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

Paulo Roberto de Oliveira da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay

Eindhoven University of Technology, 5612 AZ Eindhoven, Netherlands
{p.r.d.oliveira.da.costa, j.s.rhuggenaath, yqzhang, a.e.akcay}@tue.nl

**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 has been shown to be NP-hard [20] 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 [2] which are guaranteed to find an optimal solution but are often too computationally expensive to be used in practice. The latter are based on handcrafted (meta-)heuristics that exploit properties of the problem to construct approximated solutions requiring less computational time, such as heuristics based on edge swaps like $k$-opt [7]. 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 [4,18]. For the TSP, both supervised learning [23,11] and reinforcement learning [3,25,15,5,12] 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 given a set of input nodes. These methods employed sequence representations [23,3], graph neural networks [12,11] and attention mechanisms [15,5,25] 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 that can search for high-quality solutions remains relevant. That is, we focus on methods in which a given solution is improved sequentially until reaching an (local) optimum. 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 [25] 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 and TSP size, which makes it less favorable to expand to more general $k$-opt [7] moves and to learn general policies independent of the number of nodes.

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 [23] 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 [23,11,15,5,12,3] and improvement [25] heuristics. Compared to [25], our method can be easily adapted to general $k$-opt and requires fewer samples to achieve state-of-the-art-performance. 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 [21] and are close to optimal solutions computed by Concorde [2].

## 2 Related Work

Exact approaches for the TSP, such as linear programming, may require a large amount of computational time to find optimal solutions. For this reason, designing fast heuristics for the TSP is necessary. However, classical heuristics require specialized knowledge and may have sub-optimal handcrafted design rules. Therefore, methods that can automatically learn good heuristics have the potential to be more effective than handcrafted ones.

In machine learning, early works for the TSP have focused on Hopfield networks [9] and deformable template models [1]. However, the performance of these approaches has not been on par with classical heuristics [16]. Recent deep learning methods have achieved high performance learning construction heuristics for the TSP. Pointer Networks (PtrNet) [23] introduced a sequence model coupled with an attention mechanism trained to output TSP tours using solutions generated by Concorde [2]. In [3], 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) [12] model was trained to output the ordering of partial tours using Deep Q-Learning (DQN). Later, graph attention modules [5], showed that a hybrid approach using 2-opt local search on top of tours produced via Policy Gradient improves performance. Graph attention was extended in [15] training via REINFORCE [24] with a greedy rollout baseline, resulting in lower optimality gaps. Recently, the supervised approach was revisited using Graph Convolution Networks (GCN) [11] to learn probabilities of edges occurring on a TSP tour, achieving state-of-the-art results up to 100 nodes whilst also combining with search heuristics.

Important to previous methods are 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 [25], based on the transformer architecture, encoded employed Graph Attention Network (GAT) [22] coupled with 2-opt and node swap operations. The limitations of this approach are related to the fixed output embeddings. These are vectors with fixed dimensions defined by the squared number of nodes. This choice makes the model specific to an instance size and expanding to general $k$-opt [7] moves requires increasing the dimension of the output vector. Moreover, the approach 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 [23] to learn a stochastic policy of the action selection task. Our approach resembles classical 2-opt heuristics [6] and can outperform previously 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

$$L(S) = \|x_{s_n} - x_{s_1}\|_2 + \sum_{i=1}^{n-1} \|x_{s_i} - x_{s_{i+1}}\|_2 \ , \tag{1}$$

where $\|\cdot\|_2$ denotes the $\ell_2$ 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) [7] 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 [17], considers 2-opt (Figure 1) and 3-opt moves as 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, 3-opt sequential operations can be decomposed into one, two or three 2-opt operations [7]. 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 [6].

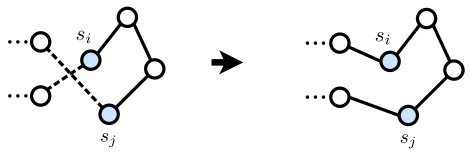

**Fig. 1.** TSP solution before a 2-opt move (left), and the output sequence after a 2-opt move (right). Replaced edges are represented in dashed lines. Note that the sequence $s_i, \ldots, s_j$ is inverted.

To avoid local optima, different metaheuristcs 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, allowing a larger exploration of the search space 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 sequentially improve a TSP solution with one in its neighborhood. Our policy iterates over feasible solutions and the best solution (minimum cost) is returned at the end. The idea behind our method is that by taking future improvements into account we can (potentially) find better solutions than handcrafted heuristics.

## 4 Reinforcement Learning Formulation

Our formulation considers the task of solving the TSP via 2-opt moves as a Markov Decision Process (MDP), detailed below. In our MDP, a given solution (tour) at step $t$ is an observation $S_t$ and the proposed *policy gradient neural architecture* (Section 5) is used as function approximation for the stochastic policy $\pi_\theta(A_t \mid \bar{S}_t)$ where action $A_t$ is selected given a state $\bar{S}_t = (S_t, S'_t)$. Each state $\bar{S}_t$ is represented as a tuple of the current solution $S_t$ and the best solution $S'_t$ (minimum cost) seen up to $t$, where $\theta$ represents the trainable parameters of our *policy* network. Each $A_t$ corresponds to a 2-opt move in which nodes are sampled sequentially. Our architecture also encompasses a *value* network that outputs value estimates $V_\phi(\bar{S}_t)$, with $\phi$ as learnable parameters. We assume that TSP samples are drawn from the same distribution $\mathcal{S}$ and use Policy Gradient to learn the actions of an agent optimizing the parameters of the policy and value networks (Section 6).

**States** $S_t$ represents a solution to a TSP instance at search step $t$, i.e. a tour. A state is composed of the tuple $\bar{S}_t = (S_t, S'_t)$, where $S'_t$ is the lowest cost solution encountered up to step $t$, defined as

$$S'_t = \begin{cases} S_t, & \text{if } L(S_t) < L(S'_{t-1}), \\ S'_{t-1}, & \text{otherwise}\,. \end{cases} \tag{2}$$

where $S'_0 = S_0$ is an intial solution.

**Actions** Actions correspond to 2-opt operations that change the current solution $S_t$ to a solution $S_{t+1}$. We model actions as tuples $A_t = (a_1, a_2)$ where $a_1, a_2 \in \{1, \dots, n\}$, $a_1 \neq a_2$ correspond to index positions of solution $S_t = (s_1, \dots, s_n)$.

**Transitions** Transitioning to a next state $\bar{S}_{t+1}$ is defined from state-action pairs $(\bar{S}_t, A_t)$. That is, given $A_t = (a_1 = i, a_2 = j)$ transitioning to the a next state defines a deterministic change to solution $S_t = (\dots, s_i, \dots, s_j, \dots)$, resulting in a new solution $S_{t+1} = (\dots, s_{i-1}, s_j, \dots, s_i, s_{j+1}, \dots)$ and state $\bar{S}_{t+1} = (S_{t+1}, S'_{t+1})$.

**Rewards** Similar to [25], we attribute rewards to actions that can improve upon the current best-found solution over a number of time steps. Thus, we define our reward function as $R_t = L(S'_t) - min\Big(L(S'_t), L(S_{t+1})\Big)$. Since this reward function automatically results in the agent favoring swaps of very long edges with short edges, we clip rewards to 1 to assign similar rewards in those cases.

**Environment** Our environment runs for a maximum number of steps $\mathbb{T}$. Within each run, we define episodes over a number of steps $T \leq \mathbb{T}$ after which a new episode starts from the last state seen in the previous episode. This ensures that the agent has access to poor quality solutions at $t = 0$, and high

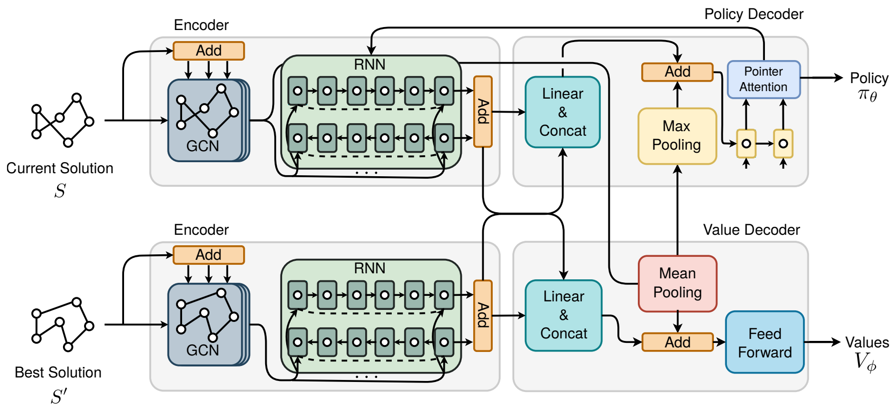

**Fig. 2.** In the proposed architecture, a state $\bar{S} = (S, S')$ is passed to a dual encoder where graph and sequence information are extracted. A policy decoder takes graph and sequential node information to query node indices via pointing attention and output actions. A value decoder operates on the same representations to output state values.

quality solutions as $t$ grows. In our experiments, treating the environment as continuous and bootstrapping [19] 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, represented in Figure 2, follows the general encoder-decoder architecture. Our encoder maps independently each component of a state $\bar{S}_t = (S_t, S'_t)$[1]. That is, each encoding unit reads nodes coordinates $X = (x_1, \ldots, x_n)$, where $x_i$ are the coordinates associated with node $s_i$ in solution $S$. The encoder then transforms the inputs to a set of node representations $Z = (z_1, \ldots, z_n)$ that embed graph topology. Later, we map these representations to a learned sequential embedding $O = (o_1, \ldots, o_n)$ that encodes the positions of each node in a solution. Given node and sequence embeddings from $\bar{S}$, the *policy* decoder is autoregressive and samples output actions $A = (a_1, \ldots, a_k)$ one element at a time, where each $a_i$ corresponds to an index position of the input and $k$ denotes the number of nodes to output, i.e., $k = 2$ for 2-opt. The *value* decoder operates on the same representations but generates real-valued outputs to estimate state values. We detail each component of the architecture in the subsequent sections.

[1] Search step index $t$ is omitted in subsequent definitions to avoid notation clutter. Network parameters are shared for all steps $t$.

## 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) [14] and sequence embedding via Recurrent Neural Networks (RNN) [8] to build its representation. Furthermore, we use edge information to build a more informative encoding of the TSP graph. Incorporating the neighboring edge information accelerates the convergence of the algorithm.

**Embedding Layer** We input two dimensional coordinates $x_i \in [0,1]^2$, $\forall i \in (1,\dots,n)$, which are embedded to $d$-dimensional features as[2]

$$x_i^0 = W_x x_i\,, \tag{3}$$

where $W_x \in \mathbb{R}^{d\times 2}$. 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 [14] as

$$\tilde{e}_{i,j} = \frac{e_{i,j}}{\sqrt{\sum_{j=1}^{n} e_{i,j} \sum_{i=1}^{n} e_{i,j}}}\,. \tag{4}$$

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 $x_i^\ell$ the node feature vector at GCN layer $\ell$ 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

$$x_i^{\ell+1} = x_i^\ell + \mathrm{ReLU}\Big(\sum\nolimits_{j\in\mathcal{N}(i)} \tilde{e}_{i,j} W_g^\ell x_j^\ell\Big), \tag{5}$$

where $W_g^\ell, \in \mathbb{R}^{d\times d}$, ReLU is the rectified linear unit and $\mathcal{N}(i)$ of node $i$ corresponds to the remaining $n-1$ nodes of the complete TSP graph. 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 $(1,\dots,n)$ has the same cost as the tour $(n,\dots,1)$. 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) [8] as our RNN functions, computed using hidden vectors from the previous node in the tour and the current node embedding resulting in

$$(h_i^{\rightarrow}, c_i^{\rightarrow}) = \mathrm{RNN}(z_i^{\rightarrow}, (h_{i-1}^{\rightarrow}, c_{i-1}^{\rightarrow})), \quad i \in (1,\dots,n) \tag{6}$$

[2] In the definitions, bias terms are omitted unless otherwise specified.

$$(h_i^{\leftarrow}, c_i^{\leftarrow}) = \text{RNN}(z_i^{\leftarrow}, (h_{i+1}^{\leftarrow}, c_{i+1}^{\leftarrow})), \quad i \in (n, \ldots, 1) \tag{7}$$

where in (6) a forward RNN goes over the embedded nodes from left to right, in (7) 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. $(h_0^{\rightarrow}, c_0^{\rightarrow}) = \text{RNN}(z_n, 0)$ and $(h_{n+1}^{\leftarrow}, c_{n+1}^{\leftarrow}) = \text{RNN}(z_1, 0)$. Afterwards, we combine forward and backward representations to form unique node representations in a tour as $o_i = \tanh(W_f h_i^{\rightarrow} + W_b h_i^{\leftarrow})$, and a tour representation $h_n = h_n^{\rightarrow} + h_n^{\leftarrow}$, where $h_i, o_i \in \mathbb{R}^d$ and $W_f, W_b \in \mathbb{R}^{d \times d}$.

**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 use the aforementioned encoding layers to 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 $\pi_\theta(A \mid \bar{S})$ that given a state $\bar{S}$, assigns high probabilities to moves that reduce the cost of a tour. Following [3], our architecture uses the chain rule to factorize the probability of a $k$-opt move as

$$\pi_\theta(A \mid \bar{S}) = \prod_{i=1}^{k} p_\theta\left(a_i \mid a_{<i}, \bar{S}\right), \tag{8}$$

and then uses individual softmax functions to represent each term on the RHS of (8), where $a_i$ corresponds to node positions in a tour, $a_{<i}$ corresponds to previously sampled nodes and $k = 2$. During training, we divide (8) by $k$ to normalize loss values. At each output step $i$, we map the tour embedding vectors to the following *query* vector

$$q_i = \tanh(W_q q_{i-1} + W_o o_{i-1}), \tag{9}$$

where $W_q, W_o \in \mathbb{R}^{d \times d}$ 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 $h_{\bar{s}} = W_s h_n \| W_{s'} h'_n$ and a *max pooling* graph representation $z_g = \max(z_1, \ldots, z_n)$ from $S$ to form

$$q_0 = h_{\bar{s}} + z_g, \tag{10}$$

where learnable parameters $W_s, W_{s'} \in \mathbb{R}^{\frac{d}{2} \times d}$, and $\cdot\|\cdot$ represents the concatenation operation. Similar to [23,3,5], 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 $v \in \mathbb{R}^{d}$,

$$u_j^i = \begin{cases} v^T \tanh(Ko_j + Qq_i) & \text{if } j > a_{i-1} \\ -\infty, & \text{otherwise}\,, \end{cases} \tag{11}$$

where

$$p_\theta\left(a_i \mid a_{<i}, \bar{S}\right) = \text{softmax}(C \tanh(u^i)) \tag{12}$$

predicts a distribution over the set of $n$ 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 need to consider choices of nodes in which $a_i > a_{i-1}$ due to symmetry. This ensures a smaller action space for our model as we only consider $n(n-1)/2$ possible moves and feasible permutations of the input. We clip logits in $[-C, +C]$ [3], 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

$$V_\phi(\bar{S}) = W_r \ \text{ReLU}\Big(W_z\Big(\frac{1}{n}\sum_{i=1}^{n} z_i + h_v\Big)\Big)\,, \tag{13}$$

with $h_v = W_v h_n \| W_{v'} h'_n$. Where $W_z \in \mathbb{R}^{h\times h}$, $W_r \in \mathbb{R}^{h\times 1}$ are learned parameters that map the state representation to a real valued output and $W_v, W_{v'} \in \mathbb{R}^{\frac{d}{2}\times d}$ map the tours to a combined value representation. Similar to [25], we use a *mean pooling* operation 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

$$J(\theta \mid \bar{S}) = \mathbb{E}_{\pi_\theta}[G_t \mid \bar{S}]\,. \tag{14}$$

Thus, we define the total training objective over a distribution $\mathcal{S}$ of TSP solutions as

$$J(\theta) = \mathbb{E}_{\mathcal{S}}[J(\theta \mid \bar{S})]. \tag{15}$$

To optimize our policy we resort to the Policy Gradient learning rule, which provides an unbiased gradient estimate w.r.t. the modelâĂŹs parameters $\theta$. During

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

training, we draw $B$ i.i.d. graphs and approximate the gradient in (15), indexed at $t = 0$ as

$$\nabla J(\theta) \approx \frac{1}{B}\frac{1}{T}\Big[\sum_{b=1}^{B}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t^b \mid \bar{S}_t^b)(G_t^b - V_\phi(\bar{S}_t^b))\Big] \tag{16}$$

and define $\mathcal{A}_t^b = G_t^b - V_\phi(\bar{S}_t^b)$. To avoid premature convergence to a sub-optimal policy [19], we add an entropy bonus

$$H(\theta) = \frac{1}{B}\sum_{b=1}^{B}\sum_{t=0}^{T-1} H(\pi_\theta(\cdot \mid \bar{S}_t^b))\,, \tag{17}$$

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 (16) we normalize loss values in (17) 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

$$\mathcal{L}(\phi) = \frac{1}{B}\frac{1}{T}\Big[\sum_{b=1}^{B}\sum_{t=0}^{T-1} \left\|G_t^b - V_\phi(\bar{S}_t^b))\right\|_2^2\Big]\,. \tag{18}$$

Afterwards, we combine the previous objectives via stochastic gradient descent updates via Adaptive Moment Estimation (ADAM) [13], with $\beta_H, \beta_V$ representing weights of (17) and (18), respectively. Our model is close to REINFORCE [24] but with periodic episode length updates, and to Advantage Actor-Critic (A2C) [19] 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 random in the unit square $[0, 1]^2$.

### 7.1 Experimental Settings

All our experiments use a similar set of hyperparameters. We use a batch size of $B = 512$ for TSP20 and TSP50; $B = 256$ for TSP100 due to GPU memory. For this reason, we 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 $\gamma = 0.99$, $\ell_2$ penalty of $1 \times 10^{-5}$ and learning rate $\lambda = 0.001$, $\lambda$ 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. $\beta_H$ 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. Vector $v$ 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 single RTX 2080Ti GPU, generating random 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. Due to GPU memory capacity, we employ mixed precision training [10] for TSP50 and TSP100. Similar to [25], we train for a maximum step limit of 200. During testing, we run our policy for 500, 1,000 and 2,000 steps to showcase that the policy generalizes to larger horizons than the ones trained upon. Our implementation will be made available online.

### 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 graphs and 200 steps (same as

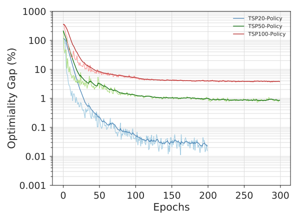

**Fig. 3.** Performance on validation data per epoch.

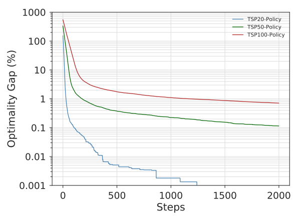

**Fig. 4.** Performance on test data for 2,000 steps.

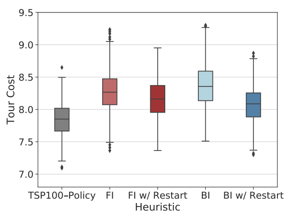

**Fig. 5.** Tour cost of 2-opt heuristics for 1,000 steps.

training). We can observe that instances with a lower number of nodes result in lower optimality gaps as solving instances with a high number of nodes is harder. Moreover, we observe that increasing regularly the size of the episodes leads to improved performance. In Figure 4, we show the best found tour cost for 512 test instances over 2,000 steps using the best performing policies on validation data. Here, we note that we can quickly reduce the optimality gap at the start of the run and later steps attempt to fine-tune the best tour as rewards become harder to obtain. Moreover, results show that the learned policies can be seen as a solver requiring only random initial solutions.

To showcase that, we compare the learned policies with original 2-opt *first improvement* (FI) and *best improvement* (BI) heuristics, which select the first and best cost-reducing 2-opt operation and are the inspiration for our learned policies. Since simple local search methods can easily get stuck in local optima, we also include a version of the heuristics using *restarts*. That is, similar to [25], we restart the local search at a random solution as soon as we reach a local optimum. We run all heuristics and learned polices for a maximum of 1,000 steps over 512 instances starting from the same solutions. The boxplots in Figure 5, show that our policies (TSP100-Policy) have lower median and interquartile range than the other heuristics based on 2-opt, which supports our initial hypothesis of considering future rewards in the choice of 2-opt moves. Moreover, we point out that our method does not scan the neighborhood before picking the next solution, i.e. it avoids the worst case $O(n^2)$ complexity of selecting the next solution.

**Comparison to Classical Heuristics, Exact and Learning Methods** Our comparison results are reported on the same 10,000 instances for each TSP size as reported in [15]. We report optimal results obtained by Concorde [2] and compare against Nearest, Random and Farthest Insertion constructions heuristics based on their optimality gaps reported in [15]. Additionally, we compare to the vehicle routing solver of OR-Tools [21] which includes 2-opt and LKH as improvement heuristics [3]. Furthermore, we compare to recent state-of-the-art deep learning methods based on construction heuristics, including supervised [23,11] and reinforcement [15,5,12,3] learning methods. We note, however, that supervised learning is not ideal for combinatorial optimization problems due to the lack of optimal labels for larger problems. We present the optimality gaps as reported in [15,11,25] using greedy, sampling and search decoding and refer to the methods by their neural network architecture. We also compare to the learned improvement heuristic [25]. We focus our attention on GAT [15] and GAT-T [25] (GAT-Transformer) representing the best performing construction and improvement heuristics, respectively. Our results are summarized in Table 1.

In Table 1, we observe that with only 500 steps, our method outperforms traditional construction heuristics, construction deep learning methods based on 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 [25] 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 [5] applying 2-opt local search on top of generated tours. Our method only falls short of the supervised learning method GCN [11], using beam search and shortest tour heuristic. However, GCN [11], similar to samples in GAT [15], uses a beam width of 1,280. Increasing the number of samples (steps) increases the performance of our method considerably. When augmenting the number of samples to 1,000 (280 samples short of GCN [11] and GAT [15]) we outperform all previous methods that do no employ further local search improvement and perform on par with GAT-T [25] 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 2-opt moves results in even lower gaps, 0.00%, 0.12% and 0.87% for TSP20, TSP50 and TSP100, respectively.

**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. Result, in Table 2, show that we are able to extract general enough information to still perform well on 100 nodes. Similar to the policy trained on 100 nodes, our 50 nodes policy can outperform previous reinforcement learning construction approaches and requires fewer samples. With 1,000 samples our TSP50 policy performs similarly to GAT-T [25] using 3,000 samples, reaching 1.86% optimality gap. These results are closer to optimal than previous learning methods without further local search improvement as in GCN [11]. When increasing to 2,000 steps, we outperform previous deep learning and classical heuristics methods getting as close to 1.37% of the optimal solutions.

**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 [15,11,25] and our running times using the available GPU. We focus on learning methods, as classical heuristics and solvers are efficiently implemented using multi-threaded CPUs and can be run much faster than learning methods. We point out 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

**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 [15,11,25] and ours.

| | Method | Type | TSP20 | | | TSP50 | | | TSP100 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Cost | Gap | Time | Cost | Gap | Time | Cost | Gap | Time |
| | Concorde [2] | Solver | 3.84 | 0.00% | (1m) | 5.70 | 0.00% | (2m) | 7.76 | 0.00% | (3m) |
| Heuristics | OR-Tools [21] | 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 [23] | SL | 3.88 | 1.15% | | 7.66 | 34.48% | | | - | |
| | GCN [11] | SL | 3.86 | 0.60% | (6s) | 5.87 | 3.10% | (55s) | 8.41 | 8.38% | (6m) |
| | PtrNet [3] | RL | 3.89 | 1.42% | | 5.95 | 4.46% | | 8.30 | 6.90% | |
| | S2V [12] | RL | 3.89 | 1.42% | | 5.99 | 5.16% | | 8.31 | 7.03% | |
| | GAT [5] | RL,T | 3.85 | 0.42% | (4m) | 5.85 | 2.77% | (26m) | 8.17 | 5.21% | (3h) |
| | GAT [15] | RL | 3.85 | 0.34% | (0s) | 5.80 | 1.76% | (2s) | 8.12 | 4.53% | (6s) |
| Const.+Search | GCN [11] | SL,B | 3.84 | 0.10% | (20s) | 5.71 | 0.26% | (2m) | 7.92 | 2.11% | (10m) |
| | GCN [11] | SL,BS | 3.84 | 0.01% | (12m) | **5.70** | **0.01**% | (18m) | 7.87 | 1.39% | (40m) |
| | PtrNet [3] | RL,S | | - | | 5.75 | 0.95% | | 8.00 | 3.03% | |
| | GAT [5] | RL,S | 3.84 | 0.11% | (5m) | 5.77 | 1.28% | (17m) | 8.75 | 12.70% | (56m) |
| | GAT [5] | RL,S,T | 3.84 | 0.09% | (6m) | 5.75 | 1.00% | (32m) | 8.12 | 4.64% | (5h) |
| | GAT {1280} [15] | RL,S | 3.84 | 0.08% | (5m) | 5.73 | 0.52% | (24m) | 7.94 | 2.26% | (1h) |
| Impr.+Sampling | GAT-T {1000} [25] | RL | 3.84 | 0.03% | (12m) | 5.75 | 0.83% | (16m) | 8.01 | 3.24% | (25m) |
| | GAT-T {3000} [25] | RL | 3.84 | 0.00% | (39m) | 5.72 | 0.34% | (45m) | 7.91 | 1.85% | (1h) |
| | GAT-T {5000} [25] | 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) |

neither surprising nor discouraging, as one can see greedy construction heuristics 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 [25] 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 [15] is not so straightforward as they employ a GTX 1080Ti over 1,280 samples. For this reason, we run GAT [15] using the hardware at hand and report running times whilst sampling the same number of solutions in Table 3. As it can be observed, our method is slower than the construction model for TSP20 and TSP50 sampling 2,000 solutions. However, as we reach TSP100, our method can be computed faster than GAT [15]. Moreover, if we consider only running times, our method can produce shorter tours in less time.

**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.** Performance of GAT [15] vs our method with the same number of samples.

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

## 8 Conclusions and Future Work

We introduced a novel deep reinforcement learning approach for approximating an improvement heuristic for the 2D Euclidean Traveling Salesman Problem. We proposed graph and sequence embeddings to learn local search policies using 2-opt operators. Our experimental results show that we are able to outperform state-of-the-art deep learning construction and improvement heuristics. As future work, we will explore expanding the model to consider $k$-opt operations dynamically. Moreover, we intend to explore general improvement heuristics that can be applied to a large number of combinatorial problems.

## References

1. Angeniol, B., Vaubois, G.D.L.C., Le Texier, J.Y.: Self-organizing feature maps and the travelling salesman problem. Neural Networks **1**(4), 289–293 (1988)
2. Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The traveling salesman problem: a computational study. Princeton university press (2006)
3. Bello, I., Pham, H.: Neural combinatorial optimization with reinforcement learning. In: Proceedings of the 5th International Conference on Learning Representations (ICLR) (2017)
4. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. arXiv preprint arXiv:1811.06128 (2018)
5. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.M.: 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). pp. 170–181 (2018)

6. Hansen, P., Mladenović, N.: First vs. best improvement: An empirical study. Discrete Applied Mathematics **154**(5), 802–817 (2006)
7. Helsgaun, K.: General k-opt submoves for the lin–kernighan tsp heuristic. Mathematical Programming Computation **1**(2-3), 119–163 (2009)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
9. Hopfield, J.J., Tank, D.W.: Neural computation of decisions in optimization problems. Biological cybernetics **52**(3), 141–152 (1985)
10. Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al.: Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. arXiv preprint arXiv:1807.11205 (2018)
11. Joshi, C.K., Laurent, T., Bresson, X.: An efficient graph convolutional network technique for the travelling salesman problem. arXiv preprint arXiv:1906.01227 (2019)
12. Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS). pp. 6348–6358 (2017)
13. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: International Conference on Machine Learning (2015)
14. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Conference Track Proceedings (2017)
15. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems! In: Proceedings of the 7th International Conference on Learning Representations (ICLR) (2019)
16. La Maire, B.F., Mladenov, V.M.: Comparison of neural networks for solving the travelling salesman problem. In: 11th Symposium on Neural Network Applications in Electrical Engineering. pp. 21–24. IEEE (2012)
17. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Operations research **21**(2), 498–516 (1973)
18. Lombardi, M., Milano, M.: Boosting combinatorial problem modeling with machine learning. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI). pp. 5472–5478 (2018)
19. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: Proceedings of the 33rd International Conference on Machine Learning (ICML). pp. 1928–1937 (2016)
20. Papadimitriou, C.H.: The euclidean travelling salesman problem is np-complete. Theoretical Computer Science **4**(3), 237–244 (1977)
21. Perron, L., Furnon, V.: Or-tools, https://developers.google.com/optimization/
22. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph Attention Networks. In: Proceedings of the 6th International Conference on Learning Representations (ICLR) (2018)
23. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS). pp. 2692–2700 (2015)
24. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning **8**(3-4), 229–256 (1992)
25. Wu, Y., Song, W., Cao, Z., Zhang, J., Lim, A.: Learning improvement heuristics for solving the travelling salesman problem. arXiv preprint arXiv:1912.05784 (2019)