Attention Solves Your TSP, Approximately

Wouter Kool
University of Amsterdam
w.w.m.kool@uva.nl

Herke van Hoof
University of Amsterdam
h.c.vanhoof@uva.nl

Max Welling
University of Amsterdam
m.welling@uva.nl

Abstract

The development of efficient (heuristic) algorithms for practical combinatorial optimization problems is costly, so we want to automatically learn them instead. We show the feasibility of this approach on the important Travelling Salesman Problem (TSP). We learn a heuristic algorithm that uses a Neural Network policy to construct a tour. As an alternative to the Pointer Network, our model is based entirely on (graph) attention layers and is invariant to the input order of the nodes. We train the model efficiently using REINFORCE with a simple and robust baseline based on a deterministic (greedy) rollout of the best policy so far. We significantly improve over results from previous works that consider learned heuristics for the TSP, reducing the optimality gap for a single tour construction from 1.51% to 0.32% for instances with 20 nodes, from 4.59% to 1.71% for 50 nodes and from 6.89% to 4.43% for 100 nodes. Additionally, we improve over a recent Reinforcement Learning framework for two variants of the Vehicle Routing Problem (VRP).

1 Introduction

Imagine yourself traveling to a scientific conference. The field is popular, and surely you do not want to miss out on anything. You have selected several posters you want to visit, and naturally you must return to the place where you are now: the coffee corner. In which order should you visit the posters, to minimize your time walking around? This is the Travelling Scientist Problem (TSP).

You realize that your problem is mathematically equivalent to the Travelling Salesman Problem (conveniently also abbreviated as TSP). This seems discouraging as you know people have studied the problem and it is (NP-)hard [6]. Fortunately, complexity theory analyzes the worst-case scenario, and your Bayesian view does not consider this very likely. In particular, you have a strong prior: the posters will probably be laid out regularly. Therefore, you want a specialized algorithm that solves not any, but *this* type of problem instance well. As you have some months left, you can prepare your algorithm upfront. As a machine learner, you wonder whether your algorithm can be learned?

1.1 Motivation

Machine learning algorithms have replaced humans as the engineers of algorithms to solve various tasks. A decade ago, computer vision algorithms used hand-crafted features but today they are learned *end-to-end* by Deep Neural Networks (DNNs). DNNs have outperformed classic approaches in speech recognition, machine translation, image captioning and other problems, by learning from data [13]. While DNNs are mainly used to make *predictions*, Reinforcement Learning (RL) has enabled algorithms to learn to make *decisions*, either by interacting with an environment to learn to play Atari games [14], or by inducing knowledge through look-ahead search: this was used to master the game of Go [20].

The world is not a game, and we desire to train models that make decisions to solve real problems. These must learn to select good solutions for a problem from a combinatorially large set of potential solutions. Classically, approaches to this problem of *combinatorial optimization* can be divided into

Preprint. Work in progress.

exact methods, that guarantee finding optimal solutions, and *heuristics*, that trade off optimality for computational cost, although exact methods can use heuristics internally and vice versa. Heuristics are typically expressed in the form of rules, which can be interpreted as policies to make decisions. We believe that these policies can be parameterized using DNNs, and be trained to obtain new and stronger algorithms for many different combinatorial optimization problems, similar to the way DNNs have boosted performance in the applications mentioned before.

1.2 Attention model trained with REINFORCE

In this paper, we take a step forward towards our goal of learning algorithms for combinatorial optimization problems. Our contributions are the following:

- We introduce a model based on (graph) attention layers that constructs a solution for TSP by outputting input elements as a sequence. Our model is an alternative to the Pointer Network (PN) [25] and is invariant to the order of the input elements. It can be generalized to problems defined on general graphs.
- We show how to efficiently train our model using REINFORCE with a baseline similar to Rennie et al. [18]. This simple and robust baseline is based on a deterministic (greedy) rollout of the best policy found during training so far.
- We show that our method is able to learn a strong heuristic that provides a good solution to the 2D Euclidean TSP, while constructing only *a single tour*. We improve significantly over other methods that learned algorithms for TSP as well as general baselines that construct a single solution. We show the generalizing potential of our method by learning algorithms for two variants of the Vehicle Routing Problem (VRP), and significantly improve over recent work [15].

We want to emphasize that we do not claim that the method presented here is currently a feasible alternative to state-of-the-art (non-learned) mathematical programming techniques such as implemented in Concorde [1] and Gurobi [7], but we interpret our results as evidence that there is structure in combinatorial optimization problems that can effectively be learned and used to find solutions to unseen problem instances. This shows the potential of this approach and motivates further research.

2 Related work

The application of Neural Networks (NNs) for optimizing decisions in combinatorial optimization problems dates back to Hopfield and Tank [9], who applied a Hopfield-network for solving small TSP instances. NNs have been applied to many related problems [21], although in most cases in an *online* manner, starting 'from scratch' and 'learning' a solution for every instance. More recently, (D)NNs have also been used in an *offline* manner to learn about an entire class of problem instances.

Vinyals et al. [25] introduce the Pointer Network (PN) as a model that uses attention to output a permutation of the input, and train this model offline to solve the (Euclidean) TSP, supervised by example solutions. Upon test time, their beam search procedure filters invalid tours. Bello et al. [3] introduce an Actor-Critic algorithm to train the PN without supervised solutions. They consider each instance as a training sample and use the cost (tour length) of a sampled solution for an unbiased Monte-Carlo estimate of the policy gradient. They introduce extra model depth in the decoder by an additional *glimpse* [24] at the embeddings, masking nodes already visited. For small instances ($n = 20$), they get close to the results by Vinyals et al. [25], they improve for $n = 50$ and additionally include results for $n = 100$. Nazari et al. [15] replace the LSTM encoder of the PN by element-wise projections, such that the updated embeddings after state-changes can be effectively computed. They apply this model on the Vehicle Routing Problem (VRP) with split deliveries and a stochastic variant.

Dai et al. [4] do not use a separate encoder and decoder, but a single model based on graph embeddings. They train the model to output the *order* in which nodes are *inserted* into a partial tour, using a helper function to insert at the best possible location. Their DQN [14] training method with $n = 1$ trains the algorithm per step and incremental rewards provided to the agent at every step effectively encourage greedy behavior. As mentioned in their appendix, they use the negative of the the reward, which combined with discounting encourages the agent to insert the farthest nodes first, which is known to be an effective heuristic [19].

Nowak et al. [16] train a Graph Neural Network in a supervised manner to directly output a tour as an adjacency matrix, which is converted into a feasible solution by a beam search. The model is non-autoregressive, so cannot condition its output on the partial tour and the authors report an optimality gap of 2.7% for $n = 20$, worse than autoregressive approaches mentioned in this section. Kaempfer and Wolf [11] train a model based on the Transformer [22] architecture that outputs a fractional solution to the multiple TSP (mTSP). The result can be seen as a solution to the linear relaxation of the problem and they use a beam search to obtain a feasible integer solution.

Finally, we note that the mentioned papers have not been able to improve over farthest insertion, a simple heuristic that greedily inserts the most distant node into the tour at the best possible location and is known empirically to outperform nearest or random order insertion [19]. We show that we can consistently outperform this baseline and the approaches mentioned here in Section 5.1.

3 Attention model

We define a problem instance s as a graph with n nodes, where node $i \in \{1, \dots, n\}$ is represented by features $\mathbf{x}_i$. For TSP, the graph is fully connected (with self-connections) and $\mathbf{x}_i$ is the coordinate of node i. We define a solution (tour) $\boldsymbol{\pi} = (\pi_1, \dots, \pi_n)$ as a permutation of the nodes, so $\pi_t \in \{1, \dots n\}$ and $\pi_t \neq \pi_{t'}\ \forall t \neq t'$. Our attention based encoder-decoder model defines a stochastic policy $p(\boldsymbol{\pi}|s)$ for selecting a solution $\boldsymbol{\pi}$ given a problem instance s. It is factorized and parameterized by $\boldsymbol{\theta}$ as

$$p_\theta(\boldsymbol{\pi}|s) = \prod_{t=1}^{n} p_{\boldsymbol{\theta}}(\pi_t|s, \boldsymbol{\pi}_{1:t-1}). \qquad (1)$$

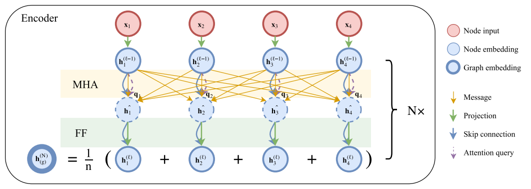


Figure 1: Attention based encoder. Input nodes are embedded and processed by N sequential layers, each consisting of a multi-head attention (MHA) and node-wise feed-forward (FF) sublayer. The embedding of the graph is computed as the mean of all node embeddings. Best viewed in color.

The encoder produces embeddings of all input nodes. The decoder produces the sequence $\boldsymbol{\pi}$ of input nodes, one node at a time. It takes as input the encoder embeddings and a problem specific mask and context. For TSP, when a partial tour has been constructed, it cannot be changed and the 'remaining' problem is to find a path from the last node, through all unvisited nodes, to the first node. The order and coordinates of other nodes already visited are irrelevant. To know the first and last node, the decoder context consists (next to the graph embedding) of embeddings of the first and last node. Similar to Bello et al. [3], the decoder observes a mask to know which nodes have been visited.

3.1 Encoder

The encoder that we use (Figure 1) is similar to the encoder used in the Transformer architecture by Vaswani et al. [22], but we do not use positional encoding such that the resulting node embeddings are invariant to the input order. From the d_x-dimensional input features $\mathbf{x}_i$ (for TSP d_x = 2), the encoder computes initial d_h-dimensional node embeddings $\mathbf{h}_i^{(0)}$ (we use $d_\text{h} = 128$) through a learned linear projection with parameters W^x and $\mathbf{b}^\text{x}$: $\mathbf{h}_i^{(0)} = W^\text{x}\mathbf{x}_i + \mathbf{b}^\text{x}$. The embeddings are updated using N attention layers, each consisting of two sublayers. We denote with $\mathbf{h}_i^{(\ell)}$ the node embeddings produced by layer $\ell \in \{1, .., N\}$. The encoder computes an aggregated embedding $\bar{\mathbf{h}}^{(N)}$ of the input graph as the mean of the final node embeddings $\mathbf{h}_i^{(N)}$: $\bar{\mathbf{h}}^{(N)} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{h}_i^{(N)}$. Both the node embeddings $\mathbf{h}_i^{(N)}$ and the graph embedding $\bar{\mathbf{h}}^{(N)}$ are used as input to the decoder.

Attention layer Following the Transformer architecture [22], each attention layer consist of two sublayers: a multi-head attention (MHA) layer that executes message passing between the nodes and an node-wise fully connected feed-forward (FF) layer. Each sublayer adds a skip-connection [8] and batch normalization (BN) [10] (which we found to work better than Layer Normalization [2]):

$$\hat{\mathbf{h}}_i = \text{BN}^\ell\left(\mathbf{h}_i^{(\ell-1)} + \text{MHA}_i^\ell\left(\mathbf{h}_1^{(\ell-1)}, \dots, \mathbf{h}_n^{(\ell-1)}\right)\right) \qquad (2)$$

$$\mathbf{h}_i^{(\ell)} = \text{BN}^\ell\left(\hat{\mathbf{h}}_i + \text{FF}^\ell(\hat{\mathbf{h}}_i)\right). \qquad (3)$$

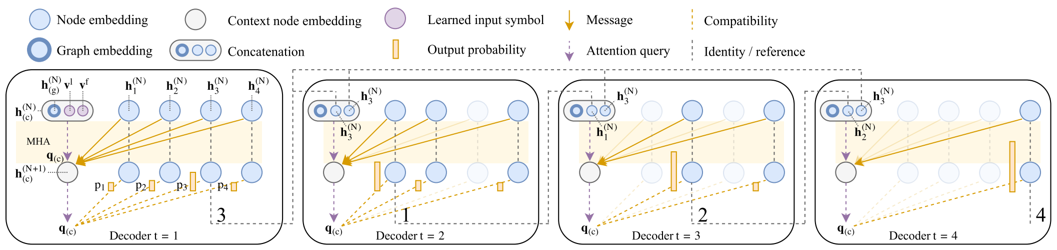


Figure 2: Attention based decoder for the TSP problem. The decoder takes as input the graph embedding and node embeddings. At each time step t, the context consist of the graph embedding and the embeddings of the first and last (previously output) node of the partial tour, where learned placeholders are used if $t = 1$. Nodes already in the output are masked. The example shows how a tour $\boldsymbol{\pi} = (3, 1, 2, 4)$ is constructed. Best viewed in color.

We superscript BN, FF and MHA with the layer index ℓ to indicate that the layers do *not* share parameters. The MHA sublayer uses $M = 8$ heads with dimensionality $\frac{d_h}{M} = 16$, and the FF sublayer has one hidden (sub)sublayer with dimension 512 and ReLu activation. See Appendix A for details.

3.2 Decoder

Decoding happens sequentially, and at timestep $t \in \{1, \dots n\}$, the decoder outputs the node π_t based on the embeddings from the encoder and the outputs $\pi_{t'}$ generated at time $t' < t$. During decoding, we augment the graph with a special *context node* (c) to represent the decoding context. The decoder computes an attention (sub)layer on top of the encoder, but with messages only to the context node for efficiency.[1] The final probabilities are computed using a single-head attention mechanism. See Figure 2 for an illustration of the decoding process.

Context embedding The context of the decoder at time t comes from the encoder and the output up to time t. As mentioned, for the TSP it consists of the embedding of the graph, the previous (last) node π_{t-1} and the first node π_1. For $t = 1$ we use learned d_h-dimensional parameters $\mathbf{v}^\text{l}$ and $\mathbf{v}^\text{f}$ as input placeholders:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \mathbf{h}_{\pi_1}^{(N)}\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{v}^\text{l}, \mathbf{v}^\text{f}\right] & t = 1. \end{cases} \tag{4}$$

Here $[\cdot, \cdot, \cdot]$ is the horizontal concatenation operator and we write the $(3 \cdot d_\text{h})$-dimensional result vector as $\mathbf{h}_{(c)}^{(N)}$ to indicate we interpret it as the embedding of the special context node (c) and use the superscript (N) to align with the node embeddings $\mathbf{h}_i^{(N)}$. We could project the embedding back to d_h dimensions, but we absorb this transformation in the parameter W^Q in Equation (5).

Now we compute a new context node embedding $\mathbf{h}_{(c)}^{(N+1)}$ using the (M-head) attention mechanism described in Appendix A. The keys and values come from the node embeddings $\mathbf{h}_i^{(N)}$, but we only compute a single query $\mathbf{q}_{(c)}$ (per head) from the context node (we omit the (N) for readability):

$$\mathbf{q}_{(c)} = W^Q \mathbf{h}_{(c)} \quad \mathbf{k}_i = W^K \mathbf{h}_i, \quad \mathbf{v}_i = W^V \mathbf{h}_i. \tag{5}$$

We compute the compatibility of the query with all nodes, and mask nodes already in the output:

$$u_{(c)j} = \begin{cases} \frac{\mathbf{q}_{(c)}^T \mathbf{k}_j}{\sqrt{d_\text{k}}} & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{6}$$

Here $d_\text{k} = \frac{d_\text{h}}{M}$ is the query/key dimensionality (see Appendix A). Again, we compute $u_{(c)j}$ and $\mathbf{v}_i$ for $M = 8$ heads and compute the final multi-head attention value for the context node using Equations (12)–(14) from Appendix A, but with (c) instead of i. This mechanism is similar to our encoder, but does not use skip-connections, batch normalization or the feed-forward sublayer for maximal efficiency. The result $\mathbf{h}_{(c)}^{(N+1)}$ is similar to the *glimpse* value described by Bello et al. [3].

[1] $n \times n$ attention between all nodes is expensive to compute in every step of the decoding process.

Calculation of log-probabilities In order for the model to compute output probabilities $p_{\boldsymbol{\theta}}(\pi_t|s, \boldsymbol{\pi}_{1:t-1})$ in Equation (1), we add one final decoder layer with a *single* attention head ($M = 1$ so $d_{\mathrm{k}} = d_{\mathrm{h}}$). For this layer, we *only* compute the compatibilities $u_{(c)j}$ using Equation (6), but following Bello et al. [3] we clip the result (before masking!) within $[-C, C]$ (C = 10) using a $\tanh$:

$$u_{(c)j} = \begin{cases} C \cdot \tanh\left(\frac{\mathbf{q}_{(c)}^T \mathbf{k}_j}{\sqrt{d_{\mathrm{k}}}}\right) & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{7}$$

We interpret these compatibilities as unnormalized log-probabilities (logits) and compute the final output probability vector $\mathbf{p}$ using a softmax (similar to Equation (12) in Appendix A):

$$p_i = p_{\boldsymbol{\theta}}(\pi_t = i|s, \boldsymbol{\pi}_{1:t-1}) = \frac{e^{u_{(c)i}}}{\sum_j e^{u_{(c)j}}}. \tag{8}$$

4 REINFORCE with rollout baseline

In Section 3 we defined our model that given an instance s defines a probability distribution $p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)$, which we can sample from to obtain a solution (tour) $\boldsymbol{\pi}|s$. In order to train our model, we define the loss $\mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}\left[L(\boldsymbol{\pi})\right]$: the expectation of the tour length $L(\boldsymbol{\pi})$. We optimize $\mathcal{L}$ by gradient descent, using the REINFORCE [26] gradient estimator with baseline $b(s)$:

$$\nabla \mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}\left[\left(L(\boldsymbol{\pi}) - b(s)\right)\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)\right]. \tag{9}$$

A good baseline $b(s)$ reduces the variance of the estimator and increases the speed of learning. A simple example is an exponential moving average $b(s) = M$ with *decay* β. Here $M = L(\boldsymbol{\pi})$ in the first iteration and gets updated as $M \leftarrow \beta M + (1-\beta)L(\boldsymbol{\pi})$ in subsequent iterations. A popular alternative is the use of a learned value function (critic) $\hat{v}(s, \boldsymbol{w})$, where the parameters $\boldsymbol{w}$ are learned from the observations $(s, L(\boldsymbol{\pi}))$. However, getting such *actor-critic* algorithms to work is non-trivial.

4.1 Rollout as baseline

We propose to use a rollout baseline in a way that is similar to self-critical training by Rennie et al. [18], but with periodic updates of the baseline policy. It is summarized as follows: define $b(s)$ to be the cost of a solution from a deterministic rollout of the algorithm defined by the best model so far.

Motivation The goal of a baseline is to estimate the difficulty of the instance s, such that it can relate to the cost $L(\boldsymbol{\pi})$ to estimate the advantage of the solution $\boldsymbol{\pi}$ selected by the model. We make the following key observation: *The difficulty of an instance can (on average) be estimated by the performance of an algorithm applied to it.* This follows from the assumption that (on average) an algorithm will have a higher cost on instances that are more difficult. Therefore we form a baseline by applying (rolling out) the algorithm defined by our model during training. To eliminate variance we force the result to be deterministic by selecting greedily the action with maximum probability.

Determining the rollout policy As the model changes during training, we further stabilize the baseline by freezing the rollout (baseline) policy $p_{\boldsymbol{\theta}^{BL}}$ for a fixed amount of steps (every epoch), which can be regarded similar to freezing of the target Q-network in DQN [14]. Since a stronger algorithm defines a stronger baseline, we evaluate the current policy against the rollout policy at the end of every epoch, and replace the parameters $\boldsymbol{\theta}^{BL}$ of the rollout policy only if the improvement is significant as determined by a paired t-test with $\alpha = 5\%$, on a separate (evaluation) dataset with 10000 instances. If the rollout policy is updated, we sample a new dataset to prevent overfitting.

Analysis Using the rollout policy as baseline $b(s)$, the function $L(\boldsymbol{\pi}) - b(s)$ is negative if the sampled solution $\boldsymbol{\pi}$ improves over the greedy rollout solution, causing actions to be reinforced, and vice versa. Combined with the evaluation at the end of each epoch, this challenges the model to improve over itself, and we see strong similarities with the self-play improvement scheme that was successfully applied to master the game of Go [20].

Algorithm We use Adam [12] as optimizer and summarize our training procedure in Algorithm 1.

Efficiency Each rollout constitutes an additional forward pass, theoretically increasing computation by 50%. However, as the baseline policy is fixed for an epoch, we can sample the data and compute baselines per epoch using larger batch sizes, allowed by the reduced memory requirement as the computations can run in pure inference mode. Empirically we find that it adds only 25% (see Appendix D), taking up 20% of total computation. If desired, the baseline rollout can be computed in parallel such that there is no increase in time per iteration, as an easy way to benefit from an additional GPU.

Algorithm 1 REINFORCE with Rollout Baseline

```
1: Input: number of epochs $E$, steps per epoch $T$, batch size $B$,
   significance $\alpha$
2: Init $\boldsymbol{\theta}$, $\boldsymbol{\theta}^{BL} \leftarrow \boldsymbol{\theta}$
3: for $epoch = 1, \dots, E$ do
4:     for $step = 1, \dots, T$ do
5:         $s_i \leftarrow$ RandomInstance() $\forall i \in \{1, \dots, B\}$
6:         $\boldsymbol{\pi}_i \leftarrow$ SampleRollout($s_i, p_{\boldsymbol{\theta}}$) $\forall i \in \{1, \dots, B\}$
7:         $\boldsymbol{\pi}_i^{BL} \leftarrow$ GreedyRollout($s_i, p_{\boldsymbol{\theta}^{BL}}$) $\forall i \in \{1, \dots, B\}$
8:         $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \left( L(\boldsymbol{\pi}_i) - L(\boldsymbol{\pi}_i^{BL}) \right) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}_i)$
9:         $\boldsymbol{\theta} \leftarrow$ Adam($\boldsymbol{\theta}, \nabla \mathcal{L}$)
10:    end for
11:    if OneSidedPairedTTest($p_{\boldsymbol{\theta}}, p_{\boldsymbol{\theta}^{BL}}$) $< \alpha$ then
12:        $\boldsymbol{\theta}^{BL} \leftarrow \boldsymbol{\theta}$
13:    end if
14: end for
```

5 Experiments

We train separate models TSP20, TSP50 and TSP100 for TSP instances with 20, 50 and 100 nodes respectively. All parameters (weights and biases) are initialized uniformly within $(-1/\sqrt{d}, 1/\sqrt{d})$, where d is the input dimension. Every epoch we process 2500 batches of size 512, for a total of 1.28M instances, which takes 5:30 minutes for TSP20, 16:20 for TSP50 (single GPU 1080Ti) and 27:30 for TSP100 (on 2 1080Ti's). We report results averaged over two random seeds, training for 100 epochs with a constant learning rate $\eta = 10^{-4}$. We found $N = 3$ layers in the encoder to be a good trade-off between quality of results and complexity.[2] With the rollout baseline, we use an exponential baseline ($\beta = 0.8$) during the first epoch, to stabilize initial learning, although in many cases learning also succeeds without this 'warmup'. Our code in PyTorch [17] is publicly available.[3]

Training data is generated on the fly while validation data (used to report progress over time) and test data consists of 10000 instances. Coordinates are generated uniformly at random in the unit square. We compare results against the optimal solution that we compute using Gurobi [7] with a MIP formulation based on lazy subtour elimination constraints[4] (MIP tolerance $10^{-4} = 0.01\%$). Rather than reporting the approximation ratio $\frac{c}{c^*}$ we report the average[5] *optimality gap* $\frac{c-c^*}{c^*} = \frac{c}{c^*} - 1$.

5.1 Results

We focus on the results when constructing a single tour by greedily selecting the next node with maximum assigned probability. This is the best indicator of the quality of the decisions learned by our model. Table 1a shows how we outperform all baseline models (for details see Appendix C) that construct a single tour, as well as learned algorithms from previous work. We also test generalization performance on different n than trained for, which we plot in Figure 3. The train sizes are indicated with vertical marker bars. The models generalize when tested on different sizes, although quality degrades as the difference becomes bigger, which can be expected as there is *no free lunch* [27]. Since the architectures are the same, these differences mean the models learn to specialize on the problem sizes trained for. We can make a strong overall algorithm by selecting the trained model with highest validation performance for each instance size n (marked in Figure 3 by the red bar).

Most methods for TSP rely on systematically or heuristically searching the search space and consider many solutions while doing so. It is encouraging that we can get good results using a different approach that greedily constructs only *a single solution*. This approach can even be used in situations where it is infeasible or impractical to consider many solutions, for instance when evaluation of a solution is costly, or in the stochastic setting described by Nazari et al. [15] where the problem is not fully observed up front. For completeness, we also include results when we use our model to consider multiple solutions, and we compare against baselines that also consider multiple solutions (by sampling or search). We sample 1280 solutions (in under one second on a single 1080Ti GPU) and report the shortest tour length. Naturally, the difference with other approaches is diluted by sampling

[2]See Appendix D for results of all runs and discussions on the influence of the schedule for η and value of N.

[3]`https://github.com/wouterkool/attention-tsp`

[4]`http://examples.gurobi.com/traveling-salesman-problem/`

[5]We compute the average optimality gap using the ratio of averages: $\frac{\frac{1}{m}\sum_{i=1}^{m} c_i}{\frac{1}{m}\sum_{i=1}^{m} c_i^*} - 1$.

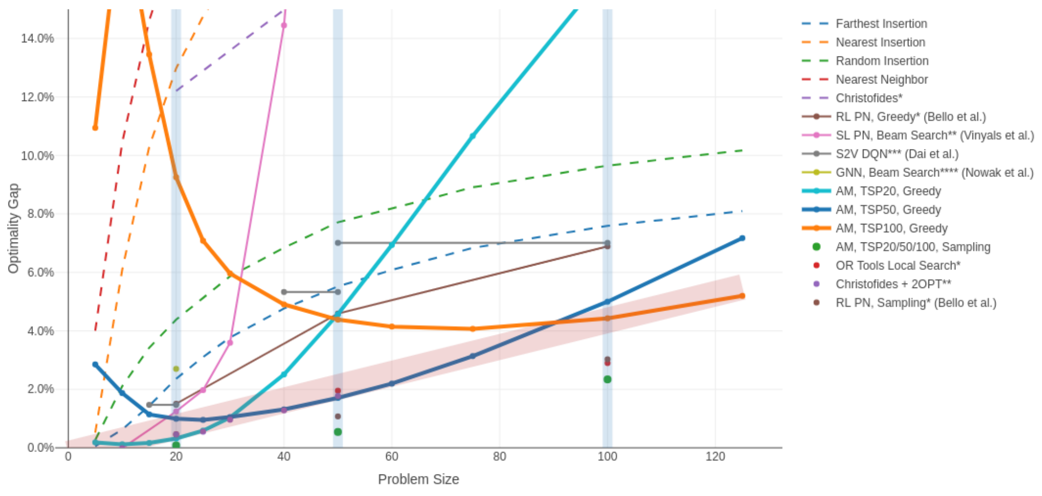


Figure 3: Optimality gap of different methods as a function of problem size $n \in \{5, 10, 15, 20, 25, 30, 40, 50, 75, 100, 125\}$. General baselines are drawn using dashed lines while learned algorithms are drawn with a solid line. Algorithms (general and learned) that perform search or sampling are plotted without connecting lines for clarity. The *, **, *** and **** indicate that values are reported from Bello et al. [3], Vinyals et al. [25], Dai et al. [4] and Nowak et al. [16] respectively. Best viewed in color.

many solutions (as in the limit even a random policy will sample good solutions occasionally), but we still improve over previous work and baselines others have reported as can be seen in Table 1b. For completeness, we also added these results in Figure 3, where we do not connect the dots to prevent cluttering and to make the distinction with methods that consider only a single solution clear.

Optimizing the run time of our method was not a priority. Nevertheless, for practical purposes, our method is fast: computing a single instance for TSP100 takes 0.03s on a 6850K CPU, whereas we solve 10000 instances using a single 1080Ti in 5s (0.0005s/instance). Our method is $O(n^2)$, and given sufficient parallelism runs in $O(n)$, which is the same for most baselines, although naturally, the constant is larger for our method. Exact runtimes of different methods can vary by multiple orders of magnitude as a result of implementation (Python vs. C++) and hardware (GPU vs CPU), thus runtimes cannot always directly be compared. As a rough indication, Bello et al. [3] report 0.01s 'averaged over the test set' (K80 GPU). For OR Tools, Bello et al. [3] report 0.1s on Intel Haswell CPU, which naturally can be traded off against the quality of the result. Computing an optimal solution via Gurobi takes around 1.5s for TSP100 on a 6600U CPU.

Table 1: Results of our Attention Model versus baseline learned algorithms and general heuristics.

(a) Main results of our method, compared against $O(n^2)$ baselines and related work. The result of Christofides' algorithm indicated with * is reported byBello et al. [3]. Results of Dai et al. [4], indidated with **, are for 15-20, 40-50 and 50-100 node graphs.

Method / Problem	TSP20	TSP50	TSP100
Optimal	3.83	5.69	7.76
Nearest Insertion	4.33 (12.98%)	6.78 (19.13%)	9.46 (21.80%)
Random Insertion	4.00 (4.38%)	6.13 (7.71%)	8.51 (9.65%)
Farthest Insertion	3.92 (2.36%)	6.00 (5.52%)	8.35 (7.59%)
Nearest Neighbor	4.50 (17.47%)	6.98 (22.75%)	9.70 (24.98%)
Christofides*	4.30 (12.21%)	6.62 (16.37%)	9.18 (18.23%)
Vinyals et al. [25]	3.88 (1.25%)	7.66 (34.65%)	-
Bello et al. [3]	3.89 (1.51%)	5.95 (4.59%)	8.30 (6.89%)
Dai et al. [4]**	3.89 (1.47%)	5.99 (5.33%)	8.31 (7.01%)
Nowak et al. [16]	3.93 (2.7%)	-	-
AM, TSP20, Greedy	**3.84 (0.32%)**	5.95 (4.57%)	9.03 (16.32%)
AM, TSP50, Greedy	3.87 (0.99%)	**5.79 (1.71%)**	8.15 (5.00%)
AM, TSP100, Greedy	4.19 (9.26%)	5.94 (4.38%)	**8.11 (4.43%)**

(b) Additional results: best of 1280 sampled solutions, compared against baselines using search and sampling by Bello et al. [3]. The results marked with * and ** are reported by Bello et al. [3] and Vinyals et al. [25]. ($T = T^*$) indicates optimized softmax temperature.

Method / Problem	TSP20	TSP50	TSP100
Optimal	3.83	5.69	7.76
OR Tools Local Search*	3.85 (0.46%)	5.80 (1.95%)	7.99 (2.90%)
Christofides + 2OPT**	3.85 (0.46%)	5.79 (1.78%)	-
Bello et al. [3]	-	5.77 (1.43%)	8.00 (3.03%)
Bello et al. [3] ($T = T^*$)	-	5.75 (1.07%)	8.00 (3.03%)
AM, TSP20/50/100, Sampling	**3.84 (0.08%)**	**5.72 (0.54%)**	**7.95 (2.34%)**

5.2 Attention Model vs. Pointer Network and different baselines

Figure 4 compares the performance of the TSP20 Attention Model (AM) and our implementation of the Pointer Network (PN) during training. We use a validation set of size 10000 with greedy decoding, and compare to using an exponential ($\beta = 0.8$) and a critic[6] baseline. We used two random seeds and a decaying learning rate of $\eta = 10^{-3} \times 0.96^{\text{epoch}}$. This performs best for the PN, while for the AM results are similar to using $\eta = 10^{-4}$ (see Appendix D). This clearly illustrates how the improvement we obtain is the result of both the AM and the rollout baseline: the AM outperforms the PN using any baseline and the rollout baseline improves the quality and convergence speed for both AM and PN. For the PN with critic baseline, we are unable to reproduce the 1.5% reported by Bello et al. [3] (also when using an LSTM based critic), but our reproduction is closer than others have reported [4, 15]. In Figure 3 we compare against the original results. Compared to the rollout baseline, the exponential baseline is around 20% faster per epoch, whereas the critic baseline is around 13% slower (see Appendix D), so the picture does not change significantly if time is used as x-axis.

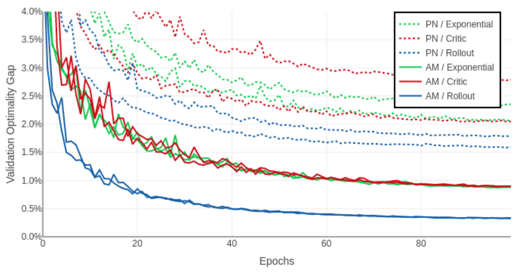


Figure 4: Held-out validation set optimality gap as a function of the number of epochs for the Attention Model (AM) and Pointer Network (PN) with different baselines (two different seeds).

6 Vehicle Routing Problem

To show the generalizing potential of our method, we make minor modifications to our model so it can be applied to the Capacitated Vehicle Routing Problem (CVRP). Details are provided in Appendix B. We implement the datasets described by Nazari et al. [15] and compare against their Reinforcement Learning (RL) framework and the strongest of their implemented baselines. Nazari et al. [15] consider the CVRP, as well as the Split Delivery (C)VRP (SDVRP) [5], which allows deliveries to be split over multiple visits to the same node. Any solution to CVRP is valid for SDVRP, so it is reasonable to expect SDVRP algorithms to have lower cost, but they may be harder to learn. We show that our method is applicable to both problems, and we observe similar solution quality with and without split deliveries. For both problems, our method (with greedy decoding) significantly improves over the method by Nazari et al. [15] (with greedy decoding). Also, it consistently outperforms all their reported baselines, and even their beam search for larger instances.

Table 2: Results for the CVRP and SDVRP, compared against the RL framework, strongest baselines and optimality by Nazari et al. [15].

Method	Problem	10	20	50	100
AM, Greedy	CVRP	4.65	6.40	10.98	16.78
RL, Greedy [15]	CVRP	4.84	6.59	11.39	17.23
RL, Beam Search (size 10) [15]	CVRP	4.68	6.40	11.15	16.96
Randomized Clarke-Wright [15]	CVRP	4.80	6.81	12.25	18.96
Randomized Sweep [15]	CVRP	5.00	7.08	12.96	20.33
OR-Tools [15]	CVRP	4.67	6.43	11.31	17.16
Optimal [15]	CVRP	4.55	6.10	-	-
AM, Greedy	SDVRP	4.66	6.40	10.94	16.84
RL, Greedy [15]	SDVRP	4.80	6.51	11.32	17.12
RL, Beam Search (size 10) [15]	SDVRP	4.65	6.34	11.08	16.86

7 Discussion

In this work we have introduced a model that is capable of learning a strong (single construction) heuristic for the TSP, specializing for different problem sizes. Additionally, we have shown that our model can learn heuristics for two variants of the VRP. We believe that our method is a powerful starting point for learning heuristic algorithms for other combinatorial optimization problems. In practice, operational constraints often lead to many variants of combinatorial problems for which no good (human-designed) heuristics are available. For such problems, the ability to learn heuristics could be of great practical value.

[6]The critic network architecture uses 3 attention layers similar to our encoder, after which the node embeddings are averaged and processed by an MLP with one hidden layer with 128 neurons and ReLu activation and a single output. We used the same learning rate as for the AM/PN in all experiments.

Acknowledgements

This research was funded by ORTEC Optimization Technology. The authors would like to thank Thomas Kipf for helpful discussions.

References

[1] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde TSP solver, 2006. URL `http://www.math.uwaterloo.ca/tsp/concorde/m`.

[2] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[4] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

[5] M. Dror, G. Laporte, and P. Trudeau. Vehicle routing with split deliveries. *Discrete Applied Mathematics*, 50(3):239–254, 1994.

[6] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness (series of books in the mathematical sciences), ed. *Computers and Intractability*, page 340, 1979.

[7] I. Gurobi Optimization. Gurobi optimizer reference manual, 2016. URL `http://www.gurobi.com`.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[10] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[11] Y. Kaempfer and L. Wolf. Learning the multiple traveling salesmen problem with permutation invariant pooling networks. *arXiv preprint arXiv:1803.09621*, 2018.

[12] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[15] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takáč. Deep reinforcement learning for solving the vehicle routing problem. *arXiv preprint arXiv:1802.04240*, 2018.

[16] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint arXiv:1706.07450*, 2017.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[18] S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel. Self-critical sequence training for image captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7008–7024, 2017.

[19] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. In *Fundamental Problems in Computing*, pages 45–69. Springer, 2009.

[20] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550 (7676):354, 2017.

[21] K. 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.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[24] O. Vinyals, S. Bengio, and M. Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[25] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[26] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[27] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

A Attention model details

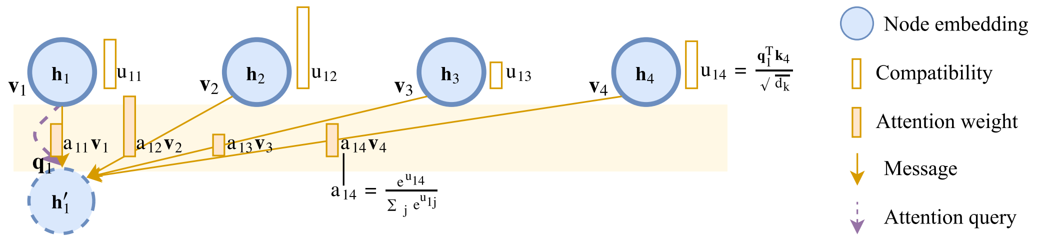


Figure 5: Illustration of weighted message passing using a dot-attention mechanism. Only computation of messages received by node 1 are shown for clarity. Best viewed in color.

Attention mechanism We interpret the attention mechanism by Vaswani et al. [22] as a weighted message passing algorithm between nodes in a graph. The weight of the message *value* that a node receives from a neighbor depends on the *compatibility* of its *query* with the *key* of the neighbor, as illustrated in Figure 5. Formally, we define dimensions d_k and d_v and compute the key $\mathbf{k}_i \in \mathbb{R}^{d_\text{k}}$, value $\mathbf{v}_i \in \mathbb{R}^{d_\text{v}}$ and query $\mathbf{q}_i \in \mathbb{R}^{d_\text{k}}$ for each node by projecting the embedding $\mathbf{h}_i$:

$$\mathbf{q}_i = W^Q \mathbf{h}_i \quad \mathbf{k}_i = W^K \mathbf{h}_i, \quad \mathbf{v}_i = W^V \mathbf{h}_i. \tag{10}$$

Here parameters W^Q and W^K are $(d_\text{k} \times d_\text{h})$ matrices and W^V has size $(d_\text{v} \times d_\text{h})$. From the queries and keys, we compute the compatibility $u_{ij} \in \mathbb{R}$ of the query $\mathbf{q}_i$ of node i with the key $\mathbf{k}_j$ of node j as the (scaled, see Vaswani et al. [22]) dot-product:

$$u_{ij} = \begin{cases} \frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_\text{k}}} & \text{if } i \text{ adjacent to } j \\ -\infty & \text{otherwise.} \end{cases} \tag{11}$$

In a general graph, defining the compatibility of non-adjacent nodes as $-\infty$ prevents message passing between these nodes. From the compatibilities u_{ij}, we compute the *attention weights* $a_{ij} \in [0, 1]$ using a softmax:

$$a_{ij} = \frac{e^{u_{ij}}}{\sum_{j'} e^{u_{ij'}}}. \tag{12}$$

Finally, the vector $\mathbf{h}_i'$ that is received by node i is the convex combination of messages $\mathbf{v}_j$:

$$\mathbf{h}_i' = \sum_j a_{ij} \mathbf{v}_j. \tag{13}$$

Multi-head attention As was noted by Vaswani et al. [22] and Veličković et al. [23], it is beneficial to have multiple attention heads. This allows nodes to receive different types of messages from different neighbors. Especially, we compute the value (13) $M = 8$ times with different parameters, using $d_\text{k} = d_\text{v} = \frac{d_\text{h}}{M} = 16$. We denote the result vectors by $\mathbf{h}_{im}'$ for $m \in 1, \dots, M$. These are projected back to a single d_h-dimensional vector using $(d_\text{h} \times d_\text{v})$ parameter matrices W_m^O. The final multi-head attention value for node i is a function of $\mathbf{h}_1, \dots, \mathbf{h}_n$ through $\mathbf{h}_{im}'$:

$$\text{MHA}_i(\mathbf{h}_1, \dots, \mathbf{h}_n) = \sum_{m=1}^{M} W_m^O \mathbf{h}_{im}'. \tag{14}$$

Feed-forward sublayer The feed-forward sublayer computes node-wise projections using a hidden (sub)sublayer with dimension $d_\text{ff} = 512$ and a ReLu activation:

$$\text{FF}(\hat{\mathbf{h}}_i) = W^{\text{ff},1} \cdot \text{ReLu}(W^{\text{ff},0} \hat{\mathbf{h}}_i + \boldsymbol{b}^{\text{ff},0}) + \boldsymbol{b}^{\text{ff},1}. \tag{15}$$

Batch normalization We use batch normalization with learnable d_h-dimensional affine parameters $\boldsymbol{w}^\text{bn}$ and $\boldsymbol{b}^\text{bn}$:

$$\text{BN}(\mathbf{h}_i) = \boldsymbol{w}^\text{bn} \odot \overline{\text{BN}}(\mathbf{h}_i) + \boldsymbol{b}^\text{bn}. \tag{16}$$

Here $\odot$ denotes the element-wise product and $\overline{\text{BN}}$ refers to batch normalization without affine transformation. We found that it is important to initialize the parameters small enough, e.g. uniform within $(-1/\sqrt{d_\text{h}}, 1/\sqrt{d_\text{h}})$.

B Vehicle Routing Problem

The Capacitated Vehicle Routing Problem (CVRP) is a generalization of the TSP in which case there is a depot and multiple routes should be created, each starting and ending at the depot. In our graph based formulation, we add a special depot node with index 0 and coordinates $\mathbf{x}_0$. A vehicle (route) has capacity $D > 0$ and each (regular) node $i \in \{1, \dots n\}$ has a demand $0 < \delta_i \leq D$. Each route starts and ends at the depot and the total demand in each route should not exceed the capacity, so $\sum_{i \in R_j} \delta_i \leq D$, where R_j is the set of node indices assigned to route j. Without loss of generality, we assume a normalized $\hat{D} = 1$ as we can use normalized demands $\hat{\delta}_i = \frac{\delta_i}{D}$.

The Split Delivery VRP (SDVRP) is a generalization of CVRP in which every node can be visited multiple times, and only a subset of the demand has to be delivered at each visit. Instances for both CVRP and SDVRP are specified in the same way: an instance with size n as a depot location $\mathbf{x}_0$, n node locations $\mathbf{x}_i, i = 1 \dots n$ and (normalized) demands $0 < \hat{\delta}_i \leq 1, i = 1 \dots n$.

B.1 Instance generation

We follow Nazari et al. [15] in the generation of instances for $n = 10, 20, 50, 100$, but normalize the demands by the capacities. The depot location as well as n node locations are sampled uniformly at random in the unit square. The demands are defined as $\hat{\delta}_i = \frac{\delta_i}{D_n}$ where δ_i is discrete and sampled uniformly from $\{1, \dots, 9\}$ and $D_{10} = 20$, $D_{20} = 30$, $D_{50} = 40$ and D_{100} = 50.

B.2 Attention Model for VRP

Encoder In order to allow our Attention Model to distinguish the depot node from the regular nodes, we use separate parameters W_0^{x} and $\mathbf{b}_0^{\mathrm{x}}$ to compute the initial embedding $\mathbf{h}_0^{(0)}$ of the depot node:

$$\mathbf{h}_i^{(0)} = \begin{cases} W_0^{\mathrm{x}} \mathbf{x}_i + \mathbf{b}_0^{\mathrm{x}} & i = 0 \\ W^{\mathrm{x}} \mathbf{x}_i + \mathbf{b}^{\mathrm{x}} & i = 1, \dots, n. \end{cases} \tag{17}$$

Capacity constraints To facilitate the capacity constraints, we keep track of the remaining demands $\hat{\delta}_{i,t}$ for the nodes $i \in \{1, \dots n\}$ and remaining vehicle capacity $\hat{D}_t$ at time t. At $t = 1$, these are initialized as $\hat{\delta}_{i,t} = \hat{\delta}_i$ and $\hat{D}_t = 1$, after which they are updated as follows (recall that π_t is the index of the node selected at decoding step t):

$$\hat{\delta}_{i,t+1} = \begin{cases} \max(0, \hat{\delta}_{i,t} - \hat{D}_t) & \pi_t = i \\ \hat{\delta}_{i,t} & \pi_t \neq i \end{cases} \tag{18}$$

$$\hat{D}_{t+1} = \begin{cases} \max(\hat{D}_t - \hat{\delta}_{\pi_t,t}, 0) & \pi_t \neq 0 \\ 1 & \pi_t = 0. \end{cases} \tag{19}$$

If we do not allow split deliveries, $\hat{\delta}_{i,t}$ will be either 0 or $\hat{\delta}_i$ for all t.

Decoder context The context for the decoder for the VRP at time t is the current/last location π_{t-1} and the remaining capacity $\hat{D}_t$. Compared to TSP, we do not need placeholders if $t = 1$ as the route starts at the depot and we do not need to provide information about the first node as the route should end at the depot:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \hat{D}_t\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_0^{(N)}, \hat{D}_t\right] & t = 1. \end{cases} \tag{20}$$

Masking The depot can be visited multiple times, but we do not allow it to be visited at two subsequent timesteps. Therefore, in both layers of the decoder, we change the masking for the depot $j = 0$ and define $u_{(c)0} = -\infty$ if (and only if) $t = 1$ or $\pi_{t-1} = 0$. The masking for the nodes depends on whether we allow split deliveries. Without split deliveries, we do not allow nodes to be visited if their remaining demand is 0 (if the node was already visited) or exceeds the remaining capacity, so for $j \neq 0$ we define $u_{(c)j} = -\infty$ if (and only if) $\hat{\delta}_{i,t} = 0$ or $\hat{\delta}_{i,t} > \hat{D}_t$. With split deliveries, we only forbid delivery when the remaining demand is 0, so we define $u_{(c)j} = -\infty$ if (and only if) $\hat{\delta}_{i,t} = 0$.

Split deliveries Without split deliveries, the remaining demand $\hat{\delta}_{i,t}$ is either 0 or $\hat{\delta}_i$, corresponding to whether the node has been visited or not, and this information is conveyed to the model via the masking of the nodes already visited. However, when split deliveries are allowed, the remaining demand $\hat{\delta}_{i,t}$ can take any value $0 \leq \hat{\delta}_{i,t} \leq \hat{\delta}_i$. This information cannot be included in the context node as it corresponds to individual nodes. Therefore we include it in the computation of the keys and values in both the attention layer (glimpse) and the output layer of the decoder, such that we compute queries, keys and values using:

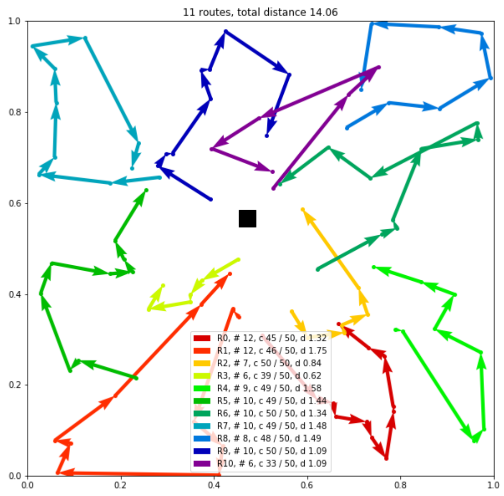
$$\mathbf{q}_{(c)} = W^Q \mathbf{h}_{(c)} \quad \mathbf{k}_i = W^K \mathbf{h}_i + W_d^K \hat{\delta}_{i,t}, \quad \mathbf{v}_i = W^V \mathbf{h}_i + W_d^V \hat{\delta}_{i,t}. \tag{21}$$

Here we W_d^K and W_d^V are $(d_k \times 1)$ parameter matrices and we define $\hat{\delta}_{i,t} = 0$ for the depot $i = 0$. Summing the projection of both $\mathbf{h}_i$ and $\hat{\delta}_{i,t}$ is equivalent to projecting the concatenation $[\mathbf{h}_i, \hat{\delta}_{i,t}]$ with a single $((d_h + 1) \times d_k)$ matrix W^K. However, using this formulation we only need to compute the first term once (instead for every t) and by the weight initialization this puts more importance on $\hat{\delta}_{i,t}$ initially (which is otherwise just 1 of $d_h + 1 = 129$ input values).

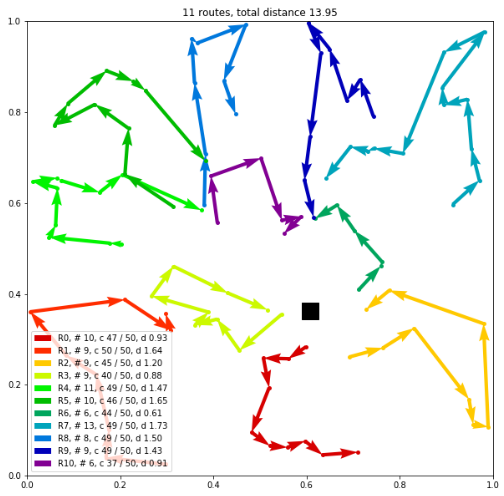
Training For the VRP, the length of the output of the model depends on the number of times the depot is visited. In general, the depot is visited multiple times, and in the case of SDVRP also some regular nodes are visited twice. Therefore the length of the solution is larger than n, which requires more memory such that we find it necessary to limit the batch size B to 256 for $n = 100$ (on 2 GPUs). To keep training times tractable and the total number of parameter updates equal, we still process 2500 batches per epoch, for a total of 0.64M training instances per epoch.

B.3 Example solutions

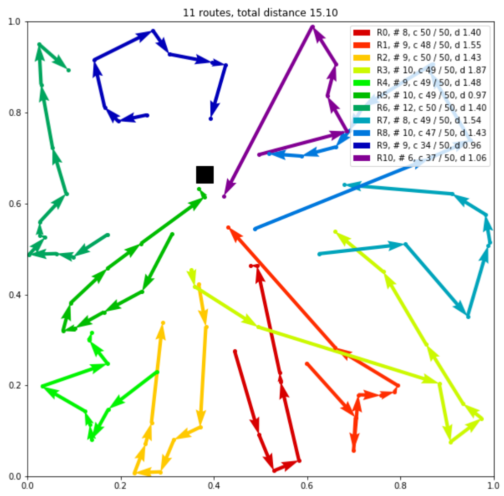
Figure 6 shows example solutions for the CVRP with $n = 100$ that were obtained by a single construction using the model with greedy decoding. These visualizations give insight in the heuristic that the model has learned. In general we see that the model constructs the routes from the bottom to the top, starting below the depot. Most routes are densely packed, except for the last route that has to serve some remaining (close to each other) customers. In most cases, the node in the route that is farthest from the depot is somewhere in the middle of the route, such that customers are served on the way to and from the farthest nodes. In some cases, we see that the order of stops within some individual routes is suboptimal, which means that the method will likely benefit from simple further optimizations on top, such as a beam search, a post-processing procedure based on local search (e.g. 2OPT) or solving the individual routes using a TSP solver.



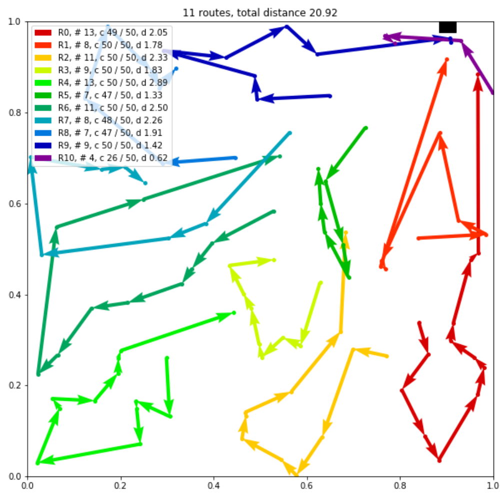
(a)



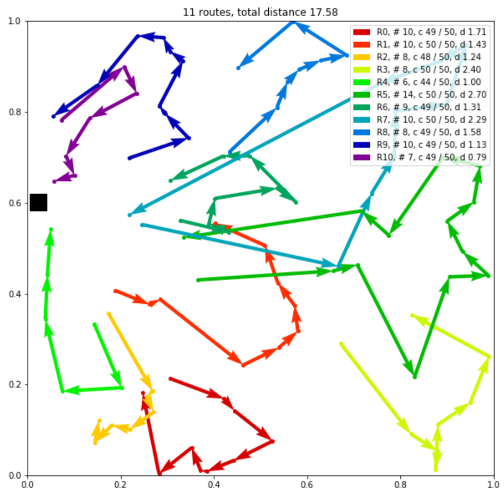
(b)



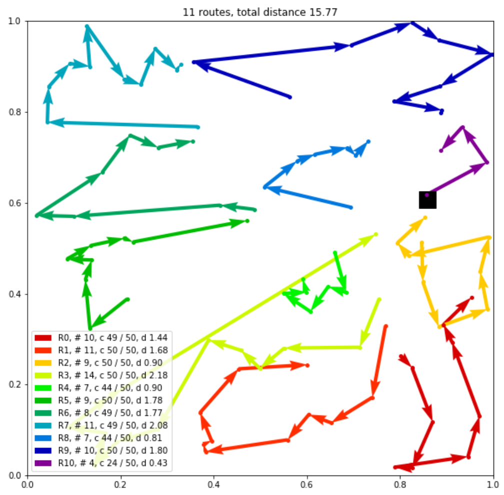
(c)



(d)



(e)



(f)

Figure 6: Example greedy solutions for the CVRP ($n = 100$). Edges from and to depot omitted for clarity. Legend order/coloring and arcs indicate the order in which the solution was generated. Legends indicate the number of stops, the used and available capacity and the distance per route.

C Details of baselines

This section describes details of the heuristics implemented for the TSP. All of the heuristics construct a single tour in a single pass, by extending a partial solution one node at the time.

C.1 Nearest neighbor

The nearest neighbor heuristic represents the partial solution as a *path* with a *start* and *end* node. The initial path is formed by a single node, selected randomly, which becomes the start node but also the end node of the initial path. In each iteration, the next node is selected as the node nearest to the end node of the partial path. This node is added to the path and becomes the new end node. Finally, after all nodes are added this way, the end node is connected with the start node to form a tour. In our implementation, for deterministic results we always start with the first node in the input, which can be considered random as the instances are generated randomly.

C.2 Farthest/nearest/random insertion

The insertion heuristics represent a partial solution as a *tour*, and extends it by *inserting* nodes one node at the time. In our implementation, we always insert the node using the *cheapest* insertion cost. This means that when node i is inserted, the place of insertion (between adjacent nodes j and k in the tour) is selected such that it minimizes the *insertion costs* $d_{ji} + d_{ik} - d_{jk}$, where d_{ji}, d_{ik} and d_{jk} represent the distances from node j to i, i to k and j to k, respectively.

The different variants of the insertion heuristic vary in the way in which the node which is inserted is selected. Let S be the set of nodes in the partial tour. *Nearest* insertion inserts the node i that is nearest to (any node in) the tour:

$$i^* = \arg\min_{i \notin S} \min_{j \in S} d_{ij}. \tag{22}$$

Farthest insertion inserts the node i such that the distance to the tour (i.e. the distance from i to the nearest node j in the tour) is maximized:

$$i^* = \arg\max_{i \notin S} \min_{j \in S} d_{ij}. \tag{23}$$

Random insertion inserts a random node. Similar to nearest neighbor, we consider the input order random so we simply insert the nodes in this order.

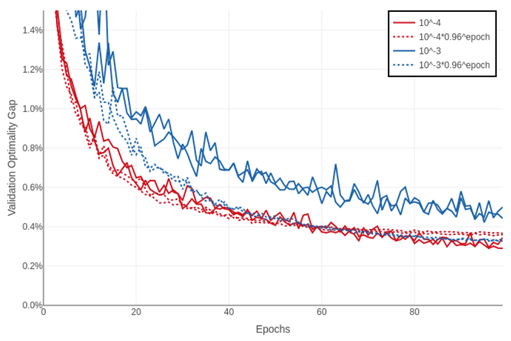
D Extended results

We found in general that using a larger learning rate of 10^{-3} works better with decay but may be unstable in some cases. A smaller learning rate 10^{-4} is more stable and does not require decay. This is illustrated in Figure 7, which shows validation results over time using both 10^{-3} and 10^{-4} with and without decay for TSP20 and TSP50 (2 seeds). As can be seen, without decay the method has not yet fully converged after 100 epochs and results may improve even further with longer training.

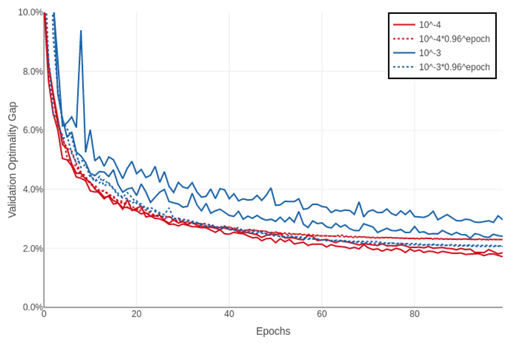
Table 3 shows the results for all runs using seeds 1234 and 1235 with the two different learning rate schedules. We did not run final experiments for $n = 100$ with the larger learning rate as we found training with the smaller learning rate to be more stable. It can be seen that in most cases the end results with different learning rate schedules are similar, except for the larger models ($N = 5$, $N = 8$) where some of the runs diverged using the larger learning rate. Experiments with different number of layers N show that $N = 3$ and $N = 5$ achieve best performance, and we find $N = 3$ is a good trade-off between quality of the results and computational complexity (runtime) of the model.

Table 3: Epoch durations and results and with different seeds and learning rate schedules for all experiments. All experiments compute 2500 batches per epoch (so with batch size $B = 256$ we use 0.64M instances per epoch to keep training times tractable).

	epoch	$\eta = 10^{-4}$		$\eta = 10^{-3} \times 0.96^{\text{epoch}}$	
	time	seed = 1234	seed = 1235	seed = 1234	seed = 1235
TSP20	5:30	3.85 (0.35%)	3.84 (0.29%)	3.84 (0.32%)	3.85 (0.33%)
TSP50	16:20	5.79 (1.77%)	5.78 (1.65%)	5.80 (1.98%)	5.80 (1.99%)
TSP100 (2GPUs)	27:30	8.12 (4.53%)	8.10 (4.33%)	-	-
N = 0	3:10	4.24 (10.59%)	4.24 (10.76%)	4.24 (10.63%)	4.24 (10.55%)
N = 1	3:50	3.87 (0.97%)	3.87 (1.03%)	3.87 (0.89%)	3.87 (0.90%)
N = 2	5:00	3.85 (0.40%)	3.85 (0.45%)	3.85 (0.38%)	3.85 (0.40%)
N = 3	5:30	3.85 (0.35%)	3.84 (0.29%)	3.84 (0.32%)	3.85 (0.33%)
N = 5	7:00	3.84 (0.26%)	3.84 (0.28%)	3.84 (0.29%)	10.44 (172.34%)
N = 8	10:10	3.84 (0.26%)	3.84 (0.32%)	10.44 (172.34%)	10.44 (172.34%)
AM / Exponential	4:20	3.87 (0.94%)	3.87 (0.94%)	3.87 (0.88%)	3.87 (0.90%)
AM / Critic	6:10	3.87 (0.97%)	3.87 (0.98%)	3.87 (0.91%)	3.87 (0.90%)
AM / Rollout	5:30	3.85 (0.35%)	3.84 (0.29%)	3.84 (0.32%)	3.85 (0.33%)
PN / Exponential	5:10	3.94 (2.88%)	3.94 (2.80%)	3.91 (2.08%)	3.92 (2.35%)
PN / Critic	7:30	3.95 (3.04%)	3.94 (2.94%)	3.91 (1.99%)	3.94 (2.82%)
PN / Rollout	6:40	3.93 (2.44%)	3.92 (2.31%)	3.89 (1.59%)	3.90 (1.79%)
CVRP10	5:20	4.64	4.65	4.66	4.66
CVRP20	9:40	6.40	6.41	6.40	6.42
CVRP50	26:40	10.98	10.98	10.98	10.95
CVRP100 ($B = 256$, 2GPUs)	37:00	16.76	16.80	-	-
SDVRP10	6:10	4.67	4.65	4.69	4.67
SDVRP20	11:10	6.39	6.40	6.39	6.36
SDVRP50	34:20	10.92	10.97	10.96	10.93
SDVRP100 ($B = 256$, 2GPUs)	45:30	16.84	16.83	-	-



(a) TSP20, four schedules for η (2 seeds)



(b) TSP50, four schedules for η (2 seeds)

Figure 7: Held-out validation set optimality gap as a function of the number of epochs for different η.