number of instances	Number of vehicles
VRP20	10 000	5
TSP20	10 000	1
VRP50	1000	5

4 Methodology

Our proposed neural network, VRPNet, is based on the Recurrent Relational Network (RRN) architecture. It can be viewed as a learned message passing algorithm. The network computes messages between nodes and updates the hidden states of the nodes over T iterations. In one iteration t, each node i in the graph sends messages to its neighboring nodes, informing them of its current state. All messages are computed in parallel. Each node then considers all incoming messages, and updates its hidden state from h_i^{t-1} to h_i^t based on the received messages. The network computes new messages based on the updated hidden states and repeats the process. The output of the network is computed based on the hidden states of the nodes. (Palm et al., 2017)

The recurrent relational network can be used to compute the individual output of each node or one output for the whole graph. We are interested in outputting the probability of each edge e_{ij} being active or inactive, i.e., $p(e_{ij})$. Hence, we modify the the architecture to include hidden states e_{ij}^t for the edges as well, which are updated based on the hidden states of the pair of nodes that the edge connects. The final output is then computed based on the hidden states of the edges.

We hypothesize, that the relational reasoning of the RRN will provide good representations of the solutions to the VRP by implicitly learning which sub-route each node belongs to. The hidden state of a node will represent the likelihood of it belonging to a sub-route. As the confidence of a node belonging to a specific sub-route increases, it will communicate this to its neighboring nodes, and the neighboring nodes will update their likelihoods accordingly. Knowing which routes are more likely for each node will help the network make a more informed decision when predicting the edge probabilities in the output than simply using the relative positions of the nodes. The model implementation, as well as the data and the results of this thesis, are available at https://github.com/hagstromf/VRPNet.

4.1 VRPNet

VRPNet receives as input the graph describing an instance of the VRP. The input graph is fully-connected, that is, each node is connected to every other node in the graph. We omit loops from a node to itself, since these would be redundant in the solved route. Each node i has an input feature vector $z_i = \begin{bmatrix} \hat{x}_i & q_i \end{bmatrix}$, consisting of the normalized coordinates and token of the node. Every edge e_{ij} has its normalized length $\hat{d}_{ij}$ as input. The hidden states of the nodes and edges are initialized to zero, $h_i^0 = \mathbf{0}$ and $e_{ij}^0 = \mathbf{0}$.

4.1.1 Message passing phase

At each iteration t, each node has has hidden state h_i^t. Each node sends a message to its neighboring nodes, which is computed using the message function f. At iteration t, the message m_{ij}^t from node i to node j is computed as:

$$m_{ij}^t = f(h_i^{t-1}, h_j^{t-1}, \hat{d}_{ij}) \tag{2}$$

We add $\hat{d}_{ij}$ as input to allow the message function f to explicitly take the distance between the nodes into account. The message function is a Multilayer Perceptron (MLP). Using an MLP allows the network to learn the best messages to send. We use an MLP with three layers: an input layer, a hidden layer and a linear output layer. We use the rectified linear unit (ReLU) as the activation function in the first two layers. We also apply batch normalization (Ioffe and Szegedy, 2015) and dropout (Hinton et al., 2012) with probability 0.2 to the first two layers to regularize the model. We found that combining both methods provided better results than using only one of them. A node j needs to consider every incoming message. To do this, we aggregate the messages with:

$$m_j^t = \sum_{i \in N(j)} m_{ij}^t, \tag{3}$$

where $N(j)$ is the neighborhood of node j. Since we input a fully-connected graph, the neighborhood is every other node in the graph.

4.1.2 Node updating phase

After aggregating all the incoming messages to node j, its hidden state h_j^t is updated using the update function g_n, which is a learned RNN. We use the Gated Recurrent Unit (GRU) module. The update function receives as input the hidden state of the node after the previous iteration h_j^{t-1}, the aggregated message m_j^t and the node input z_j:

$$h_j^t = g_n(h_i^{t-1}, m_j^t, z_j). \tag{4}$$

The input feature vector z_j is provided at each iteration so that the update function g_n will not have to remember the original input. It can instead focus on the incoming

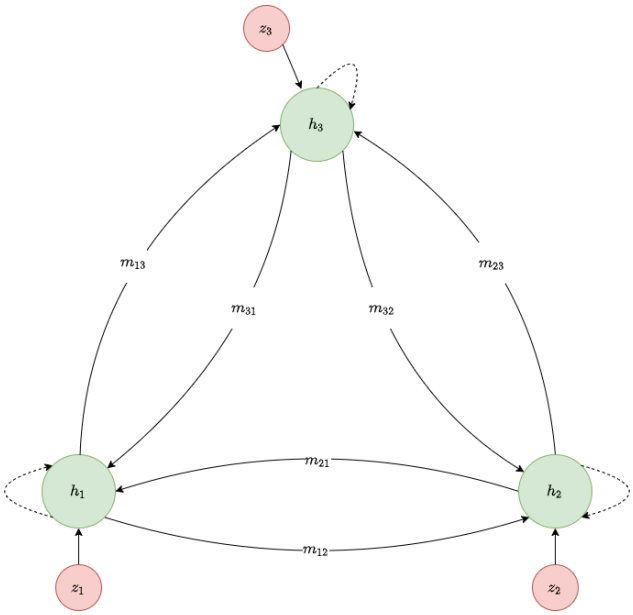


Figure 1: An illustration of the message and node updating phases of one iteration on a three node graph. The green nodes represent hidden states h_i. Messages m_{ij} are sent between each pair of nodes. Additionally, the nodes receive their input feature vector z_i, represented by the red nodes. The dashed lines represent the recurrent connection for updating the node state.

messages. Figure 1 provides an illustration of the message passing and node updating phases (2)-(4) on a small graph.

4.1.3 Edge updating phase

Knowing the states of each node is not enough, since we are interested in computing the probability $p(e_{ij})$ of each edge e_{ij} in the graph being active or inactive. To do this, we extend the RRN architecture of Palm et al. (2017) to include hidden states for each edge as well. The edge probabilities are then computed based on the hidden states of each edge.

Similarly to the node updates, the hidden states of each edge e_{ij}^t are updated using a learned update function g_e:

$$e_{ij}^t = g_e(e_{ij}^{t-1}, h_i^t, h_j^t, \hat{d}_{ij}), \tag{5}$$

where h_i^t and h_j^t are the curent hidden states of the nodes connected by edge e_{ij}. Like in (4), we insert the input feature vector $\hat{d}_{ij}$ of the edge at each iteration so that g_e does not need to memorize it. The edge update function g_e is learned through another GRU module. Figure 2 provides an illustration of the edge updating phase (5) on a small graph.

Finally we compute the edge probabilities using the output function o:

$$p_{ij}^t = o(e_{ij}^t) \tag{6}$$

The output is a 2-feature vector $p_{ij}^t \in \mathbb{R}^2$ denoting the probabilities of the edge being inactive and active respectively. The output function o is similar to the learned message function f. We use a three-layer MLP, with ReLU as the activation function in the first two layers and a linear output layer. Here we also apply batch normalization and dropout with probability 0.2 to the first two layers. The raw output values p_{ij}^t are mapped to a proper probability density $\hat{p}_{ij}^t \in [0, 1]^2$ using softmax nonlinearity.

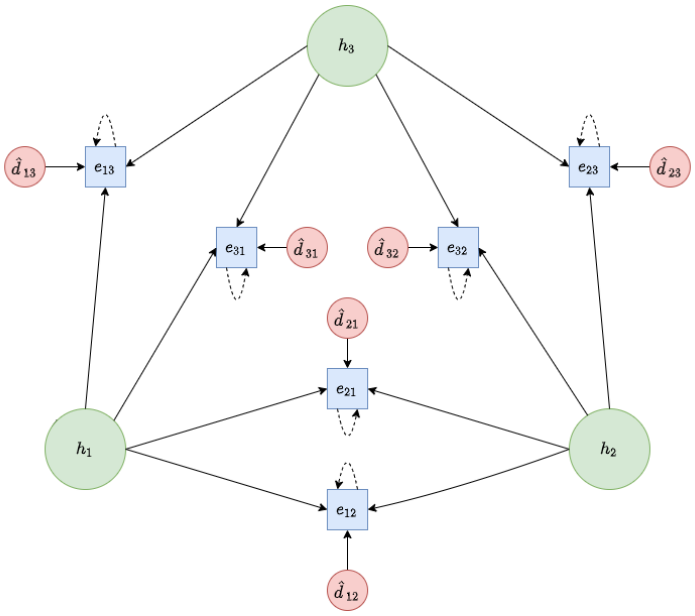


Figure 2: An illustration of the edge updating phase on a three node graph. The blue boxes represent the hidden states e_{ij} of the edge connections. They receive information from the hidden states h_i (green) of the nodes they connect as well as the edge input feature vector $\hat{d}_{ij}$ (red). The dashed lines represent the recurrent connection. The update is performed separately for edges in both directions, to allow for different input in case one way was for instance blocked off.

4.2 Loss

The target of the output is the solved VRP's adjacency matrix $\mathbf{y} = \{y_{ij}\}_{i,j=1}^n$, where n is the number of nodes in the respective VRP. Each edge y_{ij} has either the value 0 (inactive edge) or 1 (active edge). This is essentially a binary classification problem, where the network learns to classify each edge as either inactive or active, and so we train the network by minimizing the cross-entropy loss over mini-batches. Since the adjacency matrix is sparse, i.e., there are many more inactive edges than active ones, the classification problem becomes highly skewed towards the inactive class as the problem size increases. Hence, we need to balance the classes in the loss computation using appropriate class weights, as suggested by Joshi et al. (2019). Given a dataset of s adjacency matrices of size $n \times n$, and $C = 2$ classes, the weight w_c for each class y_c is computed as:

$$w_c = \frac{sn^2}{C \cdot \#y_c}, \tag{7}$$

where $\#y_c$ is the number of examples belonging to class y_c.

During training, we minimize the loss at every iteration t of the network, as proposed by Palm et al. (2017). They claim that this encourages the network to learn a convergent message passing algorithm, as well as helps with the vanishing gradient problem. During validation we only consider the output of the final iteration of the network.

4.3 Beam search

VRPNet outputs a probabilistic heat-map over the edges in the adjacency matrix of the VRP solution. This heat-map cannot be directly converted to the final adjacency matrix representation of the solution using for instance an argmax function, since this will generally yield invalid routes, with either missing or superfluous edges. We must use some search algorithm to find valid routes. We implement a beam search decoder for converting the heat-map into valid solutions. (Joshi et al., 2019)

Beam search is a limited-width breadth-first search, which finds high-probability routes through sampling a subset of the possible routes on the graph. Beam search starts at the depot node. Given the beam width b, the algorithm expands the b most probable edge connections in the neighborhood of the depot. After that, beam search iteratively expands the top-b most probable partial routes π' until each route has visited every node. The probability of a partial route π' can be computed using the chain rule of probability as:

$$p(\pi') = \prod_{j' \sim i' \in \pi'} p_{i'j'}, \tag{8}$$

where node j' follows node i' in π'. (Joshi et al., 2019)

To prevent re-visits to nodes and construct valid routes, (Joshi et al., 2019) mask out nodes when they are visited. This masking strategy only works for TSPs. To generalize the beam search to VRPs, we need to allow the depot node to be visited multiple times. Hence, we modify the masking strategy such that the depot node is not immediately masked out once it is visited. However, to obtain valid routes, we also need to ensure that the number of sub-routes in the solution does not exceed the number of available vehicles v. We keep a counter on the number of visits to the depot, and mask it out if it has been visited v times.

We implement two different strategies for choosing the final output of the beam search decoder:

- **Vanilla beam search**: The output of vanilla beam search is the complete route with the highest probability at the end of beam search. The route with the highest probability is not necessarily the solution that minimizes the longest sub-route. This strategy is used for fast validation during training, in order to track the improvement of the heat-maps outputted by VRPNet. If there

is improvement in the quality of the heat-maps then the most probable route should approach the target route.

– **Shortest beam search**: The output of shortest beam search is the solution that minimizes the longest sub-route. This takes considerably longer than vanilla beam search, since it needs to find and evaluate the length of the longest sub-route of all b complete routes after beam search. This strategy is used for final evaluation of the trained models.

5 Experiments

5.1 Training

We train VRPNet by minimizing the cross-entropy loss between the outputted probabilistic heat-map and the target adjacency matrix. The loss is minimized via stochastic gradient descent using Adam optimizer (Kingma and Ba, 2014) with a learning rate of 0.001. Higher learning rates had difficulty converging, often exhibiting huge increases in loss in the middle of training. For one epoch, the loss is minimized over all available training examples, with 8000 examples for VRP20 and TSP20, and 800 for VRP50. The training examples are divided into mini-batches that will fit the GPU memory. The batch size is 50 for VRP20 and TSP20, and 20 for VRP50. At the end of each epoch, the model is validated on the entire test set. Valid routes are constructed using vanilla beam search with beam width $b = 100$ and evaluated using average optimality gap (see section 5.2). The test set consists of 2000 examples for both VRP20 and TSP20, and 200 examples for VRP50. The test set is also divided into mini-batches with the same batch size as the training set. The GPU used is an Nvidia RTX 2060 Super.

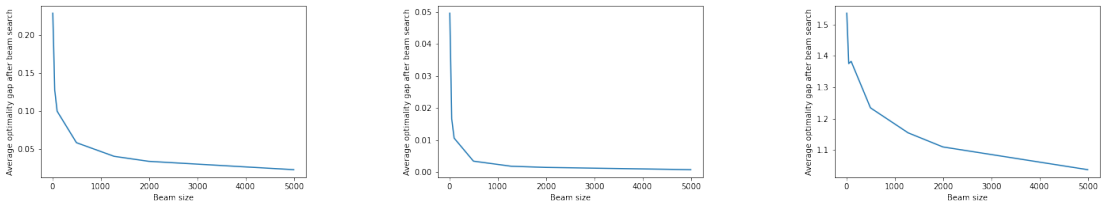
5.2 Evaluation

To evaluate the model we compare its performance to that of OR-tools. The metric used to compare performance is called the average optimality gap (Joshi et al., 2019). Given the length of the longest sub-route in the predicted solution l and the length of the longest sub-route in the target solution $\hat{l}$, the average optimality gap aog over m examples is given by:

$$aog = \frac{1}{m}\sum_{i=1}^{m}(\frac{l_i}{\hat{l}_i} - 1) \tag{9}$$

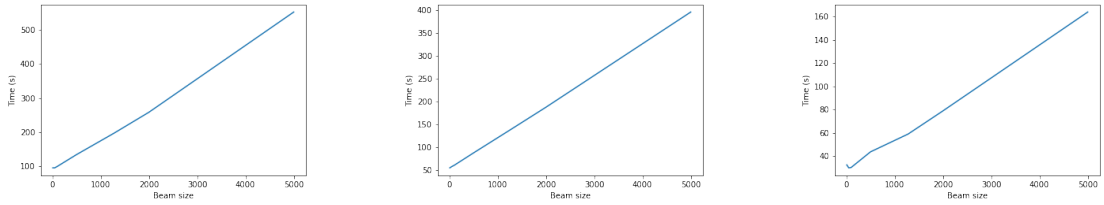
During final evaluation of a trained model, we use shortest beam search to convert the heat-maps into valid routes. One can choose arbitrarily large beam widths to improve the solution quality (see figure 3). This, however, comes at the cost of increased computation time (see figure 4). The computation time increases linearly in relation to b, since the shortest beam search strategy needs to evaluate the length of all b candidate solutions. We report the results using beam width $b = 1280$, the

same width used in Joshi et al. (2019). The trained model is evaluated over the entire test set.



(a) VRP20 (b) TSP20 (c) VRP50

Figure 3: Evaluation of shortest beam search on the trained models using different beam width b. Increasing the beam width improves solution quality, but the improvements diminish for larger beam widths.



(a) VRP20 (b) TSP20 (c) VRP50

Figure 4: Time taken for each model to solve their entire respective test sets at different beam widths b. The computation time increases linearly in relation to b, since the shortest beam search strategy needs to evaluate every candidate solution. Evaluation was performed on an Intel Core i5-9400F 2,90 GHz CPU.

5.3 Results

In table 2, we present the average optimality gap of the best trained models found for each problem size. Since the parameters Θ of our model are independent of the size n of the input graph and the number of vehicles v, we also report the performance of each model on the other test datasets. It would be highly desirable for the models to generalize well to different problem sizes not seen during training. This would allow a model to be trained on small manageable problem sizes and then applied to very large problems with thousands of nodes. However, we observe that each model performs much better on the problem size it has been trained on than the other models, indicating poor generalization.

The TSP20 model performs the best (on its learned problem size). With $aog = 0.18\%$, it falls short of Joshi et al. (2019), who report an average optimality gap of 0.01% for the same size TSP. However, they train and evaluate their model on optimal solutions computed with the Concorde solver, so a direct comparison can't be made. Since the node j following node i in a TSP route often is usually in close

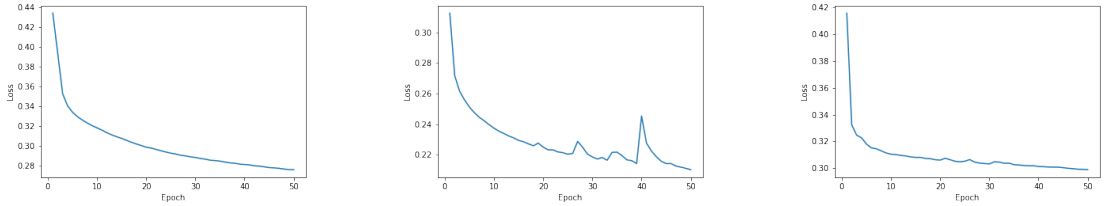
Table 2: The average optimality gap of each model evaluated on every test set. The OR-tools solver is the baseline to which the other models are compared. Solutions are obtained using the shortest beam search strategy with beam width $b = 1280$. The "uninformed" model represents the results of beam search performed on a uniform heat-map.

		Test set		
		VRP20	TSP20	VRP50
Model	OR-tools	0.00 %	0.00 %	0.00 %
	VRP20	4.06 %	10.94 %	173.50 %
	TSP20	59.74 %	0.18 %	224.12 %
	VRP50	73.02 %	5.94 %	115.48 %
	"Uninformed"	175.79 %	112.22	1049.63 %

proximity to i (Joshi et al., 2019), it is relatively easy for the network to learn a good representation of the problem. This is not the case for the VRP. Since the goal is to minimize the longest sub-route, it is most often desirable to build as many sub-routes in the solution as there are vehicles available. Hence, the optimal solution will often include connections from node j to the depot node instead of to a node in its nearest neighborhood in order to complete a sub-route. This increases the difficulty of learning a good representation, which we observe in the increase of the average optimality gap for the VRP20 model. The VRP20 model still finds solutions fairly close to OR-tools. However, for VRP50 we observe a drastic increase in the average optimality gap. This shows that while the model can learn decent representations of the data for graphs of size $n = 20$, it struggles already at size $n = 50$. Providing a larger data set for training could improve the performance slightly, but it seems unlikely to yield results anywhere close to that of the smaller problem sizes. While the VRP20 and TSP20 model perform inference over 30 iterations, VRP50 had to be restricted to 15 to fit to GPU memory, which might be a contributing factor to the poor performance. Since we find no research that evaluate their model on the same variation of the VRP as us, we cannot directly compare our performance to other frameworks in the literature.

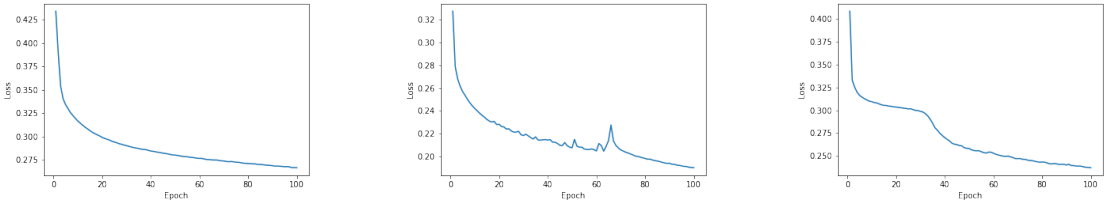
We include an "uninformed" model in table 2, where beam search is performed on a uniform heat-map. The performance of this "uninformed" beam search is considerably poorer on every test set, indicating that VRPNet is able to learn representations that are useful in improving the solutions constructed using beam search.

In figures 5 and 6 we see that, during training, the loss rapidly decreases in the first few epochs, after which the decrease steadily slows down. The loss doesn't always decrease smoothly, as exhibited in figures 5b, 6b and 6c. While loss tends to decrease over the entire training loop, average optimality gap does not. As can be seen from figures 7 and 8, the progression of average optimality gap is erratic, and does not exhibit a correlation with the decrease in loss. This indicates poor generalization of the model to unseen test data. We could train the models for longer, further decreasing the training loss as shown in figure 6, but this does not necessarily translate to an improved average optimality gap, as can be seen in in figure 8a. The



(a) VRP20 (b) TSP20 (c) VRP50

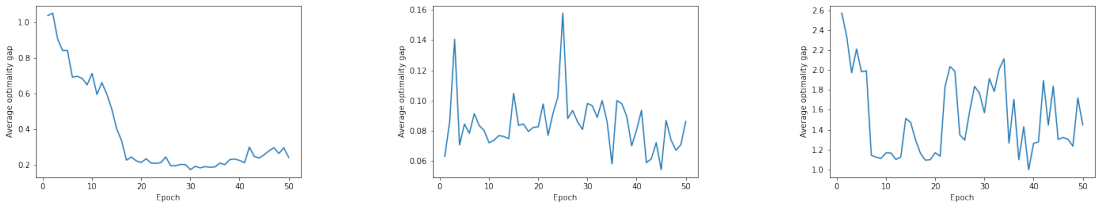
Figure 5: Progression of loss for every model trained over 50 epochs.



(a) VRP20 (b) TSP20 (c) VRP50

Figure 6: Progression of loss for every model trained over 100 epochs.

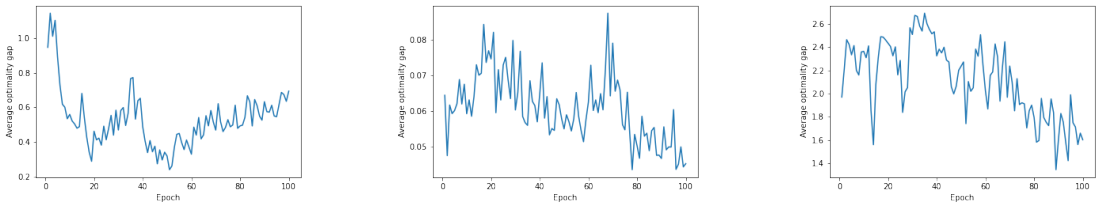
unpredictability and significant variation of the average optimality gap at the end of training suggests that the model is sensitive to the initialization of the model parameters Θ. This unpredictability also makes it impossible to determine the best cut-off point for the training loop.



(a) VRP20 (b) TSP20 (c) VRP50

Figure 7: Progression of average optimality gap for every model trained over 50 epochs.

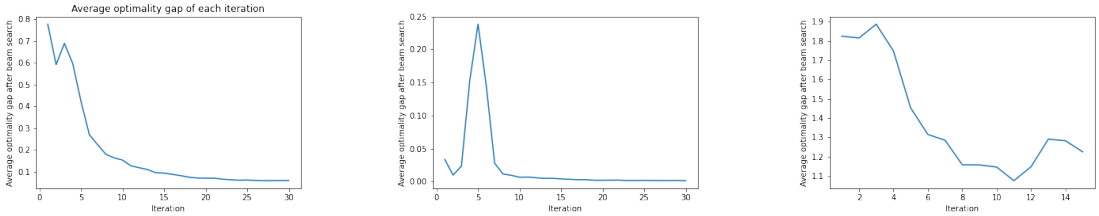
In figure 9, we display the performance of beam search using the heat-maps outputted by each iteration of the network. The average optimality gap tends to decrease towards the final iterations of the network, indicating that iteratively updating the node and edge states using the information captured by the messages can improve the representation learned by the network. However, the final iteration does not necessarily produce the best heat-map for beam search, as can be seen in 9c. It is possible that, as the problem size increases, the individual computed messages are not strong enough to produce significant variation in the aggregated messages, limiting the improvement that can be achieved through iteration. This



(a) VRP20 (b) TSP20 (c) VRP50

Figure 8: Progression of average optimality gap for every model trained over 100 epochs.

could explain the drastic increase in average optimality gap of the VRP50 model we observed in table 2. Using a deeper MLP module for the message function might lead to stronger messages, improving the performance for larger problem sizes. Alternatively, replacing the MLP module with another module, e.g., a Convolutional Neural Network module, could be necessary to capture stronger messages.



(a) VRP20 (b) TSP20 (c) VRP50

Figure 9: Performance of each model over every iteration, evaluated by performing shortest beam search on the heat-map outputted at the respective iteration.

In appendix A we provide a brief sensitivity analysis of the different parameters of the network. For examples of the solutions found with our model, refer to appendix B.

6 Conclusions

In this thesis, we demonstrate that it is possible to apply a supervised deep learning framework to find approximate solutions to the vehicle routing problem. Our developed neural network VRPNet outputs probabilistic heat-maps that can guide the search of the solution space using a suitable search algorithm. Ultimately, the model fails to generalize to problem instances not seen during training. The model also struggles to find good solutions already at problem size $n = 50$. This is a glaring weakness in developing deep learning models for the vehicle routing problem, and combinatorial optimization problems in general, in the supervised learning framework. If a trained model can only be applied to fixed problem sizes, the application of the model framework gets limited to trivially small problem sizes, since it is infeasible

to generate data sets with optimal solutions to problems with hundreds or even thousands of nodes.

To apply supervised learning to solving the vehicle routing problem, it is imperative to develop a model capable of generalizing to larger unseen problem instances. Further research could be dedicated to improving the generalization of VRPNet by exploring deeper MLP modules and alternative modules like Convolutional Neural Networks, as well as other Recurrent Neural Network modules for the node and edge updating phases, e.g., long short-term memory. We could also explore other search algorithms for traversing the solution space. Alternatively, we could explore our model in an unsupervised learning framework by incorporating reinforcement learning. This would eliminate the need for finding/generating training data sets with optimal or near-optimal solutions, allowing us to train the model for larger problems than currently feasible. Another appealing prospect of RL is the possibility to develop new heuristics capable of competing with current state-of-the-art solvers, instead of attempting to imitate an established solver using SL. Future work will also extend the model to more common variations of the VRP, e.g. capacitated VRP and VRP with split deliveries.

References

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

Jan Christiaens and Greet Vanden Berghe. Slack induction by string removals for vehicle routing problems. *Transportation Science*, 54(2):417–433, 2020.

Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In *In-*

ternational conference on the integration of constraint programming, artificial intelligence, and operations research, pages 170–181. Springer, 2018.

Lei Gao, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu. Learn to design the heuristics for vehicle routing problem. *arXiv preprint arXiv:2002.08539*, 2020.

Michel Gendreau, Jean-Yves Potvin, et al. *Handbook of metaheuristics*, volume 2. Springer, 2010.

Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 2017.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

John J Hopfield and David W Tank. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

André Hottung and Kevin Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv preprint arXiv:1911.09539*, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *arXiv preprint arXiv:1810.10659*, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Reza Moghdani, Khodakaram Salimifard, Emrah Demir, and Abdelkader Benyettou. The green vehicle routing problem: A systematic literature review. *Journal of Cleaner Production*, 279:123691, 2021.

Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V Snyder, and Martin Takáč. Reinforcement learning for solving the vehicle routing problem. *arXiv preprint arXiv:1802.04240*, 2018.

Alex Nowak, Soledad Villar, Afonso S Bandeira, and Joan Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *stat*, 1050:22, 2017.

Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. *arXiv preprint arXiv:1711.08028*, 2017.

Colin R Reeves. Genetic algorithms. In *Handbook of metaheuristics*, pages 109–139. Springer, 2010.

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431. Springer, 1998.

Yuxiang Sheng, Huawei Ma, and Wei Xia. A pointer neural network for the vehicle routing problem with task priority and limited resources. *Information Technology and Control*, 49(2):237–248, 2020.

Kate A Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Christos Voudouris, Edward PK Tsang, and Abdullah Alsheddy. Guided local search. In *Handbook of metaheuristics*, pages 321–361. Springer, 2010.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

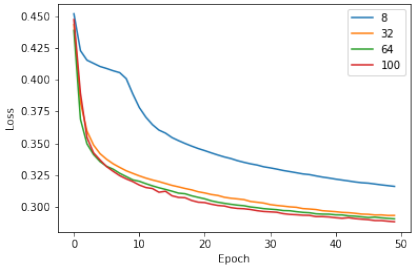
Anthony Wren and Alan Holliday. Computer scheduling of vehicles from one or more depots to a number of delivery points. *Journal of the Operational Research Society*, 23(3):333–344, 1972.

A Sensitivity Analysis

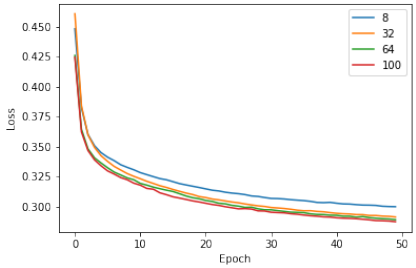
We perform a sensitivity analysis of the different parameters of the network on the VRP20 model. During the sensitivity analysis we limit the network to 15 iterations to fit the largest configurations to GPU memory. We analyse the effect of changing the number of features in the hidden states of the nodes and the edges, as well as a few different configurations of the number of features (hidden dimensions) in the layers of the message and output functions.

In figure A1, we display the progression of loss for each case. We observe in figure A1a that 8 features in the node states is not enough, but having over 32 features leads to very minor changes in the loss value. In figure A1b, we notice similar behaviour for the number of features in the edge states, though it seems that these are even less sensitive to a small number of features. From figure A1c, it seems that constantly increasing the dimension of each layer in the message function has a negative effect on the progression of loss, but other configurations, like multiple layers with the same dimension, constantly decreasing dimension, and increasing in the beginning and decreasing in the end, provide similar results. In figure A1d, we observe slightly worse performance with configurations where we decrease the number of features over each layer. Other configurations provide similar results. Notice that we can only vary the number of features in the first two layers, since the output of each edge always has two features. It seems that the message function and node states are more sensitive to changes in their configuration, indicating that exploring other representations of the messages could have a considerable effect on the quality of the produced heat-maps.

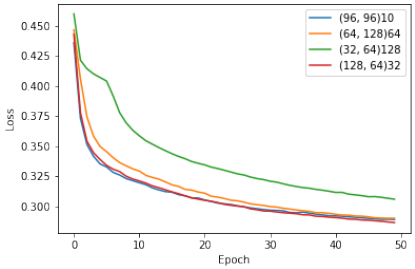
The average optimality gap of each case is presented in figure A2. The same erratic behaviour discussed in section 5.3 is observed for all configurations. As can be seen for instance in figure A2a, the configuration with the smallest loss (100 node features) does not necessarily yield the best performing model, making it difficult to determine which configuration is the best choice.



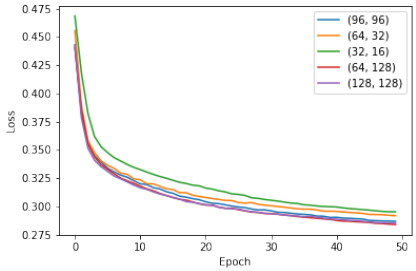
(a) Node features



(b) Edge features

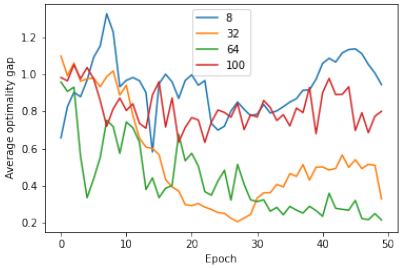


(c) Message hidden dimensions

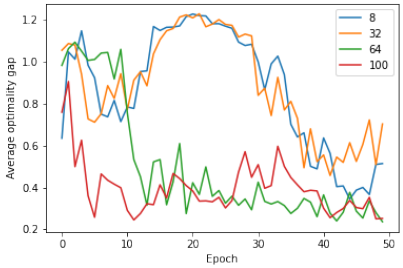


(d) Output hidden dimensions

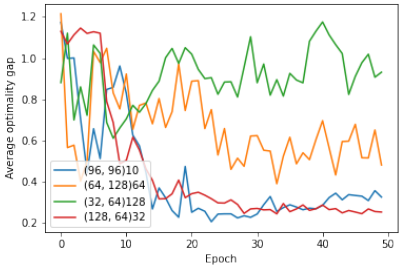
Figure A1: Progression of loss for different configurations of the VRP20 model.



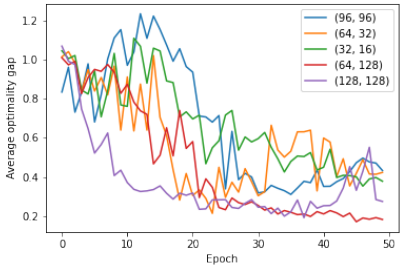
(a) Node features



(b) Edge features



(c) Message hidden dimensions

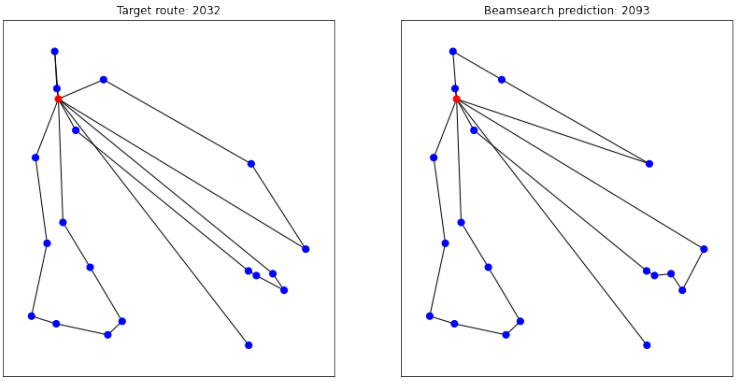


(d) Output hidden dimensions

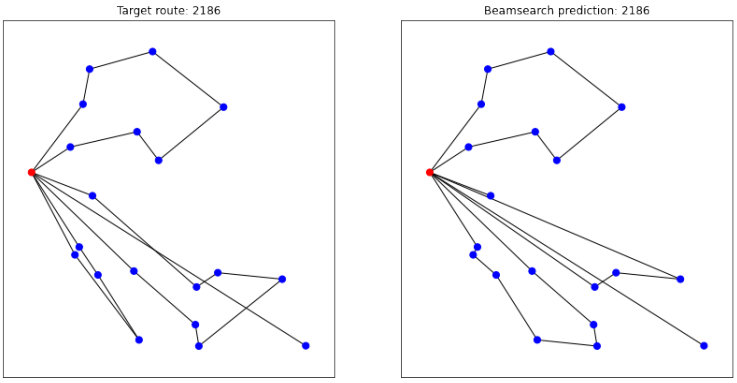
Figure A2: Progression of average optimality gap for different configurations of the VRP20 model.

B Example solutions

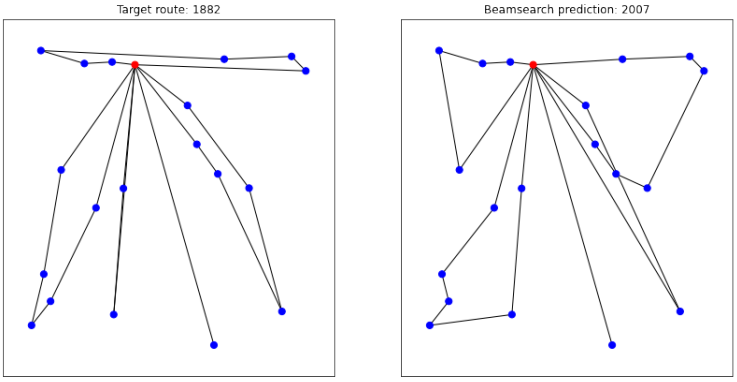
We provide a few example solutions for each trained model to demonstrate their behaviour. VRP20 (figure B1) might find a solution with one or two similar sub-routes to the solution of OR-tools, sometimes even finding the same longest sub-route (figure B1b), though in general the solution deviates from OR-tools quite a bit with a slightly longer longest sub-route. TSP20 (figure B2) generally finds the same solution as OR-tools, hence the low optimality gap of 0.18%. The best solutions to the VRP50 (figure B3) tend to have sub-routes of similar length with minimal overlap. Our model noticeably struggles, often finding a few very long sub-routes that cross each other at multiple intersections and take unnecessarily long jumps.



(a)

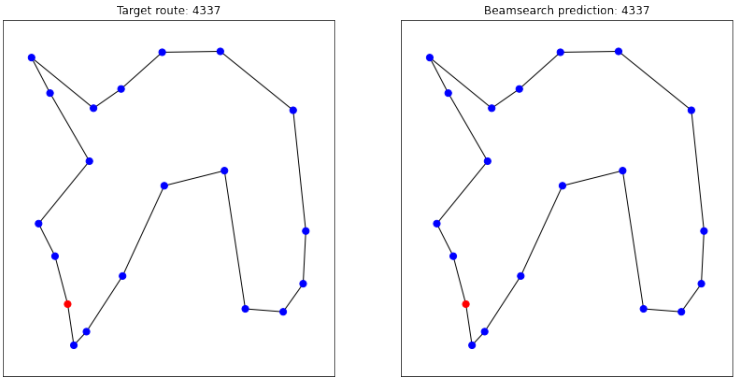


(b)

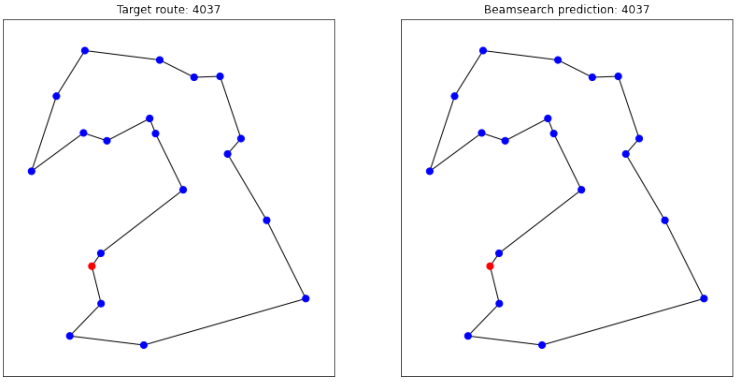


(c)

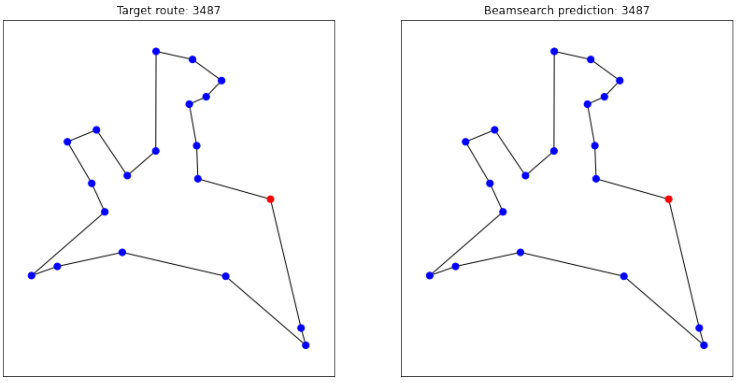
Figure B1: Example solutions of the VRP20 model. The figure on the left shows the target solution found by the OR-tools solver, and the figure on the right the result of our model after beam search, including the length of the longest sub-tour of the respective solutions. VRP20 generally produces slightly longer sub-routes than OR-tools, but occasionally finds the same longest sub-route. This does not mean that the other sub-routes in the solution will be the same as in figure B1b.



(a)

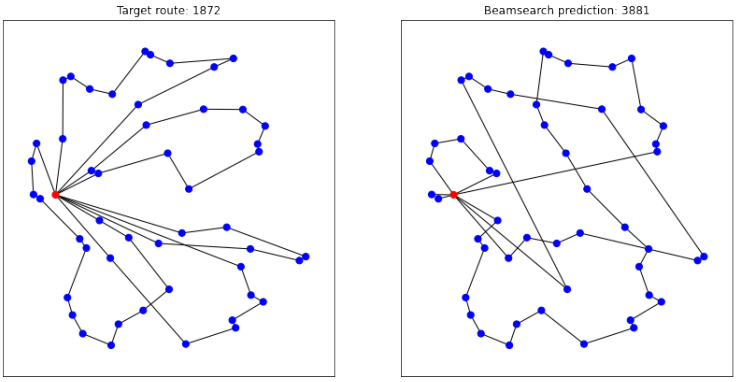


(b)

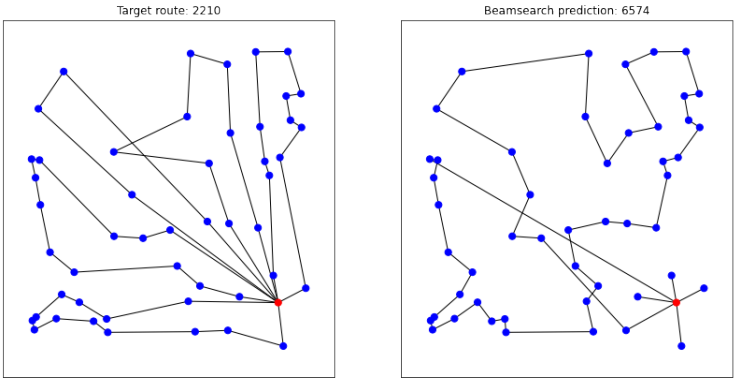


(c)

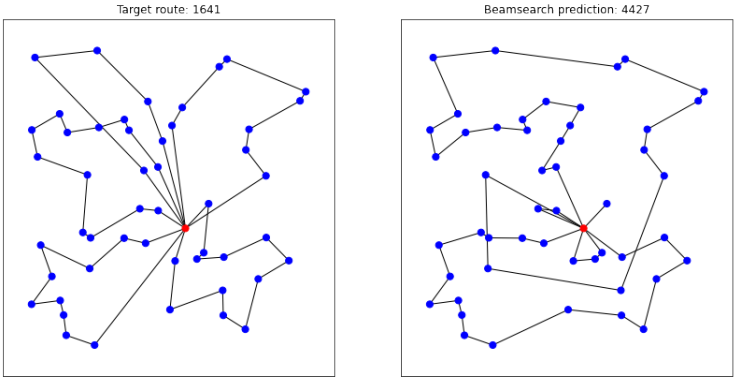
Figure B2: Example solutions of the TSP20 model. With an average optimality gap of 0.18%, our model will generally find the same solution as OR-tools.



(a)



(b)



(c)

Figure B3: Example solutions of the VRP50 model. While OR-tools generally manages to find solutions where all sub-routes are close in their lengths with good separation between them, our model struggles, finding solutions with a few very long sub-routes with long jumps and unnecessary overlap between each other.