Solving Traveling Salesman Problems Using Parallel Environments in Reinforcement Learning

Shaohua Hu
Institute of Automation, Chinese Academy of Sciences,
and Beijing Jiaotong University
Beijing, China
`22711091@bjtu.edu.cn`

Ming Zhu, Tenghai Qiu, Zhiqiang Pu
National Key Laboratory of Cognition and Decision Intelligence for Complex Systems,
Institute of Automation, Chinese Academy of Sciences,
and University of Chinese Academy of Sciences,
Beijing, China
`zhumingpassional@gmail.com`

Xiaolin Ai
National Key Laboratory of Cognition and Decision Intelligence for Complex Systems,
Institute of Automation, Chinese Academy of Sciences, Beijing, China
and National Key Laboratory of Information Systems Engineering, Changsha Hunan, China

Abstract

Traveling salesman problems (TSP) are NP-hard and difficult to solve since the search space increases significantly with problem size. Reinforcement learning (RL) is a promising method for its powerful search abilities with the help of GPUs. However, sampling speed is a bottleneck since training requires a large number of samples, and current methods ignore this issue. We propose to use GPU-based parallel environments to increase the sampling speed in RL; furthermore, we use powerful neural networks transformers with self-attention to enhance the policy so that the long-distance topology can be learned. The experimental results demonstrate the advantages of GPU-based parallel environments in terms of sampling speed and objective values. The code is available at: https://github.com/zhumingpassional/RLSolver.

1 Introduction

Traveling Salesman Problems (TSP) are classic NP-hard in the field of combinatorial optimization (CO), widely applied in various practical scenarios such as logistics and delivery. Heuristic algorithms, such as the Concorde [1] and Lin-Kernighan-Helsgaun (LKH-3) [6], often struggle to handle instances due to the combinatorial explosion. Their parameters rely on expert knowledge and complex tuning; therefore, the optimality can not be guaranteed [2, 7].

In recent years, neural combinatorial optimization (NCO) has provided a new solution to this domain through deep reinforcement learning (RL) [9]. By modeling the TSP as a sequential decision problem, neural networks can learn end-to-end solution strategies without relying on pre-computed solution pool. In particular, transformer-based architectures, with their ability to model global dependencies, have shown great potential in solving TSP [7, 8]. However, the practical application of this method is severely constrained by sampling speed, greatly limiting the feasibility of research iterations and industrial deployment.

When solving the TSP, the practical application of neural network methods is often limited by the core bottleneck of sampling speed. Although distributed RL frameworks such as A3C and IMPALA have demonstrated significant effectiveness in other fields [10, 5], for highly structured CO problems like TSP, simple parallelization strategies often fail to fully exploit their inherent symmetry and constraint properties. The main challenges lie in the sparse reward mechanism in CO, which hinders the effective propagation of learning signals. Additionally, the serial interaction between agents and the environment in standard training frameworks significantly reduces data sampling speed, and high-variance gradient estimates based on small-batch updates further exacerbate convergence instability. Our method introduces a parallel environment mechanism to improve the sampling speed and training efficiency for RL methods.

This paper proposes parallel policy optimization algorithm based on Policy Optimization with Multiple Optima (POMO) [8], which breaks through computational bottlenecks through multi-level parallelization and system-level optimization. The distributed POMO training architecture combines PyTorch Distributed Data Parallel (DDP) [11] and the POMO algorithm to achieve data parallel training on multiple GPUs, while each GPU internally generates multiple parallel trajectories through POMO. The POMO framework [8] leverages the inherent symmetry of the TSP to simultaneously generate solutions from multiple starting nodes, significantly improving solution quality and training efficiency.

We conducted a review of existing open-source RL frameworks, evaluating their development and maintenance status, functional coverage, and technical support. The results are shown in Table 1. First, most frameworks lack active maintenance, posing compatibility risks with modern libraries and affecting the reproducibility of research. Second, there are huge differences in functional coverage, with the number of environments ranging from 3 to more than 100 and data source support ranging from 1 to 12. This imbalance limits the ability of algorithms to be verified in different scenarios. Finally, there are significant differences in technical architecture, with some frameworks lacking GPU acceleration support, which leading to low training efficiency. Therefore, we believe that it is necessary to establish a unified evaluation benchmark.

Table 1: Comparison of open-source RL frameworks

Attribute	**Jumanji**	**rl4co**	**or-gym**	**RLSolver (Ours)**
Number of Envs	22	27	21	24
Supported Data Sources	24	7	21	13
Last Commit (Age)	>1y	<1y	>2y	<1w
Approved PRs (6 mo.)	0	0	0	>20
GPU Acceleration	✓	✓	×	✓

The key contributions of this paper are as follows:

1. We propose to use GPU-based parallel environments in RL methods for the TSP. Through reasonable parallelization design, training time is reduced from tens of hours to a few hours, significantly improving the practicality of NCO methods.
2. We designed a multi-level parallel architecture that effectively combines distributed data parallelism and POMO's trajectory-level parallelism. By fully leveraging the symmetry of the TSP during training, we achieve efficient sample collection and stable gradient updates.
3. We validated the effectiveness of our method on standard TSP benchmarks. Experiments show that the framework achieves a gap of 0.1%-1% from the optimal solution on TSP-20 to TSP-100, demonstrating the significant role of parallel environments in enhancing the performance of neural TSP solvers.

2 Methodology

This section provides GPU-based parallel environments in RL methods for solving TSP.

2.1 Problem Modeling

We model TSP in two-dimensional Euclidean space as a finite-time Markov decision process (MDP). Given a set of city coordinates $\{x_i \in [0,1]^2\}_{i=1}^n$, the goal is to construct the shortest Hamiltonian cycle that visits all cities exactly once and returns to the starting point.

In the MDP, the agent constructs a solution by sequentially selecting the next city to visit. Specifically, the state space $\mathcal{S}$ contains the fixed coordinates of all cities, the current city, the starting city, and the mask information of the cities that have been visited. At time step t, the state s_t encodes the partially constructed path $\pi_t = (c_1, c_2, \ldots, c_t)$. The action space $\mathcal{A}$ is the set of unvisited cities, which is a discrete space that dynamically shrinks as the decision process progresses. The reward function is sparsely designed, with a terminal reward equal to the total length of the path $R(s_n) = -L(\pi_n)$ given only when the entire path is completed, and zero rewards for intermediate steps. State transitions are deterministic; after selecting city c_{t+1}, the system transitions to a new state s_{t+1} containing updated path information.

This sequential modeling approach naturally leads to a deep learning-based solution strategy. Since TSP is essentially a graph structure problem rather than a pure sequence problem, the Transformer architecture is well suited to capture this global spatial relationship.

2.2 Reinforcement Learning Algorithm and Network Architecture

RL Framework

Our method uses Policy Optimization with Multiple Optima (POMO) as its main training framework. The key point of POMO is to utilize the rotational symmetry of TSP solutions, i.e., for instances with n cities, POMO constructs n trajectories in parallel, each starting from a different city. This design expands the training signal for each instance from 1 trajectory to n trajectories, thereby exploring multiple high-quality local optima in the space.

POMO uses the average reward of all parallel trajectories as the shared baseline, $b_i = \frac{1}{n}\sum_{k=1}^n R_{i,k}$, where $R_{i,k}$ denotes the trajectory reward of the i-th instance starting from the k-th city. Based on the REINFORCE algorithm, we minimize the policy gradient loss:

$$\mathcal{L}(\theta) = \frac{1}{Bn}\sum_{i=1}^{B}\sum_{k=1}^{n}(R_{i,k} - b_i)\sum_{t=1}^{n}\log \pi_\theta(a_{i,k,t}|s_{i,k,t}) \tag{1}$$

where B is the batch size, and π_θ is the parameterized policy network. This shared baseline mechanism significantly reduces the variance of gradient estimates, making training more stable and efficient.

Neural Network Architecture

Our method is based on a Transformer encoder-decoder architecture. The encoder first maps two-dimensional coordinates to a high-dimensional embedding space through linear projection, and then uses L layers of self-attention modules to extract global relationship features between nodes. The embedded $\mathbf{h}_j \in \mathbb{R}^d$ output by the encoder has permutation invariance, which perfectly matches the node disorder of TSP. We use PyTorch's scaled dot-product attention fusion kernel and automatically select Flash Attention [4] on supported hardware to improve computational efficiency and memory utilization.

The decoder gradually constructs the path in an autoregressive manner. Let the encoder's output for each city be $\{\mathbf{h}_i\}_{i=1}^n$, and let the global context be $\bar{\mathbf{h}} = \frac{1}{n}\sum_{j=1}^n \mathbf{h}_j$. At step t, the decoder concatenates the global context, the current city embedding $\mathbf{h}_{\pi_t}$, and the initial city embedding $\mathbf{h}_{\pi_1}$, and performs a linear projection to obtain the query vector $\mathbf{q}_t = W_q\left[\bar{\mathbf{h}} \oplus \mathbf{h}_{\pi_t} \oplus \mathbf{h}_{\pi_1}\right]$. Then, through a cross-attention mechanism, using $\mathbf{q}_t$ as the query and the encoder output as the key-value pairs, the context vector $\mathbf{c}_t$ is calculated. The scores of candidate cities are calculated by the dot product of the context vector and the encoder output: $s_t(i) = C \cdot \tanh\left(\frac{\mathbf{c}_t^\top \mathbf{h}_i}{\sqrt{d}}\right) + m_t(i)$, where C is the clipping coefficient, and the masking term $m_t(i) = 0$ if $i \notin S_t$ and $m_t(i) = -\infty$ if $i \in S_t$, to ensure that probabilities are only assigned to unvisited cities. Applying softmax to the scores yields the selection distribution $p_t(i) = \mathrm{softmax}_i\left(s_t(i)\right)$. The next node is sampled according to $\pi_{t+1} \sim \text{Categorical}\big(p_t(\cdot)\big)$ and added to the visited set $S_{t+1} = S_t \cup \{\pi_{t+1}\}$. The conditional probability of the entire path can be written as $P_\theta(\pi_{2:n} \mid \pi_1, H) = \prod_{t=1}^{n-1} p_t\big(\pi_{t+1}\big)$.

Parallel Environments

Our parallelization is divided into two levels. Within the parallelization level, n POMO trajectories are started for each instance on a single GPU. Between parallelization levels, Distributed Data Parallel (DDP) is used to perform parallelization on K blocks of GPUs, with each GPU processing B/K instances and synchronizing gradients between all processes. We use torch.compile to compile and optimize critical rollout and loss calculation functions, spreading Python overhead across CUDA Graphs. This allows us to process $B \times n$ instances simultaneously in a single training step. We use torch.compile to compile and optimize the critical rollout and loss calculation functions, offloading Python overhead to CUDA Graphs. This allows us to process $B \times n$ trajectories simultaneously in a single training step, significantly improving sampling speed compared to traditional methods.

3 Experiment

3.1 Convergence and the number of environments

GPU-based parallel environments can improve the quality of solutions during training. We conducted experiments on the TSP-30 dataset, setting the number of epochs to 50 and keeping the number of training steps per epoch constant (50 steps). The internal POMO size of our method was set to 30 (equal to the problem size), and the total number of training steps was kept constant (50 steps). Figure 1 shows the change in the objective value with the number of epochs in different GPU parallel environments (batch size). We tested different parallel environment scales ranging from 4 to 1024. The experimental results indicate that increasing the number of GPU parallel environments improves performance significantly. More parallel environments help the model identify and learn high-quality strategy patterns faster, thereby achieving faster convergence and better performance.

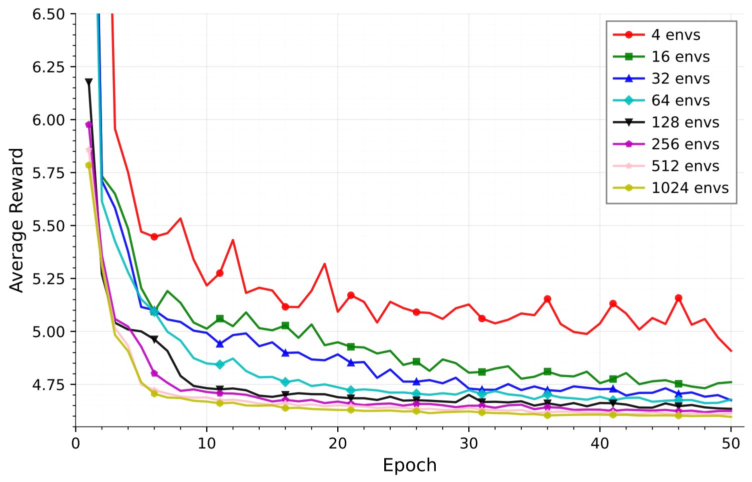

Figure 1: Performance with different numbers of GPU parallel environments

3.2 Speed of data sampling

We evaluated the sampling speed of our method and Jumanji [3] across different number of environments to demonstrate the scalability and performance characteristics of our parallel optimization method. Jumanji is a high-performance RL library that provides a suite of parallel environments and scalable simulation tools. All benchmark experiments were conducted using NVIDIA A6000 GPUs.

As shown in Figure 2, our method demonstrates superior sampling speed at smaller number of environments, with this advantage being particularly pronounced in the medium-to-large scale range (128–1024). The sampling speeds of both methods scale approximately linearly with increasing

Table 2: Comparison of TSP solvers.

	TSP20			TSP50			TSP100		
Method	Length	Gap (%)	Time (ms)	Length	Gap (%)	Time (ms)	Length	Gap (%)	Time (ms)
LKH3	3.845	0.00	9.00	5.687	0.00	80.0	7.754	0.00	390.0
Concorde	3.845	0.00	8.10	5.687	0.00	28.0	7.754	0.00	60.0
AM-greedy	3.856	0.28	0.39	5.777	1.59	0.50	8.086	4.28	0.72
AM-sampling	3.847	0.06	3.80	5.713	0.46	17.0	7.931	2.29	62.7
Ours	**3.847**	**0.07**	**0.55**	**5.712**	**0.45**	**1.37**	**7.903**	**1.92**	**2.81**

of environments. However, Jumanji exhibits a little better scaling properties when the number of environments is 4096.

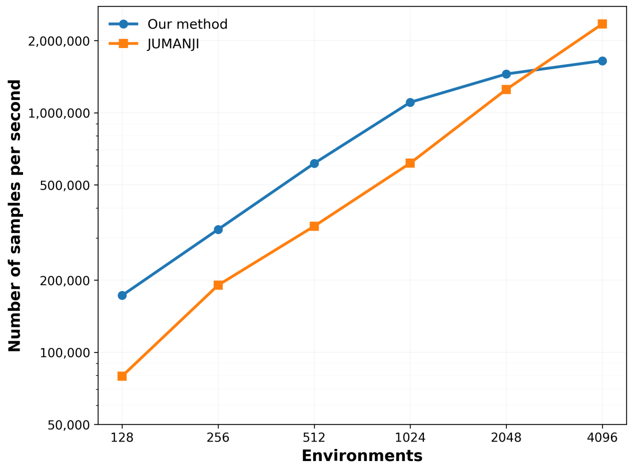

Figure 2: Sampling efficiency comparison between our method and Jumanji

3.3 Performance

This experiment was conducted on a server equipped with four NVIDIA A6000 GPUs, each with 48 GB of VRAM. The embedding dimension and hidden layer dimension of the model were both set to 128, and the multi-head attention mechanism used 8 attention heads. To control the randomness of action selection, we used a temperature parameter $C = 15.0 \times (n/20)^{0.5}$, where n is the number of nodes in TSP, enabling the model to automatically adjust the balance between exploration and exploitation based on the problem scale.

The training dataset consists of 100,000 randomly generated TSP instances, with the number of nodes set to 20, 50, and 100, and node coordinates uniformly distributed in the unit square $[0, 1) \times [0, 1)$. We independently generated 1,000 test instances using different random seeds. For RL-based methods, we uniformly set the model training to 300 epochs.In the evaluation phase, we compare our method with OR-based methods and RL-based methods, assessing the gap from the optimal solution and the solution speed.

The experimental results are shown in Table 2. In terms of solution quality, our method achieves gaps of 0.07%, 0.45%, and 1.92% on TSP20, TSP50, and TSP100, respectively. All of them are better than or close to AM-sampling and significantly better than AM-greedy. More importantly, our method demonstrates a significant advantage in solution speed, being much faster than AM-sampling and OR solvers. As the problem size increases from 20 to 100 nodes, our method's solution time increases by only approximately 5 times, while AM-sampling increases by approximately 16 times and LKH3 increases by approximately 43 times, demonstrating excellent scalability. This indicates the applicability of our method for handling TSP.

4 Conclusion

In this paper, we propose parallel environments in reinforcement learning for TSP. By combining GPU parallel environments and the POMO algorithm, we overcome the computational bottlenecks of data sampling in practical applications. Experimental results show that our method can achieve high-quality solutions, high sampling speed and training efficiency. In the future, we will optimize the management of multiple GPUs and memory.

5 Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant 62322316, the National Natural Science Foundation of China under Grant No. 62503472, the Open Fund of National Key Laboratory of Information Systems Engineering (No. 6142101240203).

References

[1] David L Applegate, Robert E Bixby, Václav Chvátal, and William J Cook. The traveling salesman problem: A computational study. Princeton University Press, 2007.

[2] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[3] Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, et al. Jumanji: A diverse suite of scalable reinforcement learning environments in jax. *arXiv preprint arXiv:2306.09884*, 2024.

[4] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[5] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018.

[6] Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. Technical report, Roskilde University, 2017.

[7] Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[8] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. In *Advances in Neural Information Processing Systems*, 2020.

[9] Xiao-Yang Liu, Ming Zhu, and Jiahao Zheng. ElegantRL: Massively parallel framework for cloud-native deep reinforcement learning. `https://github.com/AI4Finance-Foundation/ElegantRL`.

[10] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, et al. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937. PMLR, 2016.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.