Learning Improvement Heuristics for Solving the Travelling Salesman Problem

Yaoxin Wu,[a] **Wen Song,**[b] **Zhiguang Cao,**[c] **Jie Zhang,**[d] **Andrew Lim**[c]
[a] SCALE@NTU Corp Lab, Nanyang Technological University, Singapore
[b] Institute of Marine Science and Technology, Shandong University, China
[c] Department of Industrial Systems Engineering and Management, National University of Singapore, Singapore
[d] School of Computer Science and Engineering, Nanyang Technological University, Singapore
{[a]wuyaoxin, [d]zhangj}@ntu.edu.sg, {[b]wensong}@email.sdu.edu.cn, {[c]isecaoz, [c]isealim}@nus.edu.sg

Abstract

Recent studies in using deep learning to solve the Travelling Salesman Problem (TSP) focus on construction heuristics, the solution of which may still be far from optimality. To improve solution quality, additional procedures such as sampling or beam search are required. However, they are still based on the same construction policy, which is less effective in refining a solution. In this paper, we propose to directly learn the improvement heuristics for solving TSP based on deep reinforcement learning. We first present a reinforcement learning formulation for the improvement heuristic, where the policy guides selection of the next solution. Then, we propose a deep architecture as the policy network based on self-attention. Extensive experiments show that, improvement policies learned by our approach yield better results than state-of-the-art methods, even from random initial solutions. Moreover, the learned policies are more effective than the traditional hand-crafted ones, and robust to different initial solutions with either high or poor quality.

1 Introduction

The Travelling Salesman Problem (TSP) is a typical combinatorial optimization problem that has extensive applications in the real world. The problem statement is straightforward: given a set of locations, find the salesman a shortest tour that traverses each location exactly once and returns to the original one. Although having been widely studied for decades, achieving satisfactory performance is still challenging due to its NP-hard complexity. Classic approaches for solving TSP can be categorized into exact and heuristic algorithms. The former ones often employ the branch and bound framework (or its variants) and are guaranteed to find optimal solutions. Highly optimized exact solvers such as Concorde (Applegate et al. 2006b) perform reasonably well, but still suffer from (worst-case) exponential computation complexity. Heuristic algorithms, on the other hand, can produce approximate solutions with much less computational time, and are often applied in solving real-world problems. However, traditional heuristics mainly rely on the knowledge and experience of human experts, which may be limited and incapable of delivering high-quality solutions.

Recently, there is a growing trend towards applying machine learning (ML) methods, especially deep learning, to discover better heuristic algorithms for combinatorial optimization problems (Bengio, Lodi, and Prouvost 2018; Lombardi and Milano 2018). The underlying rationale comes from two aspects: 1) a class of problem instances may share similar structures, and differ only in data which follows a distribution; 2) through supervised or reinforcement learning (RL), ML can discover the underlying patterns of a given problem class, which may be used to generate alternative algorithms that are better than the human designed ones. For solving TSP, a popular family of methods (Vinyals, Fortunato, and Jaitly 2015; Bello and Pham 2017; Kool, van Hoof, and Welling 2019) focus on learning heuristics, by imitating the machine translation tasks with sequence-to-sequence (Seq2Seq) models. Based on elaborately designed deep structures such as recurrent neural network (RNN) and attention mechanism, these methods are able to learn heuristics that are expected to produce high-quality solutions.

Though showing promising results, all existing ML methods focus on learning construction heuristics, which create a complete solution incrementally by adding a node to a partial solution at each step. Despite being comparatively fast, this approach may produce solutions with relatively large optimality gap. To narrow the optimality gap, those methods rely on additional procedures, e.g., using sampling to generate multiple solutions and select the best one (Kool, van Hoof, and Welling 2019). However, a contradiction is that solutions in the sampling process are still generated by the same learned construction heuristics, which might be incapable of further improving the solution quality.

In this paper, instead of learning construction heuristics, we propose a method to directly learn the *improvement heuristics* for solving TSP. Starting from an initial solution, an improvement heuristic iteratively replaces the current solution with a new one from its neighbourhood obtained by applying a certain local operator, towards the direction of better solution quality. However, traditional improvement heuristics, such as hill climbing, tabu search and simulated annealing, are all guided by hand crafted policies, which may bring only subtle improvement to the solutions. In this paper, for the first time, we exploit deep reinforcement learning to learn better improvement heuristics for solving TSP, without the need of supervision. To this end, firstly, we present a RL formulation for the improvement heuristics, where the policy guides the decision of selecting the next solution. Then, we propose a deep architecture as

the policy network, based on the self-attention mechanism (Vaswani et al. 2017). In particular, this scheme can incorporate a large variety of commonly used pairwise local operators such as 2-opt and node swap, and can be extended to support more complex operators such as 3-opt. Finally, we use actor-critic algorithm to train the policy network. Extensive experiments show that, even by training using random initial solutions with poor-quality, our approach is able to obtain improvement policies that can yield better results than state-of-the-art deep learning based methods. Our policies are thus robust to different initial solutions, either generated randomly or using different (traditional or learned) construction heuristics. Additionally, the learned policies are indeed more effective than the sampling strategy employed in the current construction based methods and traditional hand-crafted policies, for guiding the improvement process towards solutions with higher quality.

2 Related Work

In this section, we give a brief review of early attempts on using classic ML to tackle TSP, as well as the recent advances in deep learning based models.

Classic ML. Several early works targeted at solving TSP using classic ML methods (Smith 1999). For example, the Hopfield network and Elastic Net were applied in (Hopfield and Tank 1985) and (Durbin and Willshaw 1987), respectively. In terms of learning improvement heuristic, Sadeh, Nakakuki, and Thangiah (1997) proposed to speed up the simulated annealing algorithm by dynamically updating solution distributions at checkpoint temperatures in history runs, and defining thresholds with their means and variances at these checkpoints to predict a promising run. However, all the above methods are instance-dependent, meaning that they only work on a single instance and ignore the pattern under instances from the same problem.

Deep Models. More recently, with the powerful representation and learning ability, deep models have been employed to learn effective instance-independent algorithms that can be applied to solve a class of TSP problems. In (Vinyals, Fortunato, and Jaitly 2015), a modified Seq2Seq model named Pointer Network was proposed to solve TSP in a supervised way. On top of it, Bello and Pham (2017) proposed to use RL to train Pointer Network without the need of optimal solutions as training samples. They also introduced the masking scheme to guarantee feasible solutions, and attention glimpse (Vinyals, Bengio, and Kudlur 2016) to strengthen the network capacity. Inspired by the Transformer architecture (Vaswani et al. 2017), Kool, van Hoof, and Welling (2019) replaced the sequential structures in Seq2Seq model with (self-)attention modules in both the encoder and decoder. This method is flexible and achieves state-of-the-art performance on a series of routing problems including TSP. Similarly, Kaempfer and Wolf (2018) adopted permutation invariant pooling in the Transformer architecture to solve multiple TSP. Deudon et al. (2018) also applied attention-based mechanisms for embedding, but their performances rely on additional 2-opt local search. Different from the Seq2Seq paradigm, Khalil et al. (2017) used Deep Q-learning (DQN) to train a node selection heuristic that works within the architecture of a greedy algorithm for solving TSP, where the internal states are represented using a graph embedding method.

To the best of our knowledge, all the existing deep models learn construction heuristics. In contrast, based on the deep architecture we propose, our method learns the improvement policies that can effectively improve the quality of a given initial solution, produced randomly or by either traditional or learned construction heuristics.

3 Preliminaries

In this section, we introduce TSP and the general improvement heuristics for solving it.

TSP. Formally, a TSP instance can be described as a path finding problem on a weighted graph $G = (V, E, \omega)$, where node set $V = \{1, \dots, n\}$ contains n nodes (or locations), and $x(i)$ represents the coordinate of i with $i \in V$; edge set $E = \{(u, v) | u, v \in V\}$ contains links between locations; the weight function $\omega : (u, v) \to \mathbf{R}^+$ is usually based on the Euclidean norm, i.e., $\omega(u, v) = \|x(u) - x(v)\|_2$. A solution $s = (s^1, \dots, s^n)$ to a TSP instance is a tour, i.e. a sequence of nodes, and $x(s^i)$ represents the coordinate of the ith location in s. The objective of TSP is to minimize the cost with the function $f : s \to \mathbf{R}^+$, which is the summation of Euclidean distances between all pairs of consecutive nodes in s, plus the distance between the starting node and ending node, i.e., $f(s) = \sum_{i=1}^{n-1} \left\|x(s^i) - x(s^{i+1})\right\|_2 + \left\|x(s^1) - x(s^n)\right\|_2$.

General Improvement Heuristics. Starting with an initial solution s_0, improvement heuristics iteratively replace the current solution s_t at step t with s_{t+1} picked from the neighborhood $\mathcal{N}(s_t)$, towards the direction of minimizing f. The most important component of improvement heuristics is the local operator, which is used to construct $\mathcal{N}(s_t)$. Specifically, a local operator carries out a specific operation on the current solution, so that it will yield a specific set of neighboring solutions in $\mathcal{N}(s_t)$. For solving TSP, a well-known family of local operator is λ-opt, which replaces λ links in the current tour with another λ links to get a new tour. Larger λ normally results in larger neighborhood, but also longer computational time, therefore typically λ takes the value of 2 or 3. Besides, there are also other types of local operators such as reinsertion, node swap, and genetic operators (Tao and Michalewicz 1998). In this paper, we focus on pairwise operators, which transform a solution s_t to a new one s_{t+1} by performing operation l on a pair of nodes (s_t^i, s_t^j), i.e. $s_{t+1} = l(s_t, (s_t^i, s_t^j))$. Particularly, Figure 1 illustrates node swap and 2-opt, which are two typical pairwise operators for solving TSP. In general, pairwise operators are fundamental and can be extended to more complex ones. For example, 3-opt and 4-opt can be decomposed into multiple 2-opt operations (Helsgaun 2009).

Other than local operators, the diversity of improvement heuristics also stems from three aspects, including 1) the policy used to pick the next solution, 2) the rules of solution replacement or acceptance, and 3) the termination conditions. Different combinations of the three aspects may yield different schemes of improvement heuristics. For example,

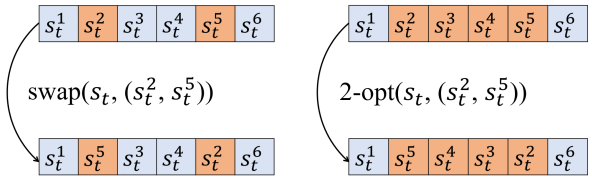


Figure 1: Two typical pairwise local operators for TSP (left: node swap operator exchanges two locations in a tour; right: 2-opt replaces two links by reversing a segment of a tour)

hill climbing with the best-improvement strategy picks from $\mathcal{N}(s_t)$ the solution $\bar{s}_{t+1}$ with the best objective value, replaces s_t with $\bar{s}_{t+1}$ only if $f(\bar{s}_{t+1}) < f(s_t)$, and terminates when no such solution exists. However, this scheme could quickly get stuck in the local optimum. Some other schemes, such as tabu search and simulated annealing, can overcome this limitation by allowing acceptance of worse solutions. However, the solution picking policies are still hand crafted and may lead to poor performance. In this paper, we use deep RL to learn high-quality solution picking policies that work in a simple scheme with the "always accept" rule, i.e. the neighboring solution picked by the policy will always be accepted as the next one. The termination condition is defined by a user-specified maximum step T. The best solution found in the process is immediately returned after termination. Intuitively, this scheme may not suffer from the issue of local optimum. Our approach also has the potential to learn better picking policies for other schemes, which we plan to demonstrate in the future.

4 The Method

We first formulate the process of improvement heuristics as a RL task, and introduce a self-attention based model to parameterize the policy, followed by the training algorithm.

4.1 RL Formulation

In this paper, we assume that the TSP instances are sampled from a distribution $\mathcal{D}$, and use RL to learn a policy of picking the next solution for the improvement heuristic we introduced above. To this end, we formulate the underlying Markov Decision Process (MDP) as follows:

State. The state s_t represents a solution to the TSP instance at time step t, i.e., a sequence of nodes. The initial state is the initial solution s_0 to be improved.

Action. Since we aim at selecting solution within the neighborhood structured by pairwise local operators, the action a_t is represented by a node pair (s_t^i, s_t^j), meaning that this node pair is selected from the current state s_t.

Transition. The next state s_{t+1} is derived deterministically from s_t by a pairwise local operator, i.e., $s_{t+1} = l(s_t, a_t)$. Taking 2-opt as an example, if $s_t = (..., s_t^i, s_t^{i+1}..., s_t^{j-1}, s_t^j...)$ and $a_t = (s_t^i, s_t^j)$, then $s_{t+1} = (..., s_t^j, s_t^{j-1}..., s_t^{i+1}, s_t^i...)$, as illustrated in Figure 1.

Reward. Our ultimate goal is to improve the initial solution as much as possible within the step limit T. To this end, we design the reward function as follows:

$$r_t = r(s_t, a_t, s_{t+1}) = f(s_t^*) - \min\{f(s_t^*), f(s_{t+1})\}, \quad (1)$$

where s_t^* is the best solution found till step t, i.e. the incumbent, which is updated only if s_{t+1} is better, i.e. $f(s_{t+1}) < f(s_t^*)$. Initially, $s_0^* = s_0$. By definition, the reward is positive only when a better solution is found, otherwise $r_t = 0$.

Hence, the cumulative reward (i.e. return) to maximize is expressed as $G_T = \sum_{t=0}^{T-1} \gamma^t r_t$, where γ is the discount factor. When $\gamma = 1$, $G_T = f(s_0) - f(s_T^*)$, which is exactly the improvement over the initial solution s_0.

Policy. Starting from s_0, the stochastic policy π picks an action a_t at each step t, which will lead to s_{t+1}, until reaching the step limit T. This process is characterized by a probability chain rule as follows:

$$P(s_T|s_0) = \prod_{t=0}^{T-1} \pi(a_t|s_t). \quad (2)$$

Remark. Note that in the above MDP, we do not define the terminal states. This is because we intend to apply the trained policy with any user-specified step limit T, in the sense that it can be used as an anytime algorithm. Therefore, we consider the improvement process as a continuing task. To accommodate this, we use the discount factor $\gamma < 1$. Also note that the agent is not forbidden to experience "bad" states with poorer quality than the incumbent. Though these "bad" transitions have the lowest immediate reward (0), in the long term, higher improvement could be gained which follows the principle of RL.

4.2 Self-attention based Policy Network

To learn the stochastic policy π in Equation (2), we parameterize it as a network π_θ. As visualized in Figure 2, the network comprises two parts, which learn node embedding and node pair selection, respectively. Regarding the former, we design a specialized architecture with the sinusoidal positional encoding and self-attention to elegantly embed the nodes. Regarding the latter, we integrate the node embedding and graph embedding, and rely on the compatibility computation for self-attention to produce a probability matrix. Each element in the matrix refers to the probability that the corresponding node pair should be selected for local operation. And we execute the selection by sampling, rather than greedily.

Node Embedding. Given the current state[1] $s = (s^1, \ldots, s^n)$, the node coordinates $x(s^i), i \in \{1, \ldots, n\}$, are first projected to node embeddings by shared linear transformation, with output dimension $d_m = 128$. However, different from the works in (Vinyals, Fortunato, and Jaitly 2015; Kool, van Hoof, and Welling 2019) that only embed the node coordinates, the position of each node in the sequence is also informative to our model. Therefore, we add a sinusoidal positional encoding $pe(i, \cdot)$, to incorporate the position information, which is defined as follows:

$$pe(i, d) = \begin{cases} \sin(i/10000^{\frac{\lfloor d/2 \rfloor}{d_m}}), & \text{if } d \bmod 2 = 0, \\ \cos(i/10000^{\frac{\lfloor d/2 \rfloor}{d_m}}), & \text{if } d \bmod 2 = 1, \end{cases} \quad (3)$$

[1]Step index t is omitted here for better readability, since the network and parameters are shared for sequences at each step.

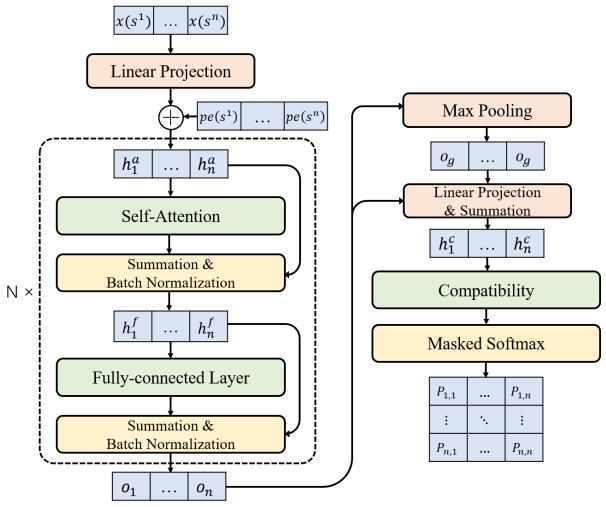


Figure 2: Self-attention based policy network (left: node embedding computation; right: node pair selection)

where i is the position index of node s^i in the sequence; d is the dimension index of the encoding; $\lfloor \cdot \rfloor$ and *mod* refer to *floor* and *modulo* functions, respectively. Then, the position-aware node embeddings are fed to the self-attention layer and fully connected layer subsequently, and processed by them for N times with separate parameters. Here we set N as 3 for a trade-off between network capacity and computational efficiency. Meanwhile, we denote the inputs (i.e., node embeddings) to the two layers as h_i^a and h_i^f, and outputs of them as $\tilde{h}_i^a$ and $\tilde{h}_i^f$, respectively, dimensions of which are all 128. Afterwards, residual connection and batch normalization, i.e., BatchNorm$(h_i^a + \tilde{h}_i^a)$ and BatchNorm$(h_i^f + \tilde{h}_i^f)$, are applied to each of the two layers for better training speed and stability.

Among the above operations, self-attention transforms node embeddings through message passing and aggregation among nodes (Veličković et al. 2018). We adopt the single-head self-attention here, since the multi-head version does not lead to any significant improvement according to our experiments. Particularly, given the input matrix $H^a \in \mathbf{R}^{d_m \times n}$ with the columns being node embeddings, i.e., $H^a = [h_1^a, \ldots, h_n^a]$, the self-attention is defined as:

$$\tilde{H}^a = V_a \cdot \text{softmax}_c(\frac{K_a^T Q_a}{\sqrt{d_k}}), \tag{4}$$

where $Q_a = W^q H^a$, $K_a = W^k H^a$, and $V_a = W^v H^a$ are the *query*, *key*, and *value* matrix of H_a, respectively. $W^q \in \mathbf{R}^{d_q \times d_m}$, $W^k \in \mathbf{R}^{d_k \times d_m}$ and $W^v \in \mathbf{R}^{d_v \times d_m}$ are all trainable parameters. Normally, $d_q = d_k$, and d_v determines the output dimension. In our model, $d_q = d_k = d_v = 128$. The $\text{softmax}_c(\cdot)$ is a column-wise softmax function so that the output of Equation (4) is the transformed node embeddings, i.e., $\tilde{H}^a = [\tilde{h}_1^a, \ldots, \tilde{h}_n^a]$.

Different from self-attention, the fully connected layer transforms each node embedding independently, with shared parameters. In our model, it is concretized as a 512-dimensional hidden sublayer with ReLU activation function.

Node Pair Selection. In this part, we further process the transformed node embeddings with three operations, i.e., feature fusion, compatibility computation and masked softmax. In doing so, we expect to produce a matrix that includes the probabilities to pick the corresponding node pairs.

Given the transformed node embeddings o_i generated by the preceding part, we further aggregate them by max-pooling, i.e., $o_g = max(\{o_1 \ldots, o_n\})$, where o_g becomes the *graph embedding*. Next, the node and graph embeddings are processed by linear transformation, respectively, and fused together afterwards. Then, the final node embeddings are represented as $h_i^c = lp_1(o_i) + lp_2(o_g)$, where both linear layers lp_1 and lp_2 have 128-dimensional inputs and outputs. Thus, we have incorporated the global graph information of the TSP instance into each node, which helps to make better decision.

Compatibility is a function of query and key vectors, and used in (self-)attention to compute the weights for value vectors. Considering its capability of representing the relations between words in sentences (Vaswani et al. 2017), we adopt a multiplicative compatibility to predict the node pair for selection in a sequence. Given node embeddings $H^c = [h_1^c, \ldots, h_n^c]$, we compute the dot product of their query and key matrices, i.e., $M = K_c^T Q_c$, where the compatibility matrix $M \in \mathbf{R}^{n \times n}$ reflects the scores to pick the corresponding node pairs.

With certain preprocessing, softmax is applied to the compatibility matrix as follows:

$$\tilde{M}_{ij} = \begin{cases} C \cdot \tanh(M_{ij}), & \text{if } i \neq j, \\ -\infty, & \text{if } i = j, \end{cases} \tag{5}$$

$$P = \text{softmax}(\tilde{M}), \tag{6}$$

where we first limit the values in the compatibility matrix within $[-C, C]$ by a $\tanh$ function. Then following (Bello and Pham 2017), we set $C = 10$ to control the entropy of $\tilde{M}_{ij}$. We also mask the diagonal elements, since local operation with two same nodes does not make sense. Consequently, the element p_{ij} in P represents the probability that node pair (s^i, s^j) should be selected for local operation. However, rather than doing it greedily, we select the node-pair by sampling based on the probability matrix P. In summary, the whole network π_θ models the stochastic policy π in Equation (2), and θ refers to the trainable parameters in the above stacked layers.

4.3 Training Algorithm

We adopt the actor-critic algorithm with Adam optimizer to train π_θ. More specifically, it is a kind of policy gradient method, based on the REINFORCE algorithm (Williams 1992) with extra trainable critic network updated by bootstrapping. The actor refers to the policy network described above. The critic v_ϕ estimates the cumulative reward at each state, and the structure is similar to that of actor, except that: 1) mean-pooling is used to obtain graph embedding; 2) the fused node and graph embeddings are processed by a fully connected layer which is similar to the one used in policy

Algorithm 1: $\mathbf{n}$-step actor-critic (continuing task)

Input: actor network π_θ with trainable parameters θ; critic network v_ϕ with trainable parameters ϕ; number of epochs E, batches B; step limit T.

1 **for** e = *1, 2,* $\cdots$, E **do**
2 generate M problem instances randomly;
3 **for** b = *1, 2,* $\cdots$, B **do**
4 retrieve batch M_b; $t \leftarrow 0$;
5 **while** $t < T$ **do**
6 reset gradients: $d\theta \leftarrow 0$; $d\phi \leftarrow 0$;
7 $t_s = t$; get state s_t;
8 **while** $t - t_s < \mathbf{n}$ *and* $t \neq T$ **do**
9 sample a_t based on $\pi_\theta(a_t|s_t)$;
10 receive reward r_t and next state s_{t+1};
11 $t \leftarrow t + 1$;
12 $R = v_\phi(s_t)$;
13 **for** $i \in \{t-1, \ldots, t_s\}$ **do**
14 $R \leftarrow r_i + \gamma R$; $\delta \leftarrow R - v_\phi(s_i)$;
15 $d\theta \leftarrow d\theta + \sum_{M_b} \delta \nabla log \pi_\theta(a_i|s_i)$;
16 $d\phi \leftarrow d\phi + \sum_{M_b} \delta \nabla v_\phi(s_i)$;
17 update θ and ϕ by $\frac{d\theta}{|M_b|(t-t_s)}$ and $\frac{d\phi}{|M_b|(t-t_s)}$

network, but with single-value output. In our algorithm, $\mathbf{n}$-step return is used for efficient reward propagation and bias-variance trade-off (Mnih et al. 2016). Additionally, since we do not define terminal state, it is necessary to bootstrap the value from the time limit state, such that the policy for the continuing task can be learned correctly (Pardo et al. 2018). The complete algorithm is given in Algorithm 1, in which lines 5 to 16 process a batch of instances in parallel and accumulate their gradients to update the networks.

5 Experimental Results

We conduct extensive experiments to evaluate our approach. The TSP instances used here are Euclidean TSP with 20, 50 and 100 nodes, and we call these instances TSP20, TSP50, and TSP100, respectively. Generally, the larger size of the instance, the more difficult to solve it. Following the settings in the state-of-the-art method (Kool, van Hoof, and Welling 2019), each node is represented by a pair of coordinates, which are randomly generated in the unit square, i.e., $[0, 1] \times [0, 1]$, with a uniform distribution.

We use fixed hyperparameters to train the agent for TSP of the three sizes. Specifically, we train for 200 epochs. In each epoch, 10,240 random instances are generated on the fly and split into 10 batches. We use random initial solutions for training, the quality of which is normally poor. For validation and testing, the initial solutions are generated using a variety of methods, which will be detailed in the corresponding subsections. As mentioned, we model the improvement heuristic as a continuing task. However, this does not mean we need to train the agent for a very large step limit T. Empirically, we set T=200 for training, since the rewards in the early stages of improvement are more dense. However, we will show later that the trained policies generalize well to unseen states and much larger T during validation and testing. We set the discount factor γ as 0.99, and $\mathbf{n}$ in $\mathbf{n}$-step return as 4. For Adam optimizer, the initial learning rate lr is set as 10^{-4}, decaying 0.99 per epoch for convergence, and other parameters follow the default settings in PyTorch (Paszke et al. 2017). With single GPU Tesla V100, each training epoch takes an average time of 8:20m (i.e., 8 minutes and 20 seconds), 16:30m and 31:00m for TSP20, TSP50 and TSP100, respectively. In the following subsections, we evaluate the performance of our method from different perspectives. Our code in PyTorch will be available soon.

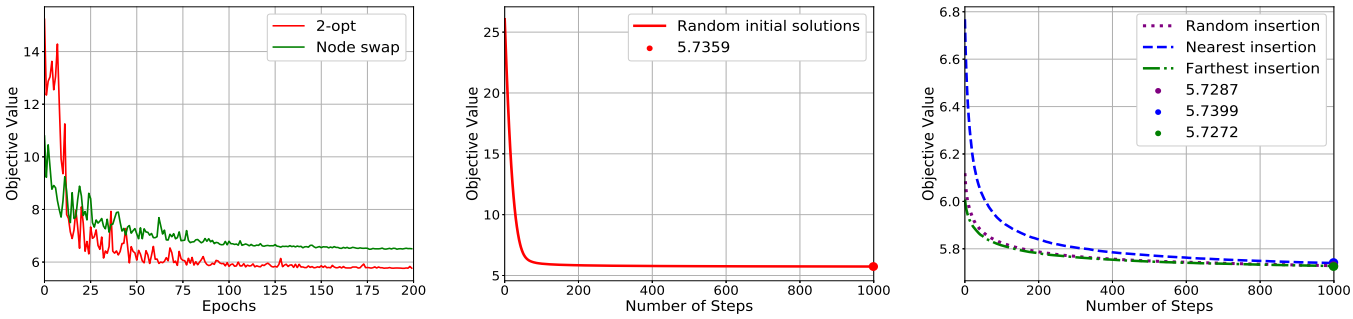
5.1 Analysis of Our Method

We first evaluate the performance of learning with different local operators. Here we choose two commonly used ones for TSP, i.e., 2-opt and node swap, and incorporate them into the improvement heuristic proposed in Section 4. We learn two policies with the two operators on TSP50, and the training curves are depicted in Figure 3(a), where the objective values are averaged over a validation set of 2,000 instances with T=1,000. Note that all the initial solutions to these validating instances are randomly generated. As shown, for both operators, our method can effectively learn good policies, since the two objective values are obviously decreasing, until converged. Comparing the two learning curves, we can clearly observe that, despite the relatively poor performance at the beginning, the policy for 2-opt converges to a significantly lower value than that for node swap. The inferiority of node swap might come from the fact that, it needs to take more steps to find an improvement over the incumbent. Hence, we adopt 2-opt as the local operator in the learning based improvement heuristic for the following experiments.

Though we train the policy only with random initial solutions of poor quality, it generalizes well to other initial solutions of various qualities. To demonstrate this, we apply the learned policy on the same 2,000 instances of TSP50 but with two types of initial solutions, i.e., generated randomly or by conventional construction heuristics. Regarding the latter, we implement three widely used variants of insertion heuristics, i.e., random, nearest, and farthest insertion. Then we plot the average objective values of the *incumbents* against the time steps for all methods in Figure 3(b) and 3(c), respectively. In particular, Figure 3(b) displays the result of improvement on random initial solutions, where the objective value drops quickly in about 100 steps; after that, the change is slight but still improving. Figure 3(c) shows the results with the conventional construction heuristics, and we can observe that the three heuristics generate different initial solutions (at step 0), whose qualities are much higher than the random ones. The learned policy constantly reduces the objective values to comparable qualities with respect to the experiments on random initial solutions. These observations indicate that, the policy can effectively generalize to unseen initial solutions with various qualities. Moreover, it does not rely on high-quality initial solutions, i.e., it can serve as an independent solver requiring only random initial solutions.

5.2 Comparison with State-of-the-art Methods

We compare the learned improvement policies with a variety of methods for solving TSP. The baseline methods include:



(a) Performance with different local operators (b) Improvement on random initial solutions (c) Improvement on insertion heuristics

Figure 3: Analysis of Our Method

Table 1: Comparison with Other Approaches

Methods	TSP20 Obj.	TSP20 Gap	TSP50 Obj.	TSP50 Gap	TSP100 Obj.	TSP100 Gap
Concorde	3.831	0.00%	5.692	0.00%	7.762	0.00%
OR-Tools	3.867	0.94%	5.854	2.87%	8.061	3.86%
Random Insertion	4.000	4.40%	6.128	7.66%	8.508	9.62%
Nearest Insertion	4.332	13.17%	6.777	19.11%	9.456	21.86%
Farthest Insertion	3.921	2.32%	6.007	5.52%	8.350	7.58%
AM greedy	3.843	0.30%	5.786	1.65%	8.101	4.37%
AM sampling (1,280)	3.834	0.06%	5.720	0.48%	7.942	2.32%
Ours (1,000)	3.832	0.03%	5.740	0.83%	8.014	3.24%
Ours (3,000)	**3.831**	**0.00%**	5.712	0.34%	7.905	1.85%
Ours (5,000)	**3.831**	**0.00%**	**5.703**	**0.20%**	**7.872**	**1.42%**

[1] Gap is computed based on exact solutions given by Concorde.

1) Concorde (Applegate et al. 2006a), an efficient and specialized exact solver for TSP; 2) OR-Tools (Google. 2016), a mature and widely used solver for routing problems based on metaheuristics; 3) three conventional construction heuristics used in the previous subsection; 4) the state-of-the-art method (Kool, van Hoof, and Welling 2019), which learns construction policies based on an attention model (AM). Specifically, they use two versions of node selection strategies to roll out a learned policy, i.e., *greedy* and *sampling*. The former selects the node with the maximum probability to visit next, while the latter does it by sampling the probability distribution. Additionally, AM sampling will produce $\mathbb{N}$ complete solutions to an instance, and the best one is selected as the final output. Therefore, AM sampling is considered as an improvement to AM greedy, and achieves the state-of-the-art performance. Here, we use the default setting in (Kool, van Hoof, and Welling 2019), i.e. $\mathbb{N}$=1,280. For our method, we run the learning based improvement heuristic with random initial solutions, and record the results for different step limit T during testing. We increase the number of testing instances up to 10,000 for the three sizes of TSP. All the results are summarized in Table 1.

From Table 1, we can see that for T=1,000, our method outperforms OR-Tools, the three construction heuristics, and AM greedy for all the three problem sizes. Besides, the results on TSP20 are also superior to that of AM sampling. We can also observe that, as T increases, our policies produce better results with much smaller optimality gap. Specifically, for T=3,000, our method outperforms AM sampling regarding all problem sizes, and achieves almost the same result as Concorde on TSP20. Though these results are already quite close to the exact solutions, with additional 2,000 steps (i.e., T=5,000), our method still can further reduce optimality gaps from 0.34% to 0.20%, and from 1.85% to 1.42% for TSP50 and TSP100, respectively. Those observations demonstrate that, with step limit T being 200 during training, the learned policies perform fairly well with much larger step limits during testing. This justifies the continuing design of our RL formulation.

5.3 Comparison with the Sampling Strategy

From a practical view, the sampling strategy used by AM and our improvement heuristic are similar, since both of them generate multiple solutions and return the best one. Therefore, one may ask whether the results of AM sampling in Table 1 can be improved with larger $\mathbb{N}$. However, as will be shown later, the improvement brought by increasing $\mathbb{N}$ is not significant, and inferior to our improvement policies given the same number of solutions generated. To validate this, we run both AM sampling and our policies by adopting the results of AM greedy as the initial solutions. We use both methods to solve 10,000 instances of TSP20, TSP50 and TSP100, respectively. For all the instances, we set T=$\mathbb{N}$=5,000, since the two parameters function similarly.

The results are shown in Figure 4, where we plot the average optimality gaps of the incumbents against the number of steps (or sampled solutions). We can observe that though our method is a bit inferior in the early stages, it performs significantly better than the sampling strategy after a certain number of steps especially for harder problems TSP50 and TSP100. Besides, for TSP20 and TSP50, our method almost achieves the optimal solutions. These results validate our observation in Section 5.1 again, that our policies can generalize to unseen initial solutions. On the other hand, the improvement of AM sampling significantly slows down when it samples more than 1000 solutions. We can also observe that the superiority of our policies over AM sampling tends

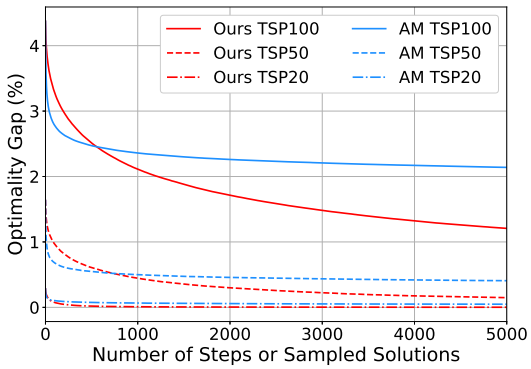


Figure 4: Comparison with AM sampling

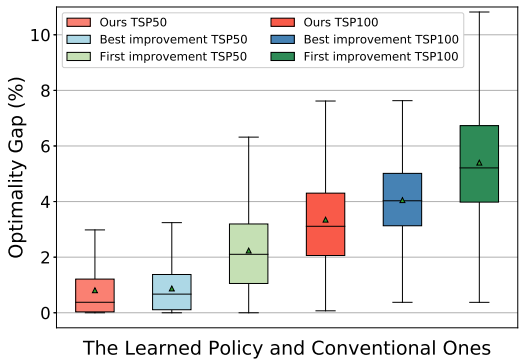


Figure 5: Comparison with Conventional Policies

Table 2: Computation Time Comparison

Methods	Sizes		
	TSP20	TSP50	TSP100
AM sampling (1000)	**9m**	26m	1h9m
Ours (1000)	12m	**16m**	**25m**
AM sampling (3000)	**28m**	1h20m	3h25m
Ours (3000)	39m	**45m**	**1h15m**
AM sampling (5000)	**47m**	2h13h	5h44m
Ours (5000)	1h6m	**1h15m**	**2h6m**

to enlarge with the increase of number of steps and problem hardness (sizes). Therefore, we can conclude that our policies are indeed more effective than the sampling strategy in terms of improving the quality of initial solutions.

We also compare computation time of the two methods. For each problem size, we record the total running time in Table 2, with T (or $\mathbb{N}$) being 1,000, 3,000 and 5,000, respectively. The results reveal that as T (or $\mathbb{N}$) increases or problem size grows, the computation time for both methods becomes longer. However, the computation time of our method is significantly shorter than that of AM sampling on larger problems, i.e., TSP50 and TSP100. This superiority might come from the fact that, to produce a new solution, our policies only need to select a node pair once with the compatibility computation, followed by a local operation with almost no time-consumption, while AM sampling needs to apply multi-head attention to determine a node for n-1 times.

5.4 Comparison with Conventional Policies

The major difference between our method and the conventional improvement heuristics is that the policies of picking solutions from the neighborhood is learned, instead of handcrafted. To show that the learned policies are indeed better, we compare them with two widely used conventional ones, i.e., *first-improvement* and *best-improvement*, which select the first and best cost-reducing solution in the neighborhood, respectively (Hansen and Mladenović 2006). However, direct comparison is not fair since they can easily get stuck in local minimum if no improvement can be found. Therefore, we apply these conventional policies also with the "always accept" rule, and implement a simple but commonly used strategy to escape local minimum, i.e., *restart*, which randomly picks a solution from the solution space. For fair comparison, we run the conventional and our learned policies for 1,000 steps on TSP50 and TSP100, and use 2-opt as the local operator. All methods use the same set of initial solutions, which are generated randomly. The results are visualized using box plots in terms of optimality gaps, as depicted in Figure 5. As can be observed, the instances solved by the learned policies clearly have the smallest mean value, median value and maximum value for both problem sizes, which justify its superiority to the conventional ones. Furthermore, the advantage of the learned policy is more significant on TSP100 than that on TSP50, which reveals that it can offer better guidance than conventional policies when facing much harder problem. The computational time here is not directly comparable, since the conventional policies are implemented on CPU. However, the learning based heuristics could be more efficient, since the policy network directly picks the next solution, without the need of traversing the whole neighborhood as in the conventional ones.

6 Conclusions and Future Work

In this paper, we propose a learning based improvement heuristic method, which adopts deep reinforcement learning and self-attention based model to guide a local operator for searching better solutions. The extensive experimental results show that our method is able to outperform the state-of-the-art methods, and the conventional heuristics as well. Moreover, the proposed method is robust to initial solutions with either high or poor quality. In future, we will, 1) apply our method to the variants of TSP, e.g., Capacitated Vehicle Routing Problem (CVRP) and multiple Travelling Salesman Problem (mTSP); 2) design a smart termination condition rather than pre-defining step limit T; 3) incorporate with more complex search schemes, like those used in tabu search and simulated annealing.

References

[Applegate et al. 2006a] Applegate, D.; Bixby, R.; Chvatal, V.; and Cook, W. 2006a. Concorde tsp solver. *URL http://www.math.uwaterloo.ca/tsp/concorde*.

[Applegate et al. 2006b] Applegate, D. L.; Bixby, R. E.; Chvatal, V.; and Cook, W. J. 2006b. *The traveling salesman problem: a computational study*. Princeton university press.

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

[Bengio, Lodi, and Prouvost 2018] Bengio, Y.; Lodi, A.; and Prouvost, A. 2018. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*.

[Deudon et al. 2018] Deudon, M.; Cournut, P.; Lacoste, A.; Adulyasak, Y.; and Rousseau, L.-M. 2018. 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)*, 170–181.

[Durbin and Willshaw 1987] Durbin, R., and Willshaw, D. 1987. An analogue approach to the travelling salesman problem using an elastic net method. *Nature* 326(6114):689.

[Google. 2016] Google. 2016. Or-tools, google optimization tools. *URL https://developers.google.com/optimization/routing*.

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

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

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

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

[Khalil et al. 2017] Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*, 6348–6358.

[Kool, van Hoof, and Welling 2019] Kool, W.; van Hoof, H.; and Welling, M. 2019. Attention, learn to solve routing problems! In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*.

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

[Mnih et al. 2016] Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 1928–1937.

[Pardo et al. 2018] Pardo, F.; Tavakoli, A.; Levdik, V.; and Kormushev, P. 2018. Time limits in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 4042–4051.

[Paszke et al. 2017] Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in pytorch. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*.

[Sadeh, Nakakuki, and Thangiah 1997] Sadeh, N. M.; Nakakuki, Y.; and Thangiah, S. R. 1997. Learning to recognize (un) promising simulated annealing runs: Efficient search procedures for job shop scheduling and vehicle routing. *Annals of Operations Research* 75:189–208.

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

[Tao and Michalewicz 1998] Tao, G., and Michalewicz, Z. 1998. Inver-over operator for the tsp. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature (PPSN)*, 803–812.

[Vaswani et al. 2017] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*, 5998–6008.

[Veličković et al. 2018] Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

[Vinyals, Bengio, and Kudlur 2016] Vinyals, O.; Bengio, S.; and Kudlur, M. 2016. Order matters: Sequence to sequence for sets. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

[Vinyals, Fortunato, and Jaitly 2015] Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS)*, 2692–2700.

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