Learning Improvement Heuristics for Solving Routing Problems

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, Andrew Lim

arXiv:1912.05784v2 [cs.AI] 10 May 2020

***Abstract*—Recent studies in using deep learning to solve routing problems focus on construction heuristics, the solutions of which are still far from optimality. Improvement heuristics have great potential to narrow this gap by iteratively refining a solution. However, classic improvement heuristics are all guided by hand-crafted rules which may limit their performance. In this paper, we propose a deep reinforcement learning framework to learn the improvement heuristics for routing problems. We design a self-attention based deep architecture as the policy network to guide the selection of next solution. We apply our method to two important routing problems, i.e. travelling salesman problem (TSP) and capacitated vehicle routing problem (CVRP). Experiments show that our method outperforms state-of-the-art deep learning based approaches. The learned policies are more effective than the traditional hand-crafted ones, and can be further enhanced by simple diversifying strategies. Moreover, the policies generalize well to different problem sizes, initial solutions and even real-world dataset.**

I. Introduction

Routing problems, e.g., Travelling Salesman Problem (TSP) and Capacitated Vehicle Routing Problem (CVRP), are a class of combinatorial optimization problems with numerous real-world applications. Although we solve them regularly in daily life, achieving satisfactory results is still challenging, due to their NP-hardness. Classical approaches to routing problems could be categorized into exact methods, approximation methods, and heuristics [1], [2]. Exact methods are often designed based on the branch-and-bound framework, which have the theoretical guarantee of finding the optimal solution, but practically limited to small instances for their exponential complexity in the worst case [3], [4]. Approximation methods can find suboptimal solutions with probable worst-case guarantees in polynomial time, but they may only exist for specific problems and still be of poor approximation ratios [5], [6]. In practice, heuristics are the most commonly applied approaches for solving routing problems. Despite lacking theoretical guarantee on the solution quality, heuristics often can find desirable solutions within reasonable computational time [7], [8], [9]. However, the development of heuristics requires substantial trial-and-error, and the the performance in terms of solution quality is highly dependent on the intuition and experience of human experts [10].

Recently, there is a growing trend towards applying deep learning (DL) to automatically discover heuristic algorithms for solving routing problems. 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), DL models can discover the underlying patterns of a given problem class, which could be used to generate alternative algorithms that are better than the human designed ones [11]. A popular family of methods consider the process of solving routing problems as a sequence generation task, and leverage the sequence-to-sequence (Seq2Seq) models [12], [13]. Based on elaborately designed deep structures such as recurrent neural network (RNN) [14], [15], [16] and attention mechanism [17], [18], [19], these methods are able to learn heuristics that can produce high-quality solutions.

Though showing promising results, as will be reviewed later, most of the existing DL based 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, their results still have a relatively large gap to the highly-optimized traditional solvers in terms of objective values. To narrow this gap, they often rely on additional procedures (e.g. sampling or beam search) to improve solution quality, which have limited capabilities since they rely on the same trained construction policy.

In this paper, rather than learning construction heuristics, we present a framework to directly learn *improvement heuristics*, which improve an initial solution by iteratively performing neighborhood search based on certain local operator, towards the direction of improving solution quality [20], [21], [22]. Traditional improvement heuristics are guided by hand-crafted search policies, which require substantial domain knowledge to design and may bring only limited improvements to the solutions. In contrast, we exploit deep reinforcement learning to *automatically* discover better improvement policies. Specifically, we first present a RL formulation for the improvement heuristics, where the policy guides the selection of next solution. Then, we propose a novel architecture based on self-attention to parameterize the policy, by which we can incorporate a large variety of commonly used pairwise local operators such as 2-opt and node swap. Finally, we apply the RL framework to two representative routing problems, i.e. TSP and CVRP, and design an actor-critic algorithm to train the policy network.

Extensive results show that our method significantly outperforms existing DL based ones on TSP and CVRP. The learned policies are indeed more effective than traditional hand-crafted rules in guiding the improvement process, and can be further enhanced by simple ensemble strategies. Moreover, the policies generalize reasonably well to different problem sizes, initial solutions and even real-world dataset. Note that similar to previous works [17], [23], our aim is not to outperform highly optimized and specialized traditional solvers, but to present a *framework* that can automatically learn good search

heuristics without human guidance on different problem types, which is of great practical value when facing real-world problems and little domain knowledge is available.

II. Related Work

The application of deep neural networks to solve routing problems starts from the seminal work of Pointer Network [14], which is a RNN based Seq2Seq model and trained in a supervised way to solve TSP. On top of it, Bello et al. proposed to use RL to train Pointer Networks in [15] without the need of using optimal solutions to label the training samples, which is costly to obtain for NP-hard problems. Nevertheless, the RNN based encoder in Pointer Network inevitably embeds the sequential information of input, which the output sequence should be insensitive to. Therefore, Nazari et al. [16] proposed to linearly map the information of each node to high dimensional space with shared parameters, and solved CVRP and its variant where customers emerge dynamically.

Inspired by the Transformer architecture [24], Kool et al. [17] replaced the RNN based sequential structures in Seq2Seq models with the attention modules in both the encoder and decoder, and achieved better performances on both TSP and CVRP. Similarly, the permutation invariant pooling in the Transformer architecture was adopted in [18] to solve multiple TSP. The attention based mechanism was also applied for embedding in [19], but its performance relies on an additional 2-opt based local search.

Different from the Seq2Seq paradigm, Khalil et al. [10] adopted deep Q-Network to train a node selection heuristic that works within a greedy algorithm framework for solving TSP, where the internal states are represented using a Graph Neural Network (GNN) [25]. In [26], GNN is also used to learn normalized embeddings, which is used to reconstruct adjacent matrix of TSP graph, in supervised way.

All the above methods learn construction heuristics that only output one solution, and are able to outperform traditional non-learning based construction heuristics by a large margin. However, the solution quality is still quite far from optimality. Independently from our work, a NeuRewriter model was proposed recently in [23], which also learns a type of improvement heuristic. On CVRP, it outperforms [17], the best method of learning construction heuristics. However, it needs to train two policies to separately decide the rewritten region and solution selection, and relies on complex node features and customized local operations. In contrast, our method involves only one policy network, and uses only raw features and typical local operators that are commonly applied to routing problems. Empirically, our method outperforms NeuRewriter both in solution quality and generalization capability.

III. Preliminaries

Formally, an instance of routing problems can be defined on a graph with a set of n nodes $V = \{1, \ldots, n\}$. Each $v \in V$ has features $x(v)$. A solution $s = (s^1, \ldots, s^I)$ is a tour, i.e. a sequence of nodes with length I, with each element s^i ($i \in \{1, ..., I\}$) being a node in V. A feasible tour should satisfy

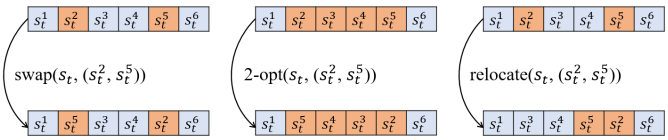


Fig. 1: Three typical pairwise operators for routing problems (left: node swap operator exchanges two locations; middle: 2-opt replaces two links by reversing a segment of locations; right: relocation puts one location after another location)

problem-specific constraints, which can be defined as follows for TSP and CVRP.

TSP. The tour visits each node exactly once, hence $I = n$.

CVRP. Another node v_d called depot is added to V. The original n nodes represent customers, each with demand $\delta(v)$. We define $\delta(v_d) = 0$. The tour consists of multiple routes $(r_1, \ldots, r_M)$, $M > 1$. Each route r_m starts from v_d, and visits a subset of customers R_m in sequence. Each customer must be visited exactly once but v_d could be visited multiple times, hence $I > n + 1$. Additionally, the total customer demand on each route can not surpass the given capacity D.

Let $c(s^i)$ be the coordinate of the ith location in s. Following previous research, we focus on minimizing the Euclidean distance of the tour s, denoted as $f(s)$.

Improvement Heuristics. Starting from an initial solution s_0, improvement heuristics iteratively replace the current solution s_t at step t with a new solution s_{t+1} picked from neighborhood $\mathcal{N}(s_t)$, towards the direction of minimizing f. The most important parts of improvement heuristics are: 1) the local operator that defines a specific operation on s_t, and accordingly yields $\mathcal{N}(s_t)$; 2) the policy used to pick s_{t+1} from $\mathcal{N}(s_t)$; 3) the rules of solution replacement or acceptance; 4) the termination conditions. Different combinations of the above aspects lead to different schemes of improvement heuristics. For example, hill climbing with the best-improvement strategy picks from $\mathcal{N}(s_t)$ the solution $\bar{s}_{t+1}$ with the smallest f, 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.

For routing problems, various local operators have been proposed [2]. In this paper, we focus on pairwise operators, which transform a solution s_t to s_{t+1} by performing operation l on a pair of nodes (s_t^i, s_t^j), i.e. $s_{t+1} = l(s_t, (s_t^i, s_t^j))$. The typical pairwise operators are 2-opt which reverse the node sequence between s_t^i and s_t^j, node swap which exchanges s_t^i and s_t^j, and etc. Particularly, Figure 1 illustrates node swap, 2-opt and relocation, which are typical pairwise operators for solving routing problems. 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 [27].

Traditional improvement heuristics use hand-crafted solution picking policies, which require substantial domain knowledge to design and could be limited in performance. For example, given the operators, the greedy improvement heuristics (e.g. hill climbing) could quickly get stuck in the local minimum. In this paper, we use deep RL to automatically learn high-quality solution picking policies that work in a

simple scheme with the following parts: 1) pairwise operators; 2) the "always accept" rule, i.e. the solution picked by the policy will always be accepted, to avoid being stuck in local minimum; 3) a user-specified maximum step T to stop the run. The best solution found in the process is returned after termination. We will show in the experiments that even with this simple scheme, our method can learn high-quality policies that outperform existing deep learning based methods. On the other hand, our method can be extended to guide more complex schemes (e.g. simulated annealing, tabu search). In addition, our method can potentially be applied to other combinatorial problems with sequential solution representations, e.g. scheduling. We plan to tap these potentials in the future.

IV. The Method

We first formulate the process of improvement heuristics as a RL task, and then introduce a self-attention based policy network, followed by the training algorithm. Finally, we provide the details of applying our method to TSP and CVRP.

A. RL Formulation

In this paper, we assume the problem instances are sampled from a distribution $\mathcal{D}$, and use RL to learn the solution picking policy for the improvement heuristic as 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 an instance at time step t, i.e. a sequence of nodes. The initial state s_0 is the initial solution 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 $s_{t+1} = (..., s_t^j, s_t^{j-1} ..., s_t^{i+1}, s_t^i ...)$.

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:

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

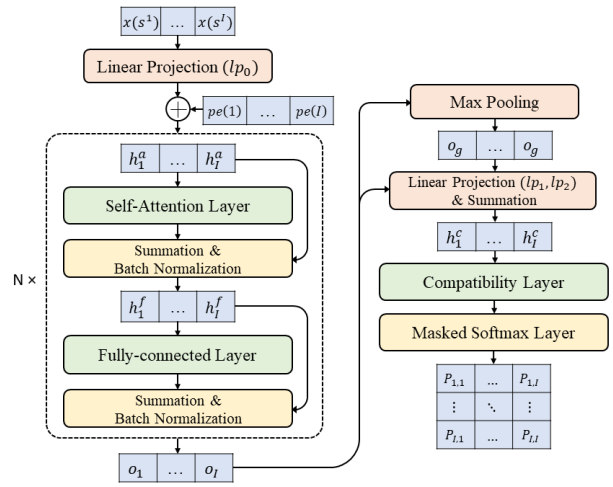


Fig. 2: The architecture of policy network (left: node embedding; right: node pair selection)

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 of an anytime algorithm. Hence, we consider the improvement process as a continuing task, and set $\gamma < 1$. Also note that the agent is allowed to experience states with poorer quality than the incumbent. Though these "bad" transitions have the lowest immediate reward (0), higher improvement could be gained in the long term which follows the principle of RL.

B. Policy Network

To learn the stochastic policy π in Equation (2), we parameterize it as a neural network π_θ, where θ refers to the trainable parameters. As visualized in Figure 2, the network comprises two parts, which learn node embedding and node pair selection, respectively. The former part elegantly embeds the nodes in sequence. The latter part adopts the compatibility computation in self-attention to produce a probability matrix of selecting each node pair. Thereby, each element in the matrix refers to the probability of selecting the corresponding node pair for local operation.

Node Embedding. Given the current state[1] $s = (s^1, \ldots, s^I)$, the features $x(s^i)$ of each node s^i are first projected to embeddings h_i by a shared linear transformation lp_0, with output dimension $d_m = 128$. But linear mapping alone cannot capture the position of each node in s, which is important since s is a sequence. Hence we add d_m-dimensional sinusoidal positional encodings $pe(i, \cdot)$ to node embeddings, refining them as $h_i = h_i + pe(i, \cdot)$. The sinusoidal positional encodings are a group of vectors defined by sine and cosine functions. In particular, $pe(i, \cdot)$ is defined as below:

$$pe(i, d) = sin(i/10000^{\frac{\lfloor d/2 \rfloor}{d_h}}), if\, d \bmod 2 = 0 \quad (3)$$

$$pe(i, d) = cos(i/10000^{\frac{\lfloor d/2 \rfloor}{d_h}}), if\, d \bmod 2 = 1 \quad (4)$$

[1]Step index t is omitted here for better readability.

where i is the location of node s_i in sequence and d is the dimension. $\lfloor \cdot \rfloor$ and mod means floor and modulo function. We also tried relative positional encoding (with embeddings of adjacent nodes added), but it performs worse than the sinusoidal one. To advance the node embeddings h_{s^i}, they are further processed by self-attention layer and fully-connected layer, each of which followed by residual connection and batch normalization. This is repeated N times for better feature extraction of the sequence, though the dimension of node embeddings keep d_h when output from each layer.

To derive better feature representations for policy learning, the above position-aware node embeddings are successively processed by N ($N = 3$) blocks with the same structure but different parameters. In each of them, we feed the embeedings to a self-attention layer, followed by a fully connected layer. After each of the two layers, skip connection [28] is applied, followed by batch normalization [29].

Self-attention layer. Self-attention transforms node embeddings through message passing and aggregation among nodes [25], [30]. We use single-head self-attention here, since the multi-head version does not lead to significant improvement in our experiments. Given the input matrix $H^a = [h_1^a, \ldots, h_I^a]$ with columns being node embeddings, the output of self-attention is computed as:

$$\tilde{H}^a = V_a \cdot \text{softmax}_\text{c}(\frac{K_a^T Q_a}{\sqrt{d_k}}), \tag{5}$$

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 $\text{softmax}_\text{c}(\cdot)$ is a column-wise softmax function so that the output of Equation (5) is the transformed node embeddings, i.e. $\tilde{H}^a = [\tilde{h}_1^a, \ldots, \tilde{h}_I^a]$.

Fully connected layer. This layer transforms each node embedding independently with shared parameters. Here we only involve one 512-dimensional hidden sublayer with ReLU activation function. Input and output dimensions retain 128.

Node Pair Selection. Given node embeddings o_i generated by the previous part, we aggregate them by max-pooling to get the *graph embedding*, i.e. $o_g = max(\{o_1 \ldots, o_I\})$. We then refine the node embeddings by converting o_i into h_i^c following $h_i^c = lp_1(o_i) + lp_2(o_g)$, where both lp_1 and lp_2 are linear projections that keep embedding size as 128. In doing so, the global graph information of an instance is effectively fused into its nodes. Then, we further process the node embeddings through a compatibility layer and a masked softmax layer to get the probability of selecting node pairs.

Compatibility layer. Inspired by [24], in which the compatibility effectively represents the relations between words in sentences, we adopt it to predict the node pair selection in a solution. While various compatibility functions have been proposed [31], [32], here we use a multiplicative version for better computational efficiency [24]. Given node embeddings $H^c = [h_1^c, \ldots, h_I^c]$, it is computed as the dot product of the query and key matrices, i.e. $Y = K_c^T Q_c$, where K_c and Q_c are computed in a similar way to K_a and Q_a in Equation (5), and the compatibility matrix $Y \in \mathbf{R}^{I \times I}$ reflects the scores of picking each node pair.

Masked softmax layer. With certain preprocessing, softmax is applied to the compatibility matrix such that:

$$\tilde{Y}_{ij} = \begin{cases} C \cdot \tanh(Y_{ij}), & \text{if } i \neq j, \\ -\infty, & \text{if } i = j, \end{cases} \tag{6}$$

$$P = \text{softmax}(\tilde{Y}), \tag{7}$$

where in Equation (6), we limit the values in the compatibility matrix within $[-C, C]$ by a $\tanh$ function; following [15], we set $C = 10$ to control the entropy of $\tilde{Y}_{ij}$; we also mask the diagonal elements, since picking pairs of same node is not meaningful. Therefore, the element p_{ij} in P represents the probability of selecting (s^i, s^j) for local operation. Rather than greedily picking the node pair with the maximum probability, we sample the probability matrix P for pair selection in both training and testing.

C. Training Algorithm

We adopt the actor-critic algorithm with Adam optimizer to train π_θ, which is a kind of policy gradient method, based on the REINFORCE [33] with extra trainable critic network updated by bootstrapping. The actor is the policy network we designed above. The critic v_ϕ is used to estimate the cumulative reward at each state. Our design of v_ϕ is similar to that of actor, except that: 1) mean-pooling is used to obtain graph embedding; 2) the fused node embeddings are processed by a fully connected layer similar to the one used in policy network, but with single-value output. We use $\mathbf{n}$-step return for efficient reward propagation and bias-variance trade-off [34]. 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 [35]. The complete algorithm is given in Algorithm 1, in which lines 5 to 17 process a batch of instances in parallel and accumulate their gradients to update the networks.

D. Deployment

In this paper, we solve two representative routing problems, i.e. TSP and CVRP. Keeping most of our approach the same, we specialize it for each problem as follows:

TSP. Given a solution, the node feature is a 2-dimensional vector that contains the node coordinate, i.e. $x(s^i) = c(s^i)$. Other parts keep the same as introduced above. To avoid solution cycling, the node pair selected at the previous step is masked to forbid the local operation to be reversed.

CVRP. Unlike TSP, solutions to CVRP have varying lengths caused by the times of visiting depot, even with the same number of customers. This makes it hard for batch training, and requires additional operations in each step to determine the number and positions of depots in the next solution. To resolve this issue, we add multiple dummy depots to the end of initial solutions, such that: 1) we can process a batch of instances using solutions with the same length; 2) the number and positions of depot in a solution (sequence) can be learned automatically, e.g. $s_t = (v_d, s^1, s^2, v_d, \ldots)$ can be

ALGORITHM 1: **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.

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

turned into $s_{t+1} = (v_d, v_d, s^2, s^1, \ldots)$ by 2-opt, with s_{t+1} being equivalent to $(v_d, s^2, s^1 \ldots)$. To better reflect the local structure, we define the node feature as a 7-dimensional vector $x(s^i) = (c(s^{i-1}), c(s^i), c(s^{i+1}), \delta(s^i))$, i.e. the coordinates of a node and its immediate left and right neighbors in s, along with its demand.[2] Lastly, we mask the node pairs in the matrix P that result in infeasible solutions, as well as the one selected at the previous step.

V. Experimental Results

The instances used in our experiments are Euclidean TSP and CVRP with 20, 50 and 100 nodes, respectively. We call them TSP20, CVRP20, etc. for convenience. We generate the instances following [17], [23], where the coordinates of each node are randomly sampled in the unit square $[0, 1] \times [0, 1]$, with a uniform distribution. For CVRP, the demand of each customer is uniformly sampled from $\{1, \ldots, 9\}$; the capacity D is 30, 40 and 50 for CVRP20, 50 and 100, respectively. More details are as follows.

TSP. In each training epoch for TSP, 10,240 random instances are generated on the fly and split into 10 batches. Training starts from random initial solutions, with mean tour distance of 8.48, 19.54 and 37.41 for TSP20, 50 and 100. As mentioned, we model improvement heuristics as continuing tasks. However, we only train the agent for a small step limit T=200, since the rewards in the early stages are more dense. We will show later that the trained policies generalize well to unseen initial solutions and much larger T in testing. We set γ to 0.99, and $\mathbf{n}$ for $\mathbf{n}$-step return to 4.

CVRP. Due to limited GPU memory, we only generate 3,840 instances in each epoch, also split into 10 batches. The initial solutions are created using the nearest insertion heuristic adopted in [23], with mean tour distance of 7.74, 13.47 and 20.36 for CVRP20, 50 and 100, which are far from optimality. We add dummy depots to the initial solutions, elongating them to the same length I^*=40, 100, and 125 for CVRP20, 50 and 100.[3] Empirically, for CVRP20, we set T=360 and $\mathbf{n}$=10; for CVRP50 and CVRP100, we set T=480 and $\mathbf{n}$=12. For all sizes, we set γ to 0.996.

We train 200 epochs for all problems, with initial learning rate 10^{-4} and decaying 0.99 per epoch for convergence. On a single Tesla V100 GPU, each epoch takes on average 8:20m (8 minutes and 20 seconds), 16:30m and 31:00m for TSP20, TSP50 and TSP100; 20:17m, 56:25m and 58:53m for CVRP20, CVRP50 and CVRP100. We have tried three common pairwise operators including 2-opt, node swap and relocation, with 2-opt producing the best results. Unless stated otherwise, we only apply 2-opt, and use the same settings of initial solutions and additional dummy depots in testing as those in training. Our code in Python and pre-trained models will be released soon.

A. Comparison with State-of-the-art Methods

We compare our method with a variety of baselines, including: 1) Concorde [36], an efficient exact solver specialized for TSP; 2) LKH3 [21], a well-known heuristic solver that achieves state-of-the-art performance on various routing problems; 3) OR-Tools, a mature and widely used solver for routing problems based on metaheuristics; 4) state-of-the-art DL based methods on TSP and CVRP, i.e. the attention model (AM) [17] and NeuRewriter [23], which learn construction and improvement heuristics, respectively.[4] We only evaluate the sampling version of AM, which samples $\mathbb{N}$ solutions using the learned construction policy and is much better than the greedy version. For fair comparison, we test AM with its default sampling size $\mathbb{N}$=1,280, and also $\mathbb{N}$=5,000 which is the maximum step of our method. For NeuRewriter, since pre-trained model is not provided and training is prohibitively time-consuming on our machine, we directly report the objective values in the original paper. For each problem size, all methods (except NeuRewriter) are tested on the same 10,000 random instances. For both our method and AM, we divide the instances into 10 batches and test each in parallel. We run Concorde, OR-Tools and LKH3 using the configurations in [17] on a Xeon W-2133 CPU@3.60 GHz. Single thread is used except LKH3, which is relatively slow hence we solve 16 instances in parallel. Since run time comparison is hard due to various factors (e.g. Python vs C++, GPU vs CPU), we follow [17] and report the total time for solving the 10,000 instances.

All results are summarized in Table I, where ours are displayed with step limit T=1,000, 3,000, 5,000. We can observe that when T=1,000, our method significantly outperforms OR-Tools for both TSP and CVRP with all sizes. As T increases,

[2] We define the left neighbor of s^1 as s^I, and the right neighbor of s^I as s^1, respectively.

[3] The number of depots in a CVRP solution cannot be greater than that of customers n, hence $I^* = 2n$ is ideal. But we cannot use I^*=200 for CVRP100 due to GPU memory constraint.

[4] We do not compare with other related methods such as [10], [15], [16], [19], [26] since they have already been outperformed by AM on the same benchmark used here [17].

TABLE I: Comparison with state-of-the-art methods

Methods	TSP20			TSP50			TSP100			CVRP20			CVRP50			CVRP100		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
Concorde	3.83	0.00%	5m	5.69	0.00%	13m	7.76	0.00%	1h	-	-	-	-	-	-	-	-	-
LKH3	3.83	0.00%	42s	5.69	0.00%	6m	7.76	0.00%	25m	6.11	0.00%	1h	10.38	0.00%	5h	15.64	0.00%	9h
OR-Tools	3.86	0.94%	1m	5.85	2.87%	5m	8.06	3.86%	23m	6.46	5.68%	2m	11.27	8.61%	13m	17.12	9.54%	46m
AM ($\mathbb{N}$=1,280)	3.83	0.06%	14m	5.72	0.48%	47m	7.94	2.32%	1.5h	6.26	2.56%	22m	10.61	2.20%	53m	16.17	3.34%	2h
AM ($\mathbb{N}$=5,000)	3.83	0.04%	47m	5.72	0.47%	2h	7.93	2.18%	5.5h	6.25	2.31%	1.5h	10.59	2.01%	3.5h	16.12	3.03%	8h
NeuRewriter	-	-	-	-	-	-	-	-	-	6.15	-	-	10.51	-	-	16.10	-	-
Ours (T=1,000)	**3.83**	**0.03%**	12m	5.74	0.83%	16m	8.01	3.24%	25m	6.16	0.90%	23m	10.71	3.16%	48m	16.30	4.16%	1h
Ours (T=3,000)	**3.83**	**0.00%**	39m	**5.71**	**0.34%**	45m	**7.91**	**1.85%**	1.5h	**6.14**	**0.61%**	1h	10.55	1.65%	2h	16.11	2.99%	3h
Ours (T=5,000)	**3.83**	**0.00%**	1h	**5.70**	**0.20%**	1.5h	**7.87**	**1.42%**	2h	**6.12**	**0.39%**	2h	**10.45**	**0.70%**	4h	**16.03**	**2.47%**	5h

[1] The gap is computed based on the best solutions here given by Concorde and LKH3.
[2] **Bold** results are those outperform the best deep learning based baseline.

our results consistently narrows the optimality gaps. AM also benefits from larger $\mathbb{N}$. However, the improvement is not much compared with our method. When T=3,000, our method consistently outperforms AM with $\mathbb{N}$=5,000 on all instance sets; for TSP, our method achieves almost the same result as Concorde on TSP20; for CVRP, our results are on par with NeuRewriter. Though the performance is already good, with additional 2,000 steps (i.e. T=5,000), our method still can further reduce the optimality gaps, and outperforms both baseline deep models with state-of-the-art results on TSP and CVRP. Note that our results still can be improved by increasing T, e.g. results of T=8,000 are 7.80 (0.48%) and 15.99 (2.24%) for TSP100 and CVRP100. The above observations justify the continuing design of our RL formulation. That is, despite training with small T, the policies perform fairly well with much larger step limits in testing. In terms of efficiency, run time of our method is roughly of the same order of magnitude as AM. This is well accepted considering the superiority of our method in solution quality. Moreover, with the increase of problem size, our run time rises much slower than other methods. For example, AM ($\mathbb{N}$=5,000) is faster than ours (T=5,000) on TSP20 and CVRP20, but is slower on TSP100 and CVRP100. OR-Tools is also faster than our method (T=1,000) on instances with 20 nodes, but on TSP100 and CVRP100, our method delivers far better solutions than OR-Tools with similar run times.

B. Comparison with Conventional Policies

The major difference between our method and the conventional improvement heuristics is that the policies of picking next solution is learned, instead of hand-crafted. To show that the automatically learned policies are indeed better than the hand-crafted rules, we compare our method with two widely used rules, *first-improvement* and *best-improvement*, which select the first and best cost-reducing solution in the neighborhood, respectively [37]. However, direct comparison is not fair because when reaching local minimum, they cannot pick any solution since no improvement exists. Hence we augment them with a simple but commonly used strategy *restart*, to randomly pick a solution from the whole space when no improvement in the neighborhood can be found. Then we apply them to the same improvement scheme as our method, i.e. 2-opt with the "always accept" rule and step limit T. We run all policies on the same test sets as those in Section V-A, using the same method to generate initial solutions as in training. The results are summarized in Table II. For the same T, our method consistently outperforms conventional rules in all instance sets, which justifies its superiority. The advantage of learned policies are more prominent on larger problems, for example our results with T=3,000 already outperforms both rules with T=5,000 on TSP100, CVRP50 and CVRP100. We can conclude that the learned policies can offer better guidance than conventional ones, especially when facing harder problems. Run time here is not directly comparable, since the conventional rules are implemented on CPU. However, our policy could be more efficient since the neural network directly picks the next solution, without the need of traversing the neighborhood as in the conventional ones.

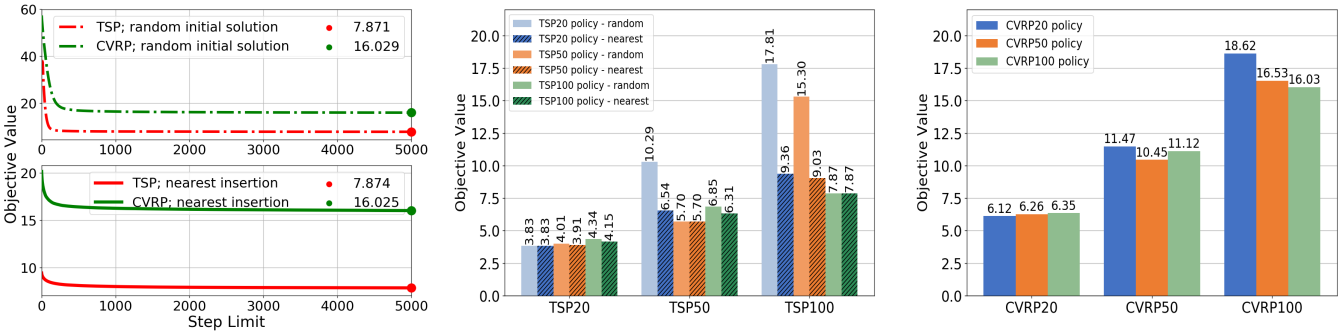
TABLE II: Comparison with conventional policies

	Methods	TSP			CVRP		
		20	50	100	20	50	100
First	T=1,000	3.84	5.81	8.17	6.18	11.08	17.14
	T=3,000	3.84	5.75	8.04	6.16	10.93	16.93
	T=5,000	3.84	5.73	8.00	6.15	10.87	16.85
Best	T=1,000	3.84	5.75	8.05	6.15	10.79	16.72
	T=3,000	3.84	5.71	7.99	6.14	10.70	16.61
	T=5,000	3.84	5.70	7.94	6.13	10.67	16.55
Ours	T=1,000	**3.83**	5.74	8.01	6.16	10.71	**16.30**
	T=3,000	**3.83**	5.71	**7.91**	6.14	**10.55**	**16.11**
	T=5,000	**3.83**	**5.70**	**7.87**	**6.12**	**10.45**	**16.03**

Bold means our method outperforms the best rule (T=5000).

C. Enhancement by Diversifying

The above results are all obtained by running the final learned policy after training only once for each instance. However, this could be less effective in terms of exploring the solution space, e.g. it might suffer from local minimum for some instances during searching. Here we show that, by coupling with two simple strategies to diversify the search process, the solution quality of our method could be further improved. The first strategy is multi-run, meaning that we directly run the final policy (i.e. the policy obtained after the last training epoch) multiple times. Since we sample the probability matrix P for pair selection during training and set



(a) Generalization to initial solutions (b) Generalization on TSP (c) Generalization on CVRP

Fig. 3: Generalization analysis

TABLE III: Enhanced results by diversifying

Methods		TSP 20	TSP 50	TSP 100	CVRP 20	CVRP 50	CVRP 100
MP(4)[1]	T=1,000	3.831	5.707	7.897	6.144	10.452	16.110
	T=3,000	3.831	5.701	7.839	6.134	10.426	16.055
	T=5,000	3.831	5.700	7.822	6.123	10.416	16.018
MP(8)	T=1,000	3.831	5.703	7.875	6.134	10.436	16.036
	T=3,000	3.831	5.700	7.824	6.127	10.404	16.000
	T=5,000	3.831	5.699	7.811	6.121	10.395	15.967
MR(4)[2]	T=1,000	3.832	5.707	7.895	6.144	10.482	16.110
	T=3,000	3.831	5.702	7.836	6.132	10.417	15.979
	T=5,000	3.831	5.700	7.821	6.122	10.399	15.923
MR(8)	T=1,000	3.831	5.703	7.866	6.135	10.445	16.057
	T=3,000	3.831	5.700	7.820	6.125	10.393	15.922
	T=5,000	3.831	**5.699**	**7.807**	**6.117**	**10.384**	**15.880**

[1] **MP(#)**: multi-policy strategy with the last # policies.
[2] **MR(#)**: multi-run strategy that runs the final policy # times.

certain value C in Equation (6) to control the entropy, we avoid the extremely dominant action choice in each step to some extent. Therefore, we could expect to obtain different high-quality solutions by running the same policy multiple times, and retrieve the best one as the final solution. The second strategy is multi-policy, for which we run polices of the last several training epochs instead of only the final one on an instance, each generating one solution. Intuitively, multi-policy could provide more diversity than multi-run, possibly reaching different regions of the solution space.

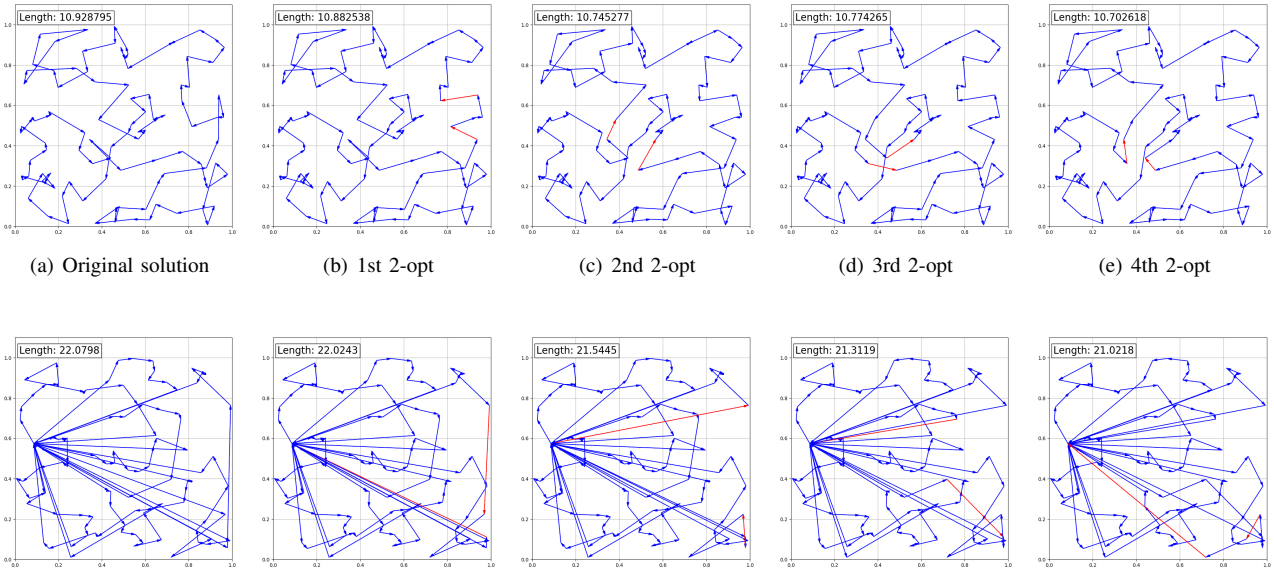
Table III shows the performance of the above two strategies. Clearly, the results for all problems are improved with the two strategies, compared to our results in Table II. We can see that with either strategy, the solution is consistently improved as more runs or policies are used, showing the benefit of diversifying. However, with the same number of runs or policies, multi-run outperforms multi-policy for all problems. This is probably because the policies only have small differences since they are in the last phase of training, and are not trained to be diverse. Notably, results of multi-run with 8 solutions are very close to the optimal solutions for TSP50 and CVRP50, with the gaps of 0.11% and 0.09%. It also narrows the optimality gaps for TSP100 and CVRP100 to 0.56% and 1.52%. We also test multi-run on CVRP100 by generating 16 and 32 solutions, and the objective values further decrease to 15.840 and 15.809 with the optimality gap of 1.24% and 1.08%.

The above analysis shows that both strategies are able to substantially improve the solution quality. Note that these strategies can be effectively parallelized, therefore little extra time is needed as long as the device has enough memory. Despite the inferior improvement in Table III, we would like to note that co-training multiple policies to collectively explore solution space is promising for promoting solution quality and efficiency, e.g. training multi-head self-attention and keeping each head searching a different solution space. We plan to investigate this in the future.

D. Generalization Analysis

Here we show that the policies learned by our method can generalize to situations unseen in training. All policies here are the ones used in Section V-A and V-B, without any further tuning. First, we evaluate the sensitivity to different initial solutions on the same problem size. We run the policies trained for TSP100 and CVRP100 on the same test sets used previously, with two types of initial solutions, i.e. generated randomly or by nearest insertion. We plot the average objective values of incumbents against time step T in Figure 3(a). We can observe that though the policy for TSP are trained with random initial solutions, it generalizes well to those created by nearest insertion, achieving almost the same quality (7.874 vs 7.871). Similarly, the policy for CVRP, which is trained with initial solutions generated by nearest insertion, can achieve comparable results when beginning from random ones (16.029 vs 16.025). These observations indicate that, for the same problem size, the learned policies generalize well to unseen initial solutions with different qualities.

Furthermore, we evaluate the generalization performance of our method on different problem sizes. Results on TSP are shown in Figure 3(b). We can see that when using random initial solutions, though the policies are able to improve (e.g. from 37.41 to 15.30 when TSP50 policy is used on TSP100), the final results are not very good. This is probably because the quality of random solutions are poor, which makes it hard for such cross distribution generalization. Hence we perform the same tests using initial solutions generated by nearest



(a) Original solution (b) 1st 2-opt (c) 2nd 2-opt (d) 3rd 2-opt (e) 4th 2-opt

(f) Original solution (g) 1st 2-opt (h) 2nd 2-opt (i) 3rd 2-opt (j) 4th 2-opt

Fig. 4: Visualization of learned policies on TSP and CVRP

insertion, and results in Figure 3(b) show that this leads to relatively good generalization. For the harder problem CVRP, results are shown in Figure 3(c), based on nearest insertion as in training. We can observe that our policies generalize well to CVRP with different sizes. The policies trained on CVRP50 and CVRP100 outperform OR-Tools on all sizes. In particular, all our results are better than those reported in [23] (e.g. 18.62 vs 18.86 and 16.53 vs 17.33 when CVRP20 and CVRP 50 policy are used on CVRP100, respectively), indicating a stronger generalization ability.

E. Visualization

Here we give a simple demonstration about what the learned policies based on 2-opt have done along the search process. In Figure 4, we visualize 5 successive states (solutions) with 4 actions (2-opt operations) for a TSP100 instance and a CVRP100 instance in Figure 4(a)-(e) and Figure 4(f)-(j), respectively. The red links in each state are two newly added links after two links in the previous state are deleted. From Figure 4(a)-(e), it can be easily seen that the 2-opt operations effectively decrease the objective value of the TSP100 instance from 10.93 to 10.70, with some cross links deleted. In addition, it verifies that multiple 2-opt could achieve complex operations as mentioned in Section III. For example, the 2nd and 3rd 2-opt together complete a 3-opt operation. For the CVRP100 insatnce in Figure 4(f)-(j), we can also observe that the learned policy constantly decreases the objective value. Different from the TSP100 instance, the learned policy on CVRP100 involves the inter-route and intra-route operations. For example, the 1st and 2nd actions are inter-route operations, generating new routes by destroying some previous routes; the 3rd and 4th actions are intra-route operations, deleting cross links in a single route as in TSP.

The visualization shows the potential of our method in learning policies that can execute various effective operations based on the 2-opt operator. We would like to note that, it is promising to represent multiple operators, and thus learn policies that simultaneously specify the local operation and next solution. Our policy network in this paper can be easily extended to empower multiple operators, for example using structures similar to the multi-head self-attention [24]. A more advanced approach is to use hierarchical RL [38] to decompose the selection of operator and next solution, and make decisions for them alternatively.

F. Test on Real-World Dataset

We further verify that our learned policies, though trained using synthetic data, performs reasonably well on instances from public benchmarks TSPlib[5][39] and CVRPlib[6] [40], which contains real-world problem instances. Note that these instances may follow distributions that are completely different those we used in training, in aspects such as node location patterns, customer demands, vehicle capacities, etc.

For TSPlib, we directly run our policy trained on TSP100 with T=3000, on the 36 symmetric and Euclidean instances up to 300 nodes, and compare with the AM policy also trained on TSP100. As shown in Table IV, our TSP100 policy performs much better than that of AM with 1,280 and 5,000 samples, indicating a stronger generalization ability on these instances. Besides the much smaller average optimality gap, our policy outperforms AM ($\mathbb{N}$=5,000) on 75% (27 out of 36) of these TSPlib instances. On the other hand, all learning based methods are inferior to OR-Tools. This is reasonable because in general, achieving good out-of-distribution generalization is very hard for machine learning models [41]. This could potentially be alleviated by adapting the trained policy to the (different) testing distribution, via transfer or few-shot

[5]http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html
[6]http://vrp.galgos.inf.puc-rio.br/index.php/en/

TABLE IV: Generalization on TSPlib

Instance	Opt.	OR-Tools	AM ($\mathbb{N}$=1,280)	AM ($\mathbb{N}$=5,000)	Ours (T=3,000)
eil51	426	436	436	**435**	438
berlin52	7,542	7,945	7,717	**7,668**	8,020
st70	675	683	691	**690**	706
eil76	538	561	564	**563**	575
pr76	108,159	111,104	111,605	111,250	**109,668**
rat99	1,211	1,232	1,483	**1,394**	1,419
KroA100	21,282	21,448	44,385	38,200	**25,196**
KroB100	22,141	23,006	35,921	35,511	**26,563**
KroC100	20,749	21,583	31,290	30,642	**25,343**
KroD100	21,294	21,636	34,775	32,211	**24,771**
KroE100	22,068	22,598	28,596	27,164	**26,903**
rd100	7,910	8,189	8,169	8,152	**7,915**
eil101	629	664	668	667	**658**
lin105	14,379	14,824	53,308	51,325	**18,194**
pr107	44,303	45,072	208,531	205,519	**53,056**
pr124	59,030	62,519	183,858	167,494	**66,010**
bier127	118,282	122,733	210,394	207,600	**142,707**
ch130	6,110	6,284	6,329	**6,316**	7,120
pr136	96,772	102,213	103,470	**102,877**	105,618
pr144	58,537	59,286	225,172	183,583	**71,006**
ch150	6,528	6,729	6,902	6,877	7,916
KroA150	26,524	27,592	44,854	42,335	**31,244**
KroB150	26,130	27,572	45,374	43,114	**31,407**
pr152	73,682	75,834	106,180	103,110	**85,616**
u159	42,080	45,778	124,951	115,372	**51,327**
rat195	2,323	2,389	3,798	3,661	**2,913**
d198	15,780	15,963	78,217	68,104	**17,962**
KroA200	29,368	29,741	62,013	58,643	**35,958**
KroB200	29,437	30,516	54,570	50,867	**36,412**
ts225	126,643	128,564	141,951	**141,628**	158,748
tsp225	3,916	4,046	25,887	24,816	**4,701**
pr226	80,369	82,968	105,724	101,992	**97,348**
gil262	2,378	2,519	2,695	**2,693**	2,963
pr264	49,135	51,954	361,160	338,506	**65,946**
a280	2,579	2,713	13,087	11,810	**2,989**
pr299	48,191	49,447	513,809	513,673	**59,786**
Avg. Gap	0	3.46%	146.12%	133.54%	17.12%

Bold means the best among three learning based methods.

learning, but is beyond the scope of this paper. Nevertheless, our method surprisingly outperforms OR-Tools on some instances, including pr76 (1.39%), rd100 (0.06%), and cil101 (4.61%). Moreover, the average gap of instances with 0-100, 101-200, and 201-300 nodes are 11.50%, 18.42%, and 23.61%, respectively, showing that the quality of our solution does not degrade very fast with the increase of problem size. Given the policy is only trained on random instances with 100 nodes, it generalizes reasonably well to large problems (up to 2 times larger than the training instances) and very different distributions.

For CVRPlib, we test the policy trained on CVRP100 with T=5000 on 22 instances with sizes between 101 to 200, each of which is generated following different depot positioning (Central, Eccentric, Random), customer positioning (Random, Clustered) and demand distribution (small or large variance). The results in Table V show that our policy significantly outperforms the AM method. Our policy achieves an average optimality gap that is more than two times smaller than AM ($\mathbb{N}$=5,000), and performs better on a majority of (13 out of 22) these CVRPlib instances. Similar to TSPlib, the quality degradation of our policy on CVRPlib is not fast, since the average gaps on the instances with 101-150 and 151-200 nodes are 12.36% and 16.27%, respectively.

TABLE V: Generalization on CVRPlib

Instance	Opt.	OR-Tools	AM ($\mathbb{N}$=1280)	AM ($\mathbb{N}$=5000)	Ours (T=5,000)
X-n101-k25	27,591	29,405	39,437	37,702	**29,716**
X-n106-k14	26,362	27,343	28,320	28,473	**27,642**
X-n110-k13	14,971	16,149	15,627	**15,443**	15,927
X-n115-k10	12,747	13,320	13,917	**13,745**	14,445
X-n120-k6	13,332	14,242	14,056	**13,937**	15,486
X-n125-k30	55,539	58,665	75,681	75,067	**60,423**
X-n129-k18	28,940	31,361	30,399	**30,176**	32,126
X-n134-k13	10,916	13,275	13,795	13,619	**12,669**
X-n139-k10	13,590	15,223	14,293	**14,215**	15,627
X-n143-k7	15,700	17,470	17,414	**17,397**	18,872
X-n148-k46	43,448	46,836	79,611	79,514	**50,563**
X-n153-k22	21,220	22,919	38,423	37,938	**26,088**
X-n157-k13	16,876	17,309	21,702	21,330	**19,771**
X-n162-k11	14,138	15,030	15,108	**15,085**	16,847
X-n167-k10	20,557	22,477	22,365	**22,285**	24,365
X-n172-k51	45,607	50,505	86,186	87,809	**51,108**
X-n176-k26	47,812	52,111	58,107	58,178	**57,131**
X-n181-k23	25,569	26,321	27,828	27,520	**27,173**
X-n186-k15	24,145	26,017	25,917	**25,757**	28,422
X-n190-k8	16,980	18,088	37,820	36,383	**20,145**
X-n195-k51	44,225	50,311	79,594	79,276	**51,763**
X-n200-k36	58,578	61,009	78,679	76,477	**64,200**
Avg. Gap	0	8.06%	32.97%	31.62%	14.27%

Bold means the best among three learning based methods.

VI. Conclusions and Future Work

This paper proposes a deep reinforcement learning framework to automatically learn improvement heuristics for routing problems. We design a novel neural architecture based on self-attention to enable learning with pairwise local operators. Empirically, our method outperforms state-of-the-art deep models on both TSP and CVRP, and further narrows the gap to highly optimized solvers. The learned policies generalize well to different initial solutions and problem sizes, and give reasonably good solutions on real-world datasets.

We would like to note that our method has great potentials in learning a variety of more complex improvement heuristics. First, we could extend the current framework to support multiple operators in the sense that the policy learns to pick both the operator and next solution. This could be achieved by using techniques such as multi-head self-attention or hierarchical RL. Second, despite the simple search scheme used in this paper, our framework can be applied to learn better solution picking policies for more advanced search schemes such as simulated annealing, tabu search, large neighborhood search, and LKH. Finally, our method is applicable to other important types of combinatorial optimization problems, e.g. scheduling. We plan to investigate these possibilities in the future.

References

[1] G. Gutin and A. P. Punnen, *The traveling salesman problem and its variations*, vol. 12. Springer Science & Business Media, 2006.

[2] P. Toth and D. Vigo, *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

[3] G. Laporte and Y. Nobert, "A branch and bound algorithm for the capacitated vehicle routing problem," *Operations-Research-Spektrum*, vol. 5, no. 2, pp. 77–85, 1983.

[4] J. Lysgaard, A. N. Letchford, and R. W. Eglese, "A new branch-and-cut algorithm for the capacitated vehicle routing problem," *Mathematical Programming*, vol. 100, no. 2, pp. 423–445, 2004.

[5] N. Bansal, A. Blum, S. Chawla, and A. Meyerson, "Approximation algorithms for deadline-tsp and vehicle routing with time-windows," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pp. 166–174, 2004.

[6] A. Das and C. Mathieu, "A quasi-polynomial time approximation scheme for euclidean capacitated vehicle routing," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 390–403, SIAM, 2010.

[7] R. Hassin and A. Keinan, "Greedy heuristics with regret, with application to the cheapest insertion algorithm for the tsp," *Operations Research Letters*, vol. 36, no. 2, pp. 243–246, 2008.

[8] T. Pichpibul and R. Kawtummachai, "An improved clarke and wright savings algorithm for the capacitated vehicle routing problem," *ScienceAsia*, vol. 38, no. 3, pp. 307–318, 2012.

[9] S. Ropke and D. Pisinger, "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows," *Transportation science*, vol. 40, no. 4, pp. 455–472, 2006.

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

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

[12] Y. Keneshloo, T. Shi, N. Ramakrishnan, and C. K. Reddy, "Deep reinforcement learning for sequence-to-sequence models," *IEEE Transactions on Neural Networks and Learning Systems*, 2019.

[13] B. Zhang, D. Xiong, J. Xie, and J. Su, "Neural machine translation with gru-gated attention model," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

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

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

[16] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Proceedings of the 32nd Conference on Neural Information Processing Systems (NIPS)*, pp. 9839–9849, 2018.

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

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

[19] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the tsp by policy gradient," in *Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pp. 170–181, 2018.

[20] D. S. Lai, O. C. Demirag, and J. M. Leung, "A tabu search heuristic for the heterogeneous vehicle routing problem on a multigraph," *Transportation Research Part E: Logistics and Transportation Review*, vol. 86, pp. 32–52, 2016.

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

[22] L. Wei, Z. Zhang, D. Zhang, and S. C. Leung, "A simulated annealing algorithm for the capacitated vehicle routing problem with two-dimensional loading constraints," *European Journal of Operational Research*, vol. 265, no. 3, pp. 843–859, 2018.

[23] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Advances in Neural Information Processing Systems*, pp. 6278–6289, 2019.

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

[25] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[26] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A note on learning algorithms for quadratic assignment with graph neural networks," *stat*, vol. 1050, p. 22, 2017.

[27] K. Helsgaun, "General k-opt submoves for the lin–kernighan tsp heuristic," *Mathematical Programming Computation*, vol. 1, no. 2-3, pp. 119–163, 2009.

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

[29] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp. 448–456, 2015.

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

[31] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the 3rh International Conference on Learning Representations (ICLR)*, 2015.

[32] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 International Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1412–1421, 2015.

[33] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

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

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

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

[37] P. Hansen and N. Mladenović, "First vs. best improvement: An empirical study," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 802–817, 2006.

[38] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in neural information processing systems*, pp. 3675–3683, 2016.

[39] G. Reinelt, "Tspliba traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[40] E. Uchoa, D. Pecin, A. Pessoa, M. Poggi, T. Vidal, and A. Subramanian, "New benchmark instances for the capacitated vehicle routing problem," *European Journal of Operational Research*, vol. 257, no. 3, pp. 845–858, 2017.

[41] Y. Sun, X. Wang, Z. Liu, J. Miller, A. A. Efros, and M. Hardt, "Test-time training for out-of-distribution generalization," *arXiv preprint arXiv:1909.13231*, 2019.