Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning

Paulo Roberto de O. da Costa P.R.D.OLIVEIRA.DA.COSTA@TUE.NL
Jason Rhuggenaath J.S.RHUGGENAATH@TUE.NL
Yingqian Zhang YQZHANG@TUE.NL
Alp Akcay A.E.AKCAY@TUE.NL
School of Industrial Engineering
Eindhoven University of Technology, 5612 AZ Eindhoven, Netherlands

Editors: Sinno Jialin Pan and Masashi Sugiyama

Abstract

Recent works using deep learning to solve the Traveling Salesman Problem (TSP) have focused on learning construction heuristics. Such approaches find TSP solutions of good quality but require additional procedures such as beam search and sampling to improve solutions and achieve state-of-the-art performance. However, few studies have focused on improvement heuristics, where a given solution is improved until reaching a near-optimal one. In this work, we propose to learn a local search heuristic based on 2-opt operators via deep reinforcement learning. We propose a policy gradient algorithm to learn a stochastic policy that selects 2-opt operations given a current solution. Moreover, we introduce a policy neural network that leverages a pointing attention mechanism, which unlike previous works, can be easily extended to more general k-opt moves. Our results show that the learned policies can improve even over random initial solutions and approach near-optimal solutions at a faster rate than previous state-of-the-art deep learning methods.

Keywords: Deep Reinforcement Learning, Combinatorial Optimization, Traveling Salesman Problem.

1. Introduction

The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem. In the TSP, given a set of locations (nodes) in a graph, we need to find the shortest tour that visits each location exactly once and returns to the departing location. The TSP is NP-hard (Papadimitriou, 1977) even in its Euclidean formulation, i.e., nodes are points in the 2D space. Classic approaches to solve the TSP can be classified in exact and heuristic methods. The former have been extensively studied using integer linear programming (Applegate et al., 2006) which are guaranteed to find an optimal solution but are often too computationally expensive to be used in practice. The latter are based on (meta)heuristics and approximate algorithms (Arora, 1998) that find solutions requiring less computational time, e.g., edge swaps such as k-opt (Helsgaun, 2009). Nevertheless, designed heuristics require specialized knowledge and their performances are often limited by algorithmic design decisions.

Recent works in machine learning and deep learning have focused on learning heuristics for combinatorial optimization problems (Bengio et al., 2018; Lombardi and Milano, 2018). For the TSP, both supervised learning (Vinyals et al., 2015; Joshi et al., 2019) and reinforcement learning (Bello and Pham, 2017; Wu et al., 2019; Kool et al., 2019; Deudon et al., 2018; Khalil et al., 2017) methods have been proposed. The idea is that a machine learning method could potentially learn better heuristics by extracting useful information directly from data, rather than having an explicitly programmed behavior. Most approaches to the TSP have focused on learning construction heuristics, i.e., methods that can generate a solution sequentially by extending a partial tour. These methods employed sequence representations (Vinyals et al., 2015; Bello and Pham, 2017), graph neural networks (Khalil et al., 2017; Joshi et al., 2019) and attention mechanisms (Kool et al., 2019; Deudon et al., 2018; Wu et al., 2019) resulting in high-quality solutions. However, construction methods still require additional procedures such as beam search, classical improvement heuristics and sampling to achieve such results. This limitation hinders their applicability as it is required to revert back to handcrafted improvement heuristics and search algorithms for state-of-the-art performance. Thus, learning improvement heuristics, i.e., when a solution is improved by local moves that search for better solutions, remains relevant. Here the idea is that if we can learn a policy to improve a solution, we can use it to get better solutions from a construction heuristic or even random solutions. Recently, a deep reinforcement learning method (Wu et al., 2019) has been proposed for such task, achieving near-optional results using node swap and 2-opt moves. However, the architecture has its output fixed by the number of possible moves, making it less favorable to expand to more general k-opt moves.

In this work, we propose a deep reinforcement learning algorithm trained via Policy Gradient to learn improvement heuristics based on 2-opt moves. Our architecture is based on a pointer attention mechanism (Vinyals et al., 2015) that outputs nodes sequentially for action selection. We introduce a reinforcement learning formulation to learn a stochastic policy of the next promising solutions, incorporating the search's history information by keeping track of the current best-visited solution. Our results show that we can learn policies for the Euclidean TSP that achieve near-optimal solutions even when starting from solutions of poor quality. Moreover, our approach can achieve better results than previous deep learning methods based on construction (Vinyals et al., 2015; Joshi et al., 2019; Kool et al., 2019; Deudon et al., 2018; Khalil et al., 2017; Bello and Pham, 2017) and improvement (Wu et al., 2019) heuristics. Compared to Wu et al. (2019), our method can be easily adapted to general k-opt and it is more sample efficient. In addition, policies trained on small instances can be reused on larger instances of the TSP. Lastly, our method outperforms other effective heuristics such as Google's OR-Tools (Perron and Furnon) and are close to optimal solutions.

2. Related Work

In machine learning, early works for the TSP have focused on Hopfield networks (Hopfield and Tank, 1985) and deformable template models (Angeniol et al., 1988). However, the performance of these approaches has not been on par with classical heuristics (La Maire and Mladenov, 2012). Recent deep learning methods have achieved high performance learning construction heuristics for the TSP. Pointer Networks (PtrNet) (Vinyals et al., 2015) learned a sequence model coupled with an attention mechanism trained to output TSP tours using solutions generated by Concorde (Applegate et al., 2006). In Bello and Pham (2017), the PtrNet was further extended to learn without supervision using Policy Gradient, trained to output a distribution over node permutations. Other approaches encoded instances via graph neural networks. A *structure2vec* (S2V) (Khalil et al., 2017) model was trained to output the ordering of partial tours using Deep Q-Learning (DQN). Later, graph attention was employed to a hybrid approach using 2-opt local search on top of tours trained via Policy Gradient (Deudon et al., 2018). Graph attention was extended in Kool et al. (2019) using REINFORCE (Williams, 1992) with a greedy rollout baseline, resulting in lower optimality gaps. Recently, the supervised approach was revisited using Graph Convolution Networks (GCN) (Joshi et al., 2019) learning probabilities of edges occurring in a TSP tour. It achieved state-of-the-art results up to 100 nodes whilst also combining with search heuristics.

It is important to previous methods to have additional procedures such as beam search, classical improvement heuristics and sampling to achieve good solutions. However, little attention has been posed on learning such policies that search for improved solutions. A recent approach, based on the transformer architecture (Wu et al., 2019), employed a Graph Attention Network (GAT) (Veličković et al., 2018) coupled with 2-opt and node swap operations. The limitations of this approach are related to the fixed output embeddings. These are vectors defined by the squared number of nodes, which makes expanding to general k-opt harder. Moreover, at inference it requires more samples than construction methods to achieve similar results. In contrast, we encode edge information using graph convolutions and use classical sequence encoding to learn tour representations. We decode these representations via a pointing attention mechanism to learn a stochastic policy of the action selection task. Our approach resembles classical 2-opt heuristics (Hansen and Mladenović, 2006) and can outperform previous deep learning methods in solution quality and sample efficiency.

3. Background

3.1. Travelling Salesman Problem

We focus on the 2D Euclidean TSP. Given an input graph, represented as a sequence of n locations in a two dimensional space $X = \{x_i\}_{i=1}^{n}$, where $x_i \in [0,1]^2$, we are concerned with finding a permutation of the nodes, i.e. a tour $S = (s_1, \ldots, s_n)$, that visits each node once (except the starting node) and has the minimum total length (cost). We define the cost of a tour as the sum of the distances (edges) between consecutive nodes in S as $L(S) = \|x_{s_n} - x_{s_1}\|_2 + \sum_{i=1}^{n-1} \left\|x_{s_i} - x_{s_{i+1}}\right\|_2$, where $\|\cdot\|_2$ denotes the ℓ_2 norm.

3.2. k-opt Heuristic for the TSP

Improvement heuristics enhance feasible solutions through a search procedure. A procedure starts at an initial solution S_0 and replaces a previous solution S_t by a better solution S_{t+1}. Local search methods such as the effective Lin-Kernighan-Helsgaun (LKH) (Helsgaun, 2009) heuristic perform well for the TSP. The procedure searches for k edge swaps (k-opt moves) that will be replaced by new edges resulting in a shorter tour. A simpler version (Lin and Kernighan, 1973) considers 2-opt (Figure 1) and 3-opt moves alternatives as these balance solution quality and the $O(n^k)$ complexity of the moves. Moreover, sequential pairwise operators such as k-opt moves can be decomposed in simpler l-opt ones, where $l < k$. For instance, sequential 3-opt operations can be decomposed into one, two or three 2-opt operations (Helsgaun, 2009). However, in local search algorithms, the quality of the initial solution usually affects the quality of the final solution, i.e. local search methods can easily get stuck in local optima (Hansen and Mladenović, 2006).

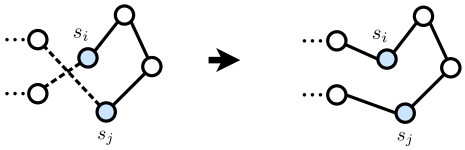

Figure 1: TSP solution before a 2-opt move (left), and after a 2-opt move (right). Replaced edges are represented in dashed lines. Note that the sequence $s_i, \dots, s_j$ is inverted.

To avoid local optima, different metaheuristics have been proposed including Simulated Annealing and Tabu Search. These work by accepting worse solutions to allow more exploration of the search space. In general, this strategy leads to better solution quality. However, metaheuristics still require expert knowledge and may have sub-optimal rules in their design. To tackle this limitation, we propose to combine machine learning and 2-opt operators to learn a stochastic policy to improve TSP solutions sequentially. A stochastic policy resembles a metaheuristic, sampling solutions in the e neighborhood of a given solution, potentially avoiding local minima. Our policy iterates over feasible solutions and the minimum cost solution is returned at the end. The main idea of our method is that by taking future improvements into account can potentially result it better policies than greedy heuristics.

4. Reinforcement Learning Formulation

Our formulation considers solving the TSP via 2-opt as a Markov Decision Process (MDP), detailed below. In our MDP, a given state $\bar{S}$ is composed of a tuple of the current solution (tour) S and the lowest-cost solution S' seen in the search. The proposed *neural architecture* (Section 5) approximates the stochastic policy $\pi_\theta(A|\bar{S})$, where θ represents trainable parameters. Each $A = (a_1, a_2)$ corresponds to a 2-opt move where a_1, a_2 are node indices. Our architecture also contains a *value* network that outputs value estimates $V_\phi(\bar{S})$, with ϕ as learnable parameters. We assume TSP samples drawn from the same distribution and use Policy Gradient to optimize the parameters of the policy and value networks (Section 6).

States A state $\bar{S}$ is composed of a tuple $\bar{S} = (S, S')$, where S and S' are the current and lowest-cost solution seen in the search, respectively. That is, given a search trajectory at time t and solution S, $S_t = S$ and $S'_t = S' = \arg\min_{S_{\bar{t}} \in \{S_0, \dots, S_t\}} L(S_{\bar{t}})$.

Actions We model actions as tuples $A = (a_1, a_2)$ where $a_1, a_2 \in \{1, \dots, n\}$, $a_2 > a_1$ correspond to index positions of solution $S = (s_1, \dots, s_n)$.

Transitions Given $A = (i, j)$ transitioning to the next state defines a deterministic change to solution $\hat{S} = (\dots, s_i, \dots, s_j, \dots)$, resulting in a new solution $S = (\dots, s_{i-1}, s_j, \dots, s_i, s_{j+1} \dots)$ and state $\bar{S} = (S, S')$. That is, selecting i and j in $\hat{S}$ implies breaking edges at positions $(i-1, i)$ and $(j, j+1)$, inserting edges $(i-1, j)$ and $(i, j+1)$ and inverting the order of nodes between i and j.

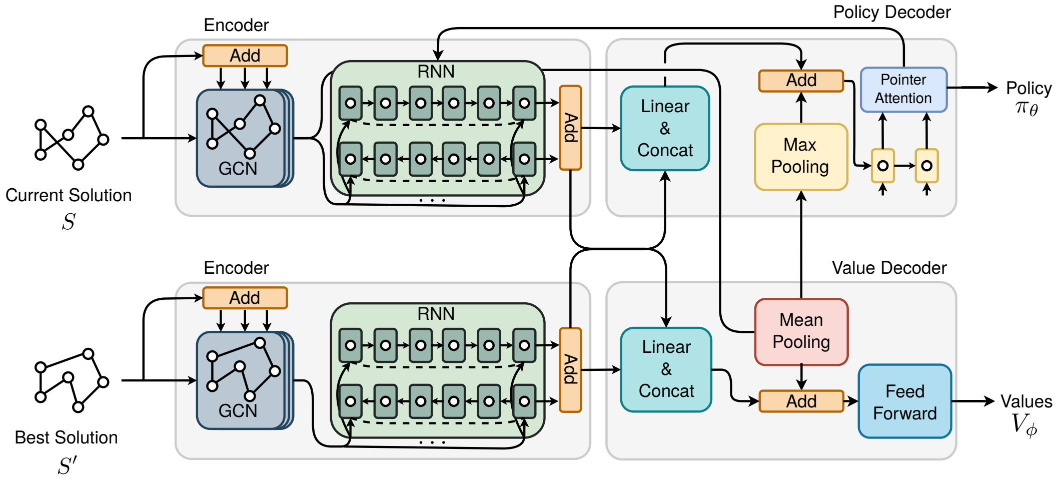

Figure 2: In the architecture, a state $\bar{S} = (S, S')$ is passed to a dual encoder where graph and sequence information are extracted. A policy decoder takes encoded inputs to query node indices and output actions. A value decoder takes encoded inputs and outputs state values.

Rewards Similar to (Wu et al., 2019), we attribute rewards to actions that can improve upon the current best-found solution, i.e., $R_t = L(S'_t) - L(S'_{t+1})$.

Environment Our environment runs for $\mathbb{T}$ steps. For each run, we define episodes of length $T \leq \mathbb{T}$, after which a new episode starts from the last state in the previous episode. This ensures access to poor quality solutions at $t = 0$, and high quality solutions as t grows. In our experiments, treating the environment as continuous and bootstrapping (Mnih et al., 2016) resulted in lower quality policies under the same conditions.

Returns Our objective is to maximize the expected return G_t, which is the cumulative reward starting at time step t and finishing at T at which point no future rewards are available, i.e., $G_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} R_{t'}$ where $\gamma \in (0, 1]$ is a discount factor.

5. Policy Gradient Neural Architecture

Our neural network, based on an encoder-decoder architecture is depicted in Figure 2. Two encoder units map each component of $\bar{S} = (S, S')$ independently. Each unit reads inputs $X = (x_1, \ldots, x_n)$, where x_i are node coordinates of node s_i in S and S' . The encoder then learns representations that embed both graph topology and node ordering. Given these representations, the *policy* decoder samples action indices $a_1, \ldots, a_k$ sequentially, where $k = 2$ for 2-opt. The *value* decoder operates on the same encoder outputs but outputs real-valued estimates of state values. We detail the components of the network in the following sections.

5.1. Encoder

The purpose of our encoder is to obtain a representation for each node in the input graph given its topological structure and its position in a given solution. To accomplish this objective, we incorporate elements from Graph Convolution Networks (GCN) (Kipf and

Welling, 2017) and sequence embedding via Recurrent Neural Networks (RNN) (Hochreiter and Schmidhuber, 1997). Furthermore, we use edge information to build a more informative encoding of the TSP graph.

Embedding Layer We input two dimensional coordinates $x_i \in [0,1]^2$, $\forall i \in 1,\ldots,n$, which are embedded to d-dimensional features as $x_i^0 = W_x x_i + b_x$, where $W_x \in \mathbb{R}^{d\times 2}$, $b_x \in \mathbb{R}^d$. We use as input the Euclidean distances $e_{i,j}$ between coordinates x_i and x_j to add edge information and weigh the node feature matrix. To avoid scaling the inputs to different magnitudes we adopt symmetric normalization (Kipf and Welling, 2017) as $\tilde{e}_{i,j} = \frac{e_{i,j}}{\sqrt{\sum_{j=1}^{n} e_{i,j} \sum_{i=1}^{n} e_{i,j}}}$. Then the normalized edges are used in combination with GCN layers to create richer node representations using its neighboring topology.

Graph Convolutional Layers In the GCN layers, we denote as x_i^{ℓ} the node feature vector at GCN layer ℓ associated with node i. We define the node feature at the subsequent layer combining features from nodes in the neighborhood $\mathcal{N}(i)$ of node i as

$$x_i^{\ell+1} = x_i^{\ell} + \mathrm{ReLU}\Big(\sum\nolimits_{j\in\mathcal{N}(i)} \tilde{e}_{i,j}(W_g^{\ell} x_j^{\ell} + b_g^{\ell})\Big), \tag{1}$$

where $W_g^{\ell} \in \mathbb{R}^{d\times d}$, $b_g^{\ell} \in \mathbb{R}^d$, ReLU is the Rectified Linear Unit and $\mathcal{N}(i)$ corresponds to the remaining $n-1$ nodes of a complete TSP network. At the input to these layers, we have $\ell = 0$ and after $\mathbb{L}$ layers we arrive at representations $z_i = x_i^{\mathbb{L}}$ leveraging node features with the additional edge feature representation.

Sequence Embedding Layers Next, we use node embeddings z_i to learn a sequence representation of the input and encode a tour. Due to symmetry, a tour from nodes $(1,\ldots,n)$ has the same cost as the tour $(n,\ldots,1)$. Therefore, we read the sequence in both orders to explicitly encode the symmetry of a solution. To accomplish this objective, we employ two Long Short-Term Memory (LSTM) as our RNN functions, computed using hidden vectors from the previous node in the tour and the current node embedding resulting in

$$(h_i^{\rightarrow}, c_i^{\rightarrow}) = \mathrm{RNN}(z_i^{\rightarrow}, (h_{i-1}^{\rightarrow}, c_{i-1}^{\rightarrow})), \quad i \in (1,\ldots,n) \tag{2}$$

$$(h_i^{\leftarrow}, c_i^{\leftarrow}) = \mathrm{RNN}(z_i^{\leftarrow}, (h_{i+1}^{\leftarrow}, c_{i+1}^{\leftarrow})), \quad i \in (n,\ldots,1) \tag{3}$$

where in (2) a forward RNN goes over the embedded nodes from left to right, in (3) a backward RNN goes over the nodes from right to left and $h_i, c_i \in \mathbb{R}^d$ are hidden vectors.

Our representation reconnects back to the first node in the tour ensuring we construct a sequential representation of the complete tour, i.e. $(h_0^{\rightarrow}, c_0^{\rightarrow}) = \mathrm{RNN}(z_n, 0)$ and $(h_{n+1}^{\leftarrow}, c_{n+1}^{\leftarrow}) = \mathrm{RNN}(z_1, 0)$. Afterwards, we combine forward and backward representations to form unique node representations in a tour as $o_i = \tanh((W_f h_i^{\rightarrow} + b_f) + (W_b h_i^{\leftarrow} + b_b))$, and a tour representation $h_n = h_n^{\rightarrow} + h_n^{\leftarrow}$, where $h_i, o_i \in \mathbb{R}^d$, $W_f, W_b \in \mathbb{R}^{d\times d}$ and $b_f, b_b \in \mathbb{R}^d$.

Dual Encoding In our formulation, a state $\bar{S} = (S, S')$ is represented as a tuple of the current solution S and the best solution seen so far S'. For that reason, we encode both S and S' using independent encoding layers (Figure 2). Thus, we abuse notation and define a sequential representation of S' after going through encoding layers as $h'_n \in \mathbb{R}^d$.

5.2. Policy Decoder

We aim to learn the parameters of a stochastic policy $\pi_\theta(A|\bar{S})$ that given a state $\bar{S}$, assigns high probabilities to moves that reduce the cost of a tour. Following (Bello and Pham, 2017), our architecture uses the chain rule to factorize the probability of a k-opt move as

$$\pi_\theta(A|\bar{S}) = \prod_{i=1}^{k} p_\theta\left(a_i|a_{<i}\,, \bar{S}\right), \tag{4}$$

and then uses individual softmax functions to represent each term on the RHS of (4), where a_i corresponds to node positions in a tour, $a_{<i}$ represents previously sampled nodes and $k = 2$. At each output step i, we map the tour embedding vectors to the following *query* vector

$$q_i = \tanh\Big((W_q q_{i-1} + b_q) + (W_o o_{i-1} + b_o)\Big), \tag{5}$$

where $W_q, W_o \in \mathbb{R}^{d\times d}$, $b_q, b_o \in \mathbb{R}^{d\times d}$ are learnable parameters and $o_0 \in \mathbb{R}^d$ is a fixed parameter initialized from a uniform distribution $\mathcal{U}(\frac{-1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$. Our initial query vector q_0 receives the tour representation from S and S' as $h_{\bar{s}} = W_s h_n + b_s \| W_{s'} h'_n + b_{s'}$ and a *max pooling* graph representation $z_g = \max(z_1, \ldots, z_n)$ from S to form $q_0 = h_{\bar{s}} + z_g$, where learnable parameters $W_s, W_{s'} \in \mathbb{R}^{\frac{d}{2}\times d}$, $b_s, b_{s'} \in \mathbb{R}^{\frac{d}{2}}$ and $\cdot\|\cdot$ represents the concatenation operation. Our query vectors q_i interact with a set of n vectors to define a pointing distribution over the action space. As soon as the first node is sampled, the query vector updates its inputs with the previously sampled node using its sequential representation to select the subsequent nodes.

Pointing Mechanism We use a pointing mechanism to predict a distribution over node outputs given encoded actions (nodes) and a state representation (query vector). Our pointing mechanism is parameterized by two learned attention matrices $K \in \mathbb{R}^{d\times d}$ and $Q \in \mathbb{R}^{d\times d}$ and vector $v \in \mathbb{R}^d$ as

$$u^i_j = \begin{cases} v^T \tanh(K o_j + Q q_i), & \text{if } j > a_{i-1} \\ -\infty, & \text{otherwise}\,, \end{cases} \tag{6}$$

where $p_\theta\left(a_i \mid a_{<i}, \bar{S}\right) = \text{softmax}(C \tanh(u^i))$ predicts a distribution over n actions, given a query vector q_i with $u^i \in \mathbb{R}^n$. We mask probabilities of nodes prior to the current a_i as we only consider choices of nodes in which $a_i > a_{i-1}$ due to symmetry. This ensures a smaller action space for our model, i.e. $n(n-1)/2$ possible feasible permutations of the input. We clip logits in $[-C, +C]$ (Bello and Pham, 2017), where $C \in \mathbb{R}$ is a parameter to control the entropy of u^i.

5.3. Value Decoder

Similar to the policy decoder, our value decoder works by reading tour representations from S and S' and a graph representation from S. That is, given embeddings Z the value decoder works by reading the outputs z_i for each node in the tour and the sequence hidden vectors h_n, h'_n to estimate the value of a state as

$$V_\phi(\bar{S}) = W_r \ \text{ReLU}\Big(W_z\Big(\frac{1}{n}\sum_{i=1}^{n} z_i + h_v\Big) + b_z\Big) + b_r\,, \tag{7}$$

with $h_v = W_v h_n + b_v \| W_{v'} h'_n + b_{v'}$. Where $W_z \in \mathbb{R}^{d\times d}$, $W_r \in \mathbb{R}^{1\times d}$, $b_z \in \mathbb{R}^d$, $b_r \in \mathbb{R}$ are learned parameters that map the state representation to a real valued output and $W_v, W_{v'} \in \mathbb{R}^{\frac{d}{2}\times d}$, $b_v, b_{v'} \in \mathbb{R}^{\frac{d}{2}}$ map the tours to a combined value representation. We use a *mean pooling* operation in (7) to combine node representations z_i in a single graph representation. This vector is then combined with the tour representation h_v to estimate current state values.

6. Policy Gradient Optimization

In our formulation, we maximize the expected rewards given a state $\bar{S}$ defined as $J(\theta \mid \bar{S}) = \mathbb{E}_{\pi_\theta}[G_t \mid \bar{S}]$. Thus, during training, we define the total objective over a distribution $\mathcal{S}$ of uniformly distributed TSP graphs (solutions) in $[0,1]^2$ as $J(\theta) = \mathbb{E}_{\bar{S}\sim\mathcal{S}}[J(\theta \mid \bar{S})]$. To optimize our policy we resort to the Policy Gradient learning rule, which provides an unbiased gradient estimate w.r.t. the model's parameters θ. During training, we draw B i.i.d. graphs and approximate the gradient of (6), indexed at $t = 0$ as

$$\nabla_\theta J(\theta) \approx \frac{1}{B}\frac{1}{T}\Big[\sum_{b=1}^{B}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^b \mid \bar{S}_t^b)(G_t^b - V_\phi(\bar{S}_t^b))\Big] \tag{8}$$

where the advantage function is defined as $\mathcal{A}_t^b = G_t^b - V_\phi(\bar{S}_t^b)$. To avoid premature convergence to a sub-optimal policy (Mnih et al., 2016), we add an entropy bonus

$$H(\theta) = \frac{1}{B}\sum_{b=1}^{B}\sum_{t=0}^{T-1} H(\pi_\theta(\cdot \mid \bar{S}_t^b))\,, \tag{9}$$

with $H(\pi_\theta(\cdot \mid \bar{S}_t^b)) = -\mathbb{E}_{\pi_\theta}[\log \pi_\theta(\cdot \mid \bar{S}_t^b)]$, and similarly to (8) we normalize values in (9) dividing by k. Moreover, we increase the length of an episode after a number of epochs, i.e. at epoch e, T is replaced by T_e. The value network is trained on a mean squared error objective between its predictions and Monte Carlo estimates of the returns, formulated as an additional objective

$$\mathcal{L}(\phi) = \frac{1}{B}\frac{1}{T}\Big[\sum_{b=1}^{B}\sum_{t=0}^{T-1} \Big\| G_t^b - V_\phi(\bar{S}_t^b))\Big\|_2^2\Big]\,. \tag{10}$$

Afterwards, we combine the previous objectives and perform gradient updates via Adaptive Moment Estimation (ADAM) (Kingma and Ba, 2015), with β_H, β_V representing weights of (9) and (10), respectively. Our model is close to REINFORCE (Williams, 1992) but with periodic episode length updates (truncation), and to Advantage Actor-Critic (A2C) (Mnih et al., 2016) bootstrapping only from terminal states. In our case, this is beneficial as at the start the agent learns how to behave over small episodes for easier credit assignment, later tweaking its policy over larger horizons. The complete algorithm is depicted in Algorithm 1.

7. Experiments and Results

We conduct extensive experiments to investigate the performance of our proposed method. We consider three benchmark tasks, Euclidean TSP with 20, 50 and 100 nodes, TSP20, TSP50 and TSP100 respectively. For all tasks, node coordinates are drawn uniformly at

Algorithm 1 Policy Gradient Training

```
Input: Policy network $\pi_\theta$, critic network $V_\phi$, number of epochs $E$, number of batches $\mathbf{N}_B$,
       batch size $B$, step limit $\mathbb{T}$, length of episodes $T_e$, learning rate $\lambda$
Initialize policy and critic parameters $\theta$ and $\phi$
for $e = 1, \ldots, E$ do
    $T \leftarrow T_e$
    for n $= 1, \ldots, \mathbf{N}_B$ do
        $t \leftarrow 0$
        Initialize random $\bar{S}_0^b, \ \forall b \in \{1, \ldots, B\}$
        while $t < \mathbb{T}$ do
            $t' \leftarrow t$
            while $t - t' < T$ do
                $A_t^b \sim \ \pi_\theta(.|\bar{S}_t^b), \ \forall b \in \{1, \ldots, B\}$
                Take actions $A_t^b$, observe $\bar{S}_{t+1}^b, R_t^b, \ \forall b \in \{1, \ldots, B\}$
                $\bar{S}_t^b \leftarrow \bar{S}_{t+1}^b, \ \forall b \in \{1, \ldots, B\}$
                $t \leftarrow t + 1$
            for $i \in \{t', \ldots, t - 1\}$ do
                $G_i^b \leftarrow \sum_{\tilde{t}=i}^{t'+T-1} \gamma^{\tilde{t}-t'} R_{\tilde{t}}^b \ , \ \forall b \in \{1, \ldots, B\}$
            $g_\theta \leftarrow \frac{1}{Bk}\left[\frac{1}{T}\sum_{b=1}^{B}\sum_{i=t'}^{t-1} \nabla_\theta \log \pi_\theta(A_i^b \mid \bar{S}_i^b)\mathcal{A}_i^b + \beta_H \nabla_\theta H(\pi_\theta(\cdot \mid \bar{S}_i^b))\right]$
            $g_\phi \leftarrow \frac{1}{BT}\left[\beta_V \sum_{b=1}^{B}\sum_{i=t'}^{t-1} \nabla_\phi \left\|G_t^b - V_\phi(\bar{S}_i^b))\right\|_2^2\right]$
            $\theta, \phi \leftarrow \mathrm{ADAM}(\lambda, -g_\theta, g_\phi)$
```

random in the unit square $[0, 1]^2$ during training. For validation, a fixed set of TSP instances with their respective optimal solutions is used for hyperparameter optimization. For a fair comparison, we use the *same* test dataset as reported in Kool et al. (2019); Joshi et al. (2019) containing 10,000 instances for each TSP size. Thus, previous results reported in Kool et al. (2019) are comparable to ours in terms of solution quality (optimality gap). Results from Wu et al. (2019) are not measured in the same data but use the same data generation process. Since at the time of submission no implementation is available, we report the optimality gaps reported in the original paper. Moreover, we report running times reported in Kool et al. (2019); Joshi et al. (2019); Wu et al. (2019). Since time can vary due to implementations and hardware, we rerun the method of Kool et al. (2019) in our hardware. Due to provided supervised samples, the method of Joshi et al. (2019) is not ideal for combinatorial problems. Thus, we compare our results in more detail to Kool et al. (2019) (running time and solution quality) and Wu et al. (2019) (solution quality and sample efficiency).

7.1. Experimental Settings

All our experiments use a similar set of hyperparameters. We use a batch size $B = 512$ for TSP20 and TSP50 and $B = 256$ for TSP100 due to GPU memory. For this reason, we

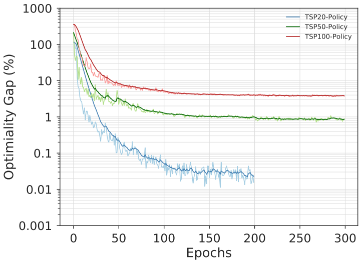

Figure 3: Optimality gaps on 256 validation instances for 200 steps over training epochs.

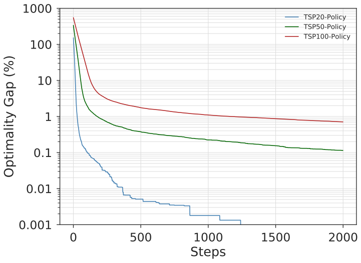

Figure 4: Optimality gaps of best found tours on 512 testing instances over 2,000 sampling steps.

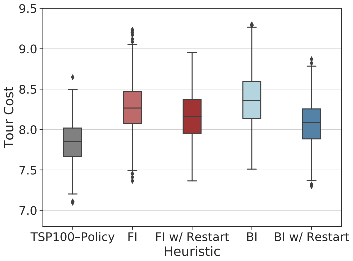

Figure 5: Tour costs of learned, FI and BI heuristics with restarts on TSP100 instances after 1,000 steps.

generate 10 random mini-batches for TSP20 and TSP50 and 20 mini-batches for TSP100 in each epoch. TSP20 trains for 200 epochs as convergence is faster for smaller problems, whereas TSP50 and TSP100 train for 300 epochs. We use the same $\gamma = 0.99$, ℓ_2 penalty of 1×10^{-5} and learning rate $\lambda = 0.001$, λ decaying by 0.98 at each epoch. Loss weights are $\beta_V = 0.5$, $\beta_H = 0.0045$ for TSP20 and TSP50, $\beta_H = 0.0018$ for TSP100. β_H decays by 0.9 after every epoch for stable convergence. In all tasks, $d = 128$, $\mathbb{L} = 3$ and we employ one RNN block. The update in episode lengths are $T_1 = 8, T_{100} = 10, T_{150} = 20$ for TSP 20; $T_1 = 8, T_{100} = 10, T_{200} = 20$ for TSP50; and $T_1 = 4, T_{100} = 8, T_{200} = 10$ for TSP100. $C = 10$ is used during training and testing. v is initialized as $\mathcal{U}(\frac{-1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$ and remaining parameters are initialized according to PyTorch's default parameters.

We train on a RTX 2080Ti GPU, generating random feasible initial solutions on the fly at each epoch. Each epoch takes an average time of 2m 01s, 3m 05s and 7m 16s for TSP20, TSP50 and TSP100, respectively. We clip rewards to 1 to favor non-greedy actions and stabilize learning. Due to GPU memory, we employ mixed precision training (Jia et al., 2018) for TSP50 and TSP100. For comparison with Wu et al. (2019), we train for a maximum step limit of 200. Note that our method is more sample efficient than the proposed in Wu et al. (2019), using 50% and 75% of the total samples for TSP20 and TSP50/100 during training. During testing, we run our policy for 500, 1,000 and 2,000 steps to compare to previous works. Our implementation is available online [1].

7.2. Experimental Results and Analysis

We learn policies for TSP20, TSP50 and TSP100, and depict the optimality gap and its exponential moving average in the log scale in Figure 3. In the figure, the optimality gap is averaged over 256 validation instances and 200 steps (same as training). The results show that we can learn effective policies that decrease the optimality gap over the training epochs. We also point out that increasing the episode length improved validation performance as we consider longer planning horizons in (8). Moreover, it is interesting to note that the optimality gap grows with the instance size as solving larger TSP instances is harder. Additionally, we report the gaps of the best performing policies in Figure 4. In the figure, we show the optimality gap of the best solution for 512 test instances over 2,000 steps. Here, results show

[1] https://github.com/paulorocosta/learning-2opt-drl

that we can quickly reduce the optimality gap at the beginning and later steps attempt to fine-tune the best tour. In the experiments, we find the optimal solution for TSP20 instances and stay within optimality gaps of 0.1% for TSP50 and 0.7% for TSP100. Overall, our policies can be seen as a solver requiring only random initial solutions and sampling to achieve near-optimal solutions.

To showcase that, we compare the learned policies with classical 2-opt *First Improvement* (FI) and *Best Improvement* (BI) heuristics, which select the first and best cost-reducing 2-opt operation, respectively. Since local search methods can get stuck in local optima, we include a version of the heuristics using *restarts*. That is, we restart the search at a random solution as soon as we reach a local optimum. We run all heuristics and learned policies on 512 TSP100 instances for a maximum of 1,000 steps starting from the same solutions. The boxplots in Figure 5 depict the results. We observe that our policy (TSP100-Policy) outperforms classical 2-opt heuristics finding tours with lower median and less dispersion. These results support our initial hypothesis that considering future rewards in the choice of 2-opt moves leads to better solutions. Moreover, our method avoids the worst case $O(n^2)$ complexity of selecting the next solution of FI and BI.

Comparison to Classical Heuristics, Exact and Learning Methods We report results on the same 10,000 instances for each TSP size as in Kool et al. (2019) and rerun the optimal results obtained by Concorde to derive optimality gaps. We compare against Nearest, Random and Farthest Insertion constructions heuristics. and include the vehicle routing solver of OR-Tools (Perron and Furnon) containing 2-opt and LKH as improvement heuristics (Bello and Pham, 2017). We add to the comparison recent deep learning methods based on construction and improvement heuristics, including supervised (Vinyals et al., 2015; Joshi et al., 2019) and reinforcement (Wu et al., 2019; Kool et al., 2019; Deudon et al., 2018; Khalil et al., 2017; Bello and Pham, 2017) learning methods. We note, however, that supervised learning is not ideal for combinatorial problems due to the lack of optimal labels for large problems. Previous works to Kool et al. (2019) are presented with their reported running times and optimality gaps as in the original paper. For recent works, we present the optimality gaps and running times as reported in (Kool et al., 2019; Joshi et al., 2019; Wu et al., 2019). We report previous results using greedy, sampling and search decoding and refer to the methods by their neural network architecture. We note that the test dataset used in Wu et al. (2019) is not the same but the data generation process and size are identical. This fact allied with the high number of samples decreases the variance of the results. We focus our attention on GAT (Kool et al., 2019) and GAT-T (Wu et al., 2019) (GAT-Transformer) representing the best construction and improvement heuristic, respectively.

Our results, in Table 1, show that with only 500 steps our method outperforms traditional construction heuristics, learning methods with greedy decoding and OR-Tools achieving 0.01%, 0.36% and 1.84% optimality gap for TSP20, TSP50 and TSP100, respectively. Moreover, we outperform GAT-T requiring half the number of steps (500 vs 1,000). We note that with 500 steps, our method also outperforms all previous reinforcement learning methods using sampling or search, including GAT (Deudon et al., 2018) applying 2-opt local search on top of generated tours. Our method only falls short of the supervised learning method GCN (Joshi et al., 2019), using beam search and shortest tour heuristic. However, GCN (Joshi et al., 2019), similar to samples in GAT (Kool et al., 2019), uses a beam width of

1,280, i.e. it samples more solutions. Increasing the number of samples (steps) increases the performance of our method. When sampling 1,000 steps (280 samples short of GCN (Joshi et al., 2019) and GAT (Kool et al., 2019)) we outperform all previous methods that do no employ further local search improvement and perform on par with GAT-T on TSP50, using 5,000 samples (5 times as many samples). For TSP100, sampling 1,000 steps results in a lower optimality gap (1.26%) than all compared methods. Lastly, increasing the sample size to 2,000 results in even lower gaps, 0.00% (TSP20), 0.12% (TSP50) and 0.87% (TSP100).

Table 1: Performance of TSP methods w.r.t. Concorde. *Type*: **SL**: Supervised Learning, **RL**: Reinforcement Learning, **S**: Sampling, **G**: Greedy, **B**: Beam Search, **BS**: **B** and Shortest Tour and **T**: 2-opt Local Search. *Time*: Time to solve 10,000 instances reported in (Kool et al., 2019; Joshi et al., 2019; Wu et al., 2019) and ours.

	Method	Type	TSP20			TSP50			TSP100		
			Cost	Gap	Time	Cost	Gap	Time	Cost	Gap	Time
	Concorde (Applegate et al., 2006)	Solver	3.84	0.00%	(1m)	5.70	0.00%	(2m)	7.76	0.00%	(3m)
Heuristics	OR-Tools (Perron and Furnon)	S	3.85	0.37%		5.80	1.83%		7.99	2.90%	
	Nearest Insertion	G	4.33	12.91%	(1s)	6.78	19.03%	(2s)	9.46	21.82%	(6s)
	Random Insertion	G	4.00	4.36%	(0s)	6.13	7.65%	(1s)	8.52	9.69%	(3s)
	Farthest Insertion	G	3.93	2.36%	(1s)	6.01	5.53%	(2s)	8.35	7.59%	(7s)
Const.+Greedy	PtrNet (Vinyals et al., 2015)	SL	3.88	1.15%		7.66	34.48%			-	
	GCN (Joshi et al., 2019)	SL	3.86	0.60%	(6s)	5.87	3.10%	(55s)	8.41	8.38%	(6m)
	PtrNet (Bello and Pham, 2017)	RL	3.89	1.42%		5.95	4.46%		8.30	6.90%	
	S2V (Khalil et al., 2017)	RL	3.89	1.42%		5.99	5.16%		8.31	7.03%	
	GAT (Deudon et al., 2018)	RL,T	3.85	0.42%	(4m)	5.85	2.77%	(26m)	8.17	5.21%	(3h)
	GAT (Kool et al., 2019)	RL	3.85	0.34%	(0s)	5.80	1.76%	(2s)	8.12	4.53%	(6s)
Const.+Search	GCN (Joshi et al., 2019)	SL,B	3.84	0.10%	(20s)	5.71	0.26%	(2m)	7.92	2.11%	(10m)
	GCN (Joshi et al., 2019)	SL,BS	3.84	0.01%	(12m)	**5.70**	**0.01%**	(18m)	7.87	1.39%	(40m)
	PtrNet (Bello and Pham, 2017)	RL,S		-		5.75	0.95%		8.00	3.03%	
	GAT (Deudon et al., 2018)	RL,S	3.84	0.11%	(5m)	5.77	1.28%	(17m)	8.75	12.70%	(56m)
	GAT (Deudon et al., 2018)	RL,S,T	3.84	0.09%	(6m)	5.75	1.00%	(32m)	8.12	4.64%	(5h)
	GAT {1280} (Kool et al., 2019)	RL,S	3.84	0.08%	(5m)	5.73	0.52%	(24m)	7.94	2.26%	(1h)
Impr.+Sampling	GAT-T {1000} (Wu et al., 2019)	RL	3.84	0.03%	(12m)	5.75	0.83%	(16m)	8.01	3.24%	(25m)
	GAT-T {3000} (Wu et al., 2019)	RL	3.84	0.00%	(39m)	5.72	0.34%	(45m)	7.91	1.85%	(1h)
	GAT-T {5000} (Wu et al., 2019)	RL	3.84	0.00%	(1h)	5.71	0.20%	(1h)	7.87	1.42%	(2h)
	Ours {500}	RL	3.84	0.01%	(5m)	5.72	0.36%	(7m)	7.91	1.84%	(10m)
	Ours {1000}	RL	**3.84**	**0.00%**	(10m)	5.71	0.21%	(13m)	7.86	1.26%	(21m)
	Ours {2000}	RL	**3.84**	**0.00%**	(15m)	5.70	0.12%	(29m)	**7.83**	**0.87%**	(41m)

Testing Learned Policies on Larger Instances Since we are interested in learning general policies that can solve the TSP regardless of its size, we test the performance of our policies when learning on TSP50 instances (TSP50-Policy) and applying on larger TSP100 instances. Results, in Table 2, show that we can extract general enough information to still perform well on 100 nodes. Similar to a TSP100-Policy, our TSP50-Policy can outperform previous reinforcement learning construction approaches and requires fewer samples. With

1,000 samples TSP50-Policy performs similarly to GAT-T (Wu et al., 2019) using 3,000 samples, at 1.86% optimality gap. These results are closer to optimal than previous learning methods without further local search improvement as in GCN (Joshi et al., 2019). When increasing to 2,000 steps, we outperform all compared methods at 1.37% optimality gap.

Table 2: Performance of policies trained on 50 and 100 nodes on TSP100 instances.

	TSP100-Policy		TSP50-Policy	
Steps	Cost	Gap	Cost	Gap
500	7.91	1.84%	7.98	2.78%
1000	7.86	1.26%	7.91	1.86%
2000	7.83	0.87%	7.87	1.37%

Table 3: Ablation studies on 512 TSP50 instances running policies for 1,000 steps.

	Epoch: 10		Epoch: 200	
	Opt. Gap (%)	Cost	Opt. Gap (%)	Cost
Proposed	**3.00 ± 0.08**	**5.87**	**0.22 ± 0.01**	**5.72**
(a) w/o bi-LSTM	203.87 ± 0.61	17.33	134.42 ± 0.56	13.37
(b) w/o GCN	9.74 ± 0.08	6.26	0.30 ± 0.01	5.72
(c) w/o bidirectional	17.94 ± 0.15	6.73	2.20 ± 0.05	5.82
(d) w/o best solution	4.55 ± 0.04	5.96	**0.22 ± 0.02**	**5.72**
(e) shared encoder	5.15 ± 0.06	6.00	0.23 ± 0.01	5.72

Running Times and Sample Efficiency Comparing running times is difficult due to varying hardware and implementations among different approaches. In Table 1, we report the running times to solve 10,000 instances as reported in (Kool et al., 2019; Joshi et al., 2019; Wu et al., 2019) and ours. We focus on learning methods, as classical heuristics and solvers are efficiently implemented using multi-threaded CPUs. We note that our method cannot compete in speed with greedy methods as we start from poor solutions and require sampling to find improved solutions. This is neither surprising nor discouraging, as one can see these methods as a way to generate initial solutions for an improvement heuristic like ours. We note, however, that while sampling 1,000 steps, our method is faster than GAT-T (Wu et al., 2019) even though we use a less powerful GPU (RTX 2080Ti vs Tesla V100). Moreover, our method requires fewer samples to achieve superior performance. The comparison to GAT (Kool et al., 2019) is not so straightforward as they use a GTX 1080Ti and different number of samples. For this reason, we run GAT (Kool et al., 2019) using our hardware and report running times sampling the same number of solutions in Table 4. Our method is slower for TSP20 and TSP50 sampling 2,000 solutions. However, as we reach TSP100, our method can be computed faster and, overall, requires less time to produce shorter tours.

Ablation Study In Table 3, we present an ablation study of the proposed method. We measure the performance at the beginning and towards the end of training, i.e. at epochs 10 and 200, rolling out policies for 1,000 steps for 512 TSP50 instances and 10 trials. We observe that removing the LSTM (a) affects performance the most leading to a large 134.42% gap at epoch 200. Removing the GCN component (b) has a lower influence but also reduces the overall quality of policies, reaching 0.30% optimality gap. We then test the effect of the bidirectional LSTM (c) replacing it by a single LSTM. In this case, gaps are even higher, at 2.20%, suggesting that encoding the symmetry of the tours is important. We also compare to two variants of the proposed model, one that does not take as input the best solution (d) and one that shares the parameters of the encoding units (e). For these cases, we note that the final performance is similar to the proposed method, i.e. 0.22% optimality gap. However, in our experiments, the proposed method can achieve better policies faster, reaching a 3.0% gap at epoch 10, whereas (d) and (e) yield policies at the 4.55% and 5.15% level, respectively.

Table 4: Performance of GAT (Kool et al., 2019) vs our method. Results are compared on the same hardware sampling the same number of solutions.

Method	TSP20		TSP50		TSP100	
	Cost	Time	Cost	Time	Cost	Time
GAT {500}	3.839	**(3m)**	5.727	(10m)	7.955	(27m)
Ours {500}	3.836	(5m)	5.716	**(7m)**	7.907	**(10m)**
GAT {1,000}	3.838	**(4m)**	5.725	(14m)	7.947	(42m)
Ours {1,000}	3.836	(10m)	5.708	**(13m)**	7.861	**(21m)**
GAT {2,000}	3.838	**(5m)**	5.722	**(22m)**	7.939	(1h13m)
Ours {2,000}	3.836	(15m)	5.703	(29m)	7.832	**(41m)**

Table 5: Performance of OR-Tools vs our method on TSPLib. See footnote 2.

Instance	Opt.	Ours {2000}	OR-Tools
eil51	426	**427**	439
berlin52	7,542	7,974	**7,944**
pr76	108,159	111,085	**110,948**
rd100	7,910	**7,944**	8,221
eil101	629	**635**	650
lin105	14,379	16,156	**15,363**
ch130	6,110	**6,175**	6,329
pr144	58,537	61,207	**59,286**
ts225	126,643	**127,731**	127,763
a280	2,579	2,898	**2,742**
Avg. Opt. Gap	0.00%	4.56%	3.79%

Generalization to Real-world TSP instances In Table 5, we study the performance of our method on TSPlib (Reinelt, 1991) instances. In general, these instances come from different node distributions than those seen during training and it is unclear whether our learned policies can be reused for these cases. We compare the results of the policy trained on TSP100 sampling actions for 2,000 steps to results obtained from OR-Tools. We note that for the 10 instances tested, our method outperforms OR-Tools in 5 instances. These results are encouraging as OR-Tools is a very specialized heuristic solver. When we compare optimality gaps (4.56% vs 3.79%)[2] we see that our learned policies are not too far from OR-Tools even though our method never trains on instances with more than 100 nodes. The difference in performance increases for large instances, indicating that fine-tuning or training policies for more nodes and different distributions can potentially reduce this difference. However, similar to results in Table 2, our method still can achieve good results on instances with more than 100 nodes, such as ts225 (0.86% gap).

8. Conclusions and Future Work

In this work, we introduced a novel deep reinforcement learning approach for approximating a 2-opt improvement heuristic for the Euclidean Traveling Salesman Problem (TSP). We proposed a neural architecture with graph and sequence embeddings capable of outperforming state-of-the-art learned construction and improvement heuristics requiring fewer samples. Our learned heuristics also outperform classical 2-opt ones reaching lower optimality gaps.

Expanding the proposed neural architecture to sample k-opt operations is an interesting topic for future work. Moreover, exploring general improvement heuristics that can be applied to a large number of combinatorial problems is another interesting idea for further development. One drawback of our policy gradient method is the large number of samples required to train a good policy. As a future direction, we intend to explore methods that can be more sample efficient and can learn good policies requiring less training time. Lastly, we

[2] We perform a more extensive comparison using 35 TSPlib instances in the Supplementary Materials. On the 35 instances the gaps are 8.61% (ours) and 3.70% (OR-Tools).

point out that future work on learning heuristics can be useful when solving problems where standard solvers are not performant, e.g., a TSP with on-route stochastic travel costs.

References

Bernard Angeniol, Gael De La Croix Vaubois, and Jean-Yves Le Texier. Self-organizing feature maps and the travelling salesman problem. *Neural Networks*, 1(4):289–293, 1988.

David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study.* Princeton university press, 2006.

Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

Irwan Bello and Hieu Pham. Neural combinatorial optimization with reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In *Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 170–181, 2018.

Pierre Hansen and Nenad Mladenović. First vs. best improvement: An empirical study. *Discrete Applied Mathematics*, 154(5):802–817, 2006.

Keld Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2-3):119–163, 2009.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

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

Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

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

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*, pages 6348–6358, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Machine Learning*, 2015.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations, (ICLR)*, 2017.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

Bert FJ La Maire and Valeri M Mladenov. Comparison of neural networks for solving the travelling salesman problem. In *11th Symposium on Neural Network Applications in Electrical Engineering*, pages 21–24. IEEE, 2012.

Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

Michele Lombardi and Michela Milano. Boosting combinatorial problem modeling with machine learning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5472–5478, 2018.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, pages 1928–1937, 2016.

Christos H Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science*, 4(3):237–244, 1977.

Laurent Perron and Vincent Furnon. Or-tools. URL https://developers.google.com/optimization/.

Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS)*, pages 2692–2700, 2015.

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

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving the travelling salesman problem. *arXiv:1912.05784*, 2019.