

# 面向车辆路径问题的可泛化神经求解器：通过可迁移的局部策略集成方法

## 摘要

机器学习在解决 NP 难的组合优化问题方面已展现出巨大潜力。其中一种流行的方法是利用深度神经网络来构建解决方案，这种方法因其高效性和较低的专业知识要求而备受关注。然而，许多针对车辆路径问题（VRPs）的神经构造方法主要集中在具有特定节点分布和有限规模的合成问题实例上，这限制了它们在处理现实世界中常见的复杂、未知节点分布以及大规模问题时的表现。为了提升神经 VRP 求解器的实用性，我们设计了一种辅助策略，该策略通过学习局部可迁移的拓扑特征，称为局部策略，并将其与传统的构造策略（从 VRP 实例的全局信息中学习）相结合，形成一种集成策略。通过联合训练，这些聚合策略能够协同合作、互补优势，从而显著提升模型的泛化能力。在两个知名基准测试集—TSP 问题的 TSPLIB 和 CVRP 问题的 CVRPLIB 上的实验结果表明，这种集成策略不仅在跨分布和跨规模的泛化性能上取得了显著提升，而且在包含数千个节点的真实世界问题中也表现出色。

**关键词：**车辆路径问题；强化学习；神经组合优化

## 1 引言

车辆路径问题（Vehicle Routing Problems, VRPs）是一类典型的 NP-hard 组合优化问题，在物流、配送等领域有着广泛的应用。传统的启发式算法如 LKH 和 HGS 等在解决大规模实例时存在效率低、依赖专家知识等局限性。近年来，随着机器学习技术的发展，利用深度神经网络来解决组合优化问题成为研究热点。神经组合优化（Neural Combinatorial Optimization, NCO）通过学习启发式规则和生成解决方案，展现出高效性和对专业知识需求少的优势。然而，现有的神经方法在解决实际问题时仍面临泛化能力不足的问题，尤其是在面对复杂节点分布和大规模实例时，其性能显著下降。

本次选择复现的论文提出了一种集成全局策略和可转移局部策略的方法，该论文通过联合训练和策略集成，实现了对不同问题实例的高效求解和良好泛化，丰富了神经组合优化的研究内容。在多个基准测试集上的实验结果表明，其在跨分布和跨规模问题上的性能显著优于现有方法，并且在实际问题中表现出良好的应用潜力。这为物流行业的车辆路径规划提供了一种新的解决方案，有助于提高物流效率、降低成本。推动了神经方法在组合优化领域的应用和发展，为解决其他类似的 NP-hard 问题提供了参考和借鉴。

## 2 相关工作

### 2.1 传统求解方法

传统求解组合优化问题主要包括精确方法和近似方法两大类。精确方法是可以求解得到问题全局最优解的一类算法，主要包括分支定界法 [18] 和动态规划法 [4]，其均采用分而治之的思想通过将原问题分解为子问题的方式进行求解 [9]，通过不断迭代求解得到问题的全局最优解。近似方法是可以求解得到问题局部最优解的方法，主要包括近似算法和启发式算法两类 [9]。近似算法是可以得到有质量保证的解的方法，包括贪心算法、局部搜索算法、线性规划和松弛算法、序列算法等 [11, 21, 24]；启发式算法是利用设定的启发式规则对解空间进行搜索的一类方法，能够在可行时间内找到一个较好的解，但是对解的质量没有保证，文献中用来求解组合优化问题的启发式算法主要包括模拟退火算法 [20]、禁忌搜索 [23]、进化算法 [10]、蚁群优化算法 [7]、粒子群算法 [25] 等。

### 2.2 神经组合优化

最近，机器学习被应用于解决组合优化问题 [5]，这种利用深度神经网络来提取特征、学习启发式方法并生成解决方案，被称为神经组合优化（NCO）[3]。根据训练方案，NCO 包括监督学习（SL）和强化学习（RL）方法。SL 方法旨在借助传统求解器所给出的最优解标签去学习特定的模式或策略。指针网络 [22] 是一种融合了注意力机制 [2] 的序列到序列模型，该模型通过逐步解码的方式构建解决方案，其训练过程由 Concorde 提供的专家级解决方案予以引导。Joshi 等人 [13] 设计了深度图卷积网络（GCN），用于构建高效的旅行商问题（TSP）图表示。在此模型中，神经网络采用束搜索的方式，以非自回归形式输出路径。Kool 等人 [14] 从经过监督预训练的 GCN 中提取信息，并将其作为一种可学习的神经启发式方法，这种方法能够助力动态规划算法实现更为高效的搜索。Hudson 等人 [12] 提出了一种混合数据驱动方法，该方法将图神经网络与引导局部搜索相结合，相较于三种最新的基于学习的 TSP 算法，此方法收敛至最优解的速度更快。

在求解旅行商问题（TSP）时，由于无法即时获取其最优解标签，近年来，无需依赖最优标签的强化学习方法逐渐受到广泛关注。根据生成解决方案的过程，基于强化学习的方法可以大致分为改进方法和构建方法。改进方法学习如何迭代地提高解决方案的质量，例如，Lu 等人 [16] 通过选择合适的算子来提升解决方案的质量。改进方法具有利用人类知识的优势，但存在搜索效率有限和推理延迟高的缺点。构建方法学习如何顺序地扩展一个部分解决方案，直到构建出一个完整的解决方案。这一过程可以被建模为马尔可夫决策过程（MDP），并通过强化学习来解决。构建方法在计算效率上非常高，可以在几秒钟内解决数百个实例，然而，大多数构建方法只在具有简单节点分布和有限规模的问题上表现良好，在实际问题中常见的跨分布和跨规模问题上，它们的性能会急剧下降。为了推广到跨分布和跨规模的问题实例，最近提出了一些方法。其中一些方法专注于推广到大规模问题。它们通常利用分而治之的框架，可以将小规模模型扩展为可推广的模型，如 cheng 等人提出了 SO [6]，在每次迭代中，从当前解中采样多个子问题，将它们并行输入神经网络模型，选择改进最大的子问题进行优化更新当前解，并按特定频率采用破坏 - 修复方法避免局部最优。Drakulic 等人 [8] 引入双概念，在确定性 MDP 中，若两状态产生相同动作 - 奖励序列则为双相似。对于给定组合优

化问题的 MDP，可通过双模拟商利用其对称性简化问题，减少状态空间。此外，一些新颖的神经网络架构也被提出，如 Luo 等人 [17] 提出 Light Encoder Heavy Decoder 的结构和 Sun 等人提出用扩散模型 [19] 求解，被发现对跨规模推广很有用。

### 3 本文方法

#### 3.1 本文方法概述

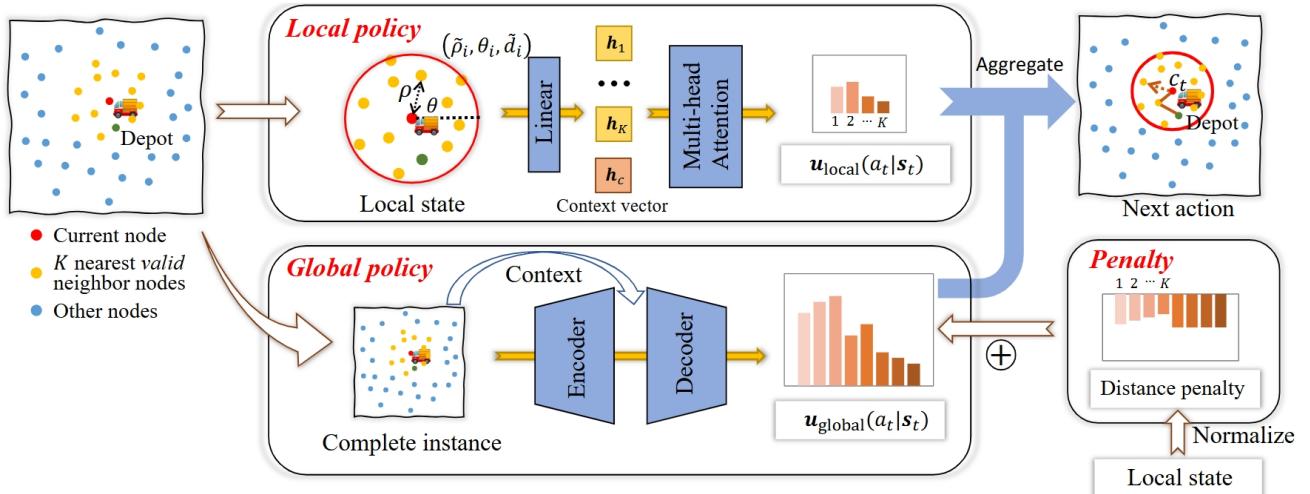


图 1. ELG 框架示意图

ELG 是 Global policy 和 Local policy 的组合，其流程如图 1 所示，其中 Global policy 使用的是 POMO [15]，Local policy 是作者设计的网络，由一层 Linear 和一层 MHA 组成。Global policy 主要关注于全局的信息，而 Local policy 则关注于当前选择的节点的 k 个最近节点的邻域。ELG 首先单独训练 Global policy，等到训练到一定阶段时加入 Local policy，两者同时训练。

#### 3.2 Local policy

ELG 使用极坐标  $(\rho, \theta)$  表示节点的位置，对于当前选择的节点  $c_t$ ，我们希望根据  $c_t$  的邻域信息选择下一个点  $c_{t+1}$ ，为此，ELG 考虑  $c_t$  的  $k$  个最近节点  $\mathcal{N}_K(c_t)$ ，对于邻域的节点位置，极径  $\rho$  是邻域节点与当前选择节点的距离，角度  $\theta$  则是极径的角度，并且他们的极径会被规范到  $[0, 1]$  之间，如公式 1 所示，这样无论节点的分布与问题规模如何变化，都可以规范到一个单位圆中，提高了模型的泛化能力。

$$\tilde{\rho}_i = \frac{\rho_i}{\max\{\rho_i | n_i \in \mathcal{N}_K(c_t)\}} \quad (1)$$

对于邻域内的节点，ELG 通过网络计算出他们的 Attention score  $u_{local}^i$  来确定下一个访问的节点，对于邻域外的节点，直接设置为 0，即只关注邻域内的节点，如公式 2 所示。

$$u_{\text{local}}^i = \begin{cases} (g_\theta(s_t))_i, & \text{if } n_i \in \mathcal{N}_K(c_t), \\ 0, & \text{otherwise,} \end{cases} \quad (2)$$

Local policy 的网络结果如图 1 上半部分所示，由一层 *Linear* 和一层 *MHA* 组成。相较于 AM，POMO 将 Positional encoding(PE) 舍弃，ELG 的 Local policy 保留了 PE，是因为 ELG 希望将距离和次序信息添加到 embedding 中，即 embedding 的计算为：

$$\mathbf{h}_i = W\mathbf{x}_i + \mathbf{b} + PE(\mathbf{x}_i) \quad (3)$$

$h_c$  类似 AM 中的 context embedding，但是因为节点使用极坐标表示，则当前节点的极坐标  $(\rho, \theta) = (0, 0)$ ，因此  $h_c$  用一个可学习的参数进行替代，则 MHA 的计算如下：

$$k_i = W^k h_i, \quad v_i = W^v h_i, \quad q = W^q h_c, \quad (4)$$

$$\tilde{h}_c = \text{MHA}([k_1, \dots, k_K], [v_1, \dots, v_K], q), \quad (5)$$

最后的概率计算如下：

$$u_{\text{local}}^i = \frac{\tilde{h}_c^T h_i}{\sqrt{d_e}}, \forall n_i \in \mathcal{N}_K(c_t) \quad (6)$$

### 3.3 Global policy

ELG 将 Global policy 与 Local policy 相结合，生成了一种 Joint policy。Joint policy 融合了 Global policy 在分布内强大的学习能力以及 Local policy 在分布外的可迁移性，以此提升泛化能力。在进行整合之前，为了进一步增强泛化效果，ELG 通过添加距离惩罚项对 Global policy 进行了调整，具体如下：Global policy 使用的是 POMO 模型，在 POMO 的基础上，ELG 添加了距离惩罚，具体的计算公式如公式 7 所示，对于邻域内的节点，Global policy 会减去一个小于等于 1 的数，对于邻域外的节点，Global policy 会减去  $\xi$ ，其中  $\xi \geq 1$ ，通过如此操作，Global policy 会优先考虑邻域内的节点，但是邻域外的节点也有被选择的可能。

$$\tilde{u}_{\text{global}}^i = \begin{cases} u_{\text{global}}^i - \frac{\rho_i}{\max\{\rho_i \mid n_i \in \mathcal{N}_K(c_t)\}}, & \text{if } n_i \in \mathcal{N}_K(c_t), \\ u_{\text{global}}^i - \xi, & \text{otherwise.} \end{cases} \quad (7)$$

### 3.4 训练与损失函数

在实际的训练过程中，ELG 首先在联合训练之前，通过引入距离惩罚对 Global policy 进行预训练，训练  $T_1$  个周期，这是因为 Global policy 的状态和动作空间比局部策略更为复杂。在联合训练阶段，ELG 采用策略梯度方法直接训练集合策略  $\pi_{\text{ens}}$ ，训练  $T_2$  个周期，其中包含全局策略的可训练参数  $\tilde{\theta}$  和局部策略的参数  $\theta$ 。参考 POMO 方法，ELG 从多个不同的起始节点进行多次 rollout，在一次前馈过程中获取多个轨迹，并使用带共享基线的 REINFORCE 算法来估计期望回报  $J$  的梯度。具体来说，多个 rollout 的平均 reward 被用作 REINFORCE Baseline，梯度  $\nabla_{\tilde{\theta}, \theta} J(\tilde{\theta}, \theta)$  通过以下公式估计：

$$\nabla_{\tilde{\theta}, \theta} J(\tilde{\theta}, \theta) = \frac{1}{N \cdot B} \sum_{i=1}^B \sum_{j=1}^N \left( R_{i,j} - \frac{1}{N} \sum_{j=1}^N R_{i,j} \right) \nabla_{\tilde{\theta}, \theta} \log \pi_{\text{ens}}(\tau_{i,j}), \quad (8)$$

## 4 复现细节

### 4.1 与已有开源代码对比

开源代码使用的环境是 Pytorch，我使用的环境是 pytorch-lightning+torchRL+Hydra，所以代码的训练框架需要更改。关于训练的代码，源码是通过一个显式的 train 函数循环多次进行训练，我复现的代码由于是基于 pytorch-lightning 框架，因此实现方式有所不同，两者的代码分别如图 2 所示，图 3。在源码中，源码通过 train 函数实现训练，train 函数通过调用实例化的 model 与 env 实现强化学习中智能体与环境的交互，通过 for 循环对训练轮数进行设置。

```
def train(model, training, T, start_steps, train_steps, mixed, train_batch_size, problem_size, distribution, multiple_width, scale_norm):
    # Initialize env
    env = TSPEEnv(multi_width=multiple_width, device=device)
    distribution_ = distribution.copy()
    gaps = np.array([1, 1, 1])
    optimizer = Optimizer(model.parameters(), lr=lr, weight_decay=1e-6)
    # REINFORCE training
    for i in trange(train_steps - start_steps + 1):
        model.train()

        # Enable joint training
        if (i == T - start_steps) and training == 'joint':
            ...

        if mixed:
            ...
        else:
            distribution_[['data_type']] = 'uniform'

        batch = generate_tsp_data(batch_size=train_batch_size, problem_size=problem_size, distribution=distribution_)
        env.load_random_problems(batch)
        reset_state, _, _ = env.reset()

        model.pre_forward(reset_state)
        solutions, probs, rewards = rollout(model=model, env=env, eval_type='sample')
        # check feasible
        check_feasible(solutions[0:1])

        optimizer.zero_grad()
        # POMO
        bl_val = rewards.mean(dim=1)[:, None] #share baseline
        log_prob = probs.log().sum(dim=1)
        advantage = rewards - bl_val #括号里的东西
        J = - advantage * log_prob
        if scale_norm:
```

图 2. ELG 源码 train 函数

在我复现的代码中，使用的是 pytorch-lightning，与 pytorch 有些不同，pytorch-lightning 通过 LightningModule 对训练过程进行控制，LightningModule 中定义了 training\_step, on\_fit\_start 等多个函数，在使用过程中，需要重写对应函数，如在 training\_step 函数中，通过调用自己编写的 play\_episode 实现智能体与环境的交互，部分代码如图 4 所示。

```

28  class REINFORCELightning(LightningModule):
147
148      def on_test_start(self) -> None:
149          self.time_estimator.reset(self.current_epoch + 1)
150
151      def training_step(self, batch, batch_idx):
152          """
153              This function is called for each batch of data. It is used to calculate the loss and update the model.
154              :param batch:
155              :param batch_idx:
156              :return: loss
157          """
158
159          if self.elg_start_step is not None and self.current_epoch == self.elg_start_step:
160              self.policy.enable_local_policy()
161
162          self.env.load_problems(batch, batch.size(0))
163          self.trained_num_episodes += self.env.env_batch_size
164
165          if self.distillation:
166              state_td, policy_out = self.play_episode(self.policy, self.env, decoder_strategy='sampling', self.first_mode)
167              self.teacher_policy.eval()
168              with torch.no_grad():
169                  self.teacher_policy.label = self.env.selected_node_list
170                  self.teacher_env.load_problems(self.env.problems, self.env.env_batch_size)
171                  teacher_state_td, teacher_policy_out = self.play_episode(self.teacher_policy, self.teacher_env, decoder_strategy='sampling', self.first_mode)
172                  loss = self.calculate_loss(policy_out, teacher_policy_out=teacher_policy_out)
173
174          else:
175              state_td, policy_out = self.play_episode(self.policy, self.env, decoder_strategy='sampling', self.first_mode)
176              loss = self.calculate_loss(policy_out)
177
178          reward = policy_out["reward"] # shape: (batch, pomo)
179          #####  

180          max_pomo_reward, _ = reward.max(dim=1) # get best results from pomo
181          score_mean = -max_pomo_reward.float().mean() # negative sign to make positive value
182
183
184

```

图 3. 对应 train 函数的复现

```

28  class REINFORCELightning(LightningModule):
300      def play_episode(self, policy, env, decoder_strategy: str = "sampling", first_mode: str = None) -> Tuple[TensorDict, dict]: 用法
325
326          reward = None
327          state_td = env.pre_step()
328          > if env.env_name == 'kp':...
329          else:
330              while not done:
331                  next_td = policy(state_td, first_mode=first_mode)
332                  prob = next_td.get("prob", None)
333                  if self.distillation:
334                      probs = next_td.get("probs", None)
335                      kd_probs = torch.cat((kd_probs, probs[:, :, :, None]), dim=-1)
336                      state_td = env.step(next_td)
337                      log_likelihood = torch.cat((log_likelihood, prob[:, :, None]), dim=2)
338                      reward = state_td["reward"]
339                      done = state_td["done"].all()
340
341
342          policy_out = {
343              "reward": reward,
344              "log_likelihood": log_likelihood,
345              "kd_probs": kd_probs
346          }
347
348          return state_td, policy_out
349
350
351
352
353
354
355
356
357
358
359

```

图 4. play\_episode

完成 LightningModule 后，需要通过 Lightning 中 Trainer 的 fit 函数进行训练，如图 5 所示，它会按照预先设定的训练参数（如 epochs、batch size 等），在训练数据集上迭代地进行前向传播、计算损失、反向传播和参数更新等操作。

```

class RLTrainer(Trainer):

    def fit(
        self,
        model: "pl.LightningModule",
        train_dataloaders: Optional[Union[TRAIN_DATALOADERS, LightningDataModule]] = None,
        val_dataloaders: Optional[EVAL_DATALOADERS] = None,
        datamodule: Optional[LightningDataModule] = None,
        ckpt_path: Optional[str] = None,
    ) -> None:
        """
        We override the 'fit' method to automatically apply and handle RL4CO magic
        to 'self.automatic_optimization = False' models, such as PPO
        """

        if not model.automatic_optimization:
            if self.gradient_clip_val is not None:
                log.warning(
                    "Overriding gradient_clip_val to None for 'automatic_optimization=False' models"
                )
            self.gradient_clip_val = None

        super().fit(
            model=model,
            train_dataloaders=train_dataloaders,
            val_dataloaders=val_dataloaders,
            datamodule=datamodule,
            ckpt_path=ckpt_path,
        )

```

图 5. 复现代码中的 Trainer

## 4.2 实验环境搭建

实验所需要使用到的库如表 1 所示，使用单张 NVIDIA V100 32GB Tensor Core GPU 进行训练。

表 1. 实验中使用的库

库	版本
numpy	1.24.4
scipy	1.10.1
pyrootutils	1.0.4
hydra-core	1.1.1
lightning	2.1.0
pyyaml	6.0.1
tensorboard	2.14.0
tensorboard-data-server	0.7.2
torchrl	0.1.1
tensordict	0.1.2

### 4.3 创新点

在改进方面，参考了 ICLR 2025 openreview 上的 ReLD [1]，目前常用的网络结构可以根据 Encoder 与 Decoder 层数的不同划分成三种，如图 6 所示。ReLD 认为，像 POMO 这种 Heavy Encoder Light Decoder 的结构，虽然能将问题信息充分 embedding，但是当问题规模增大时，embedding 的信息密度过大，Light Decoder 的结构无法充分利用 embedding 的信息，而 Light Decoder Heavy Decoder 的结构能根据下一步的子问题进行重新 embedding，降低了规模增大带来的复杂度，为了使这两种网络结构性能接近，需要在 Decoder 上进行修改。

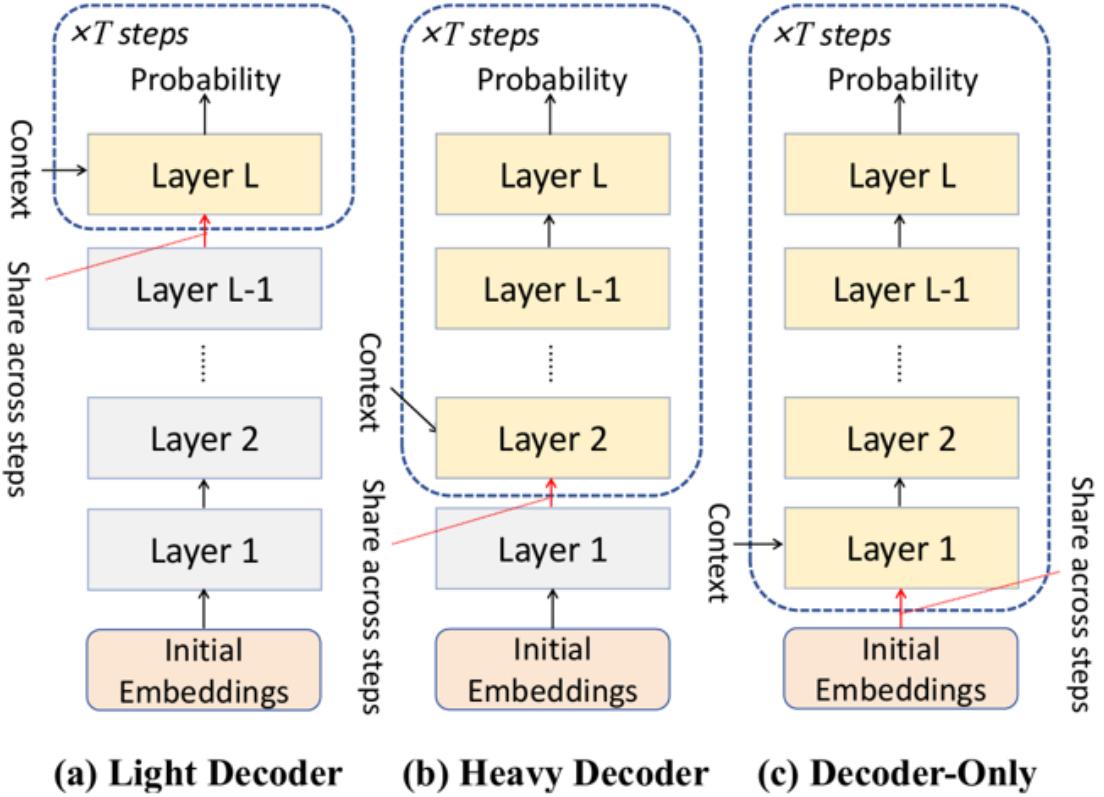


图 6. 三种网络结构

在 POMO 中，Decoder 采用 *MHA* 层和 *Compatibility* 层，如公式 11 所示，embedding  $h_c$  是间接作用于 *Compatibility* 层，无法充分利用动态信息，为了改变这个问题，可以使用残差链接进行修改，此外，*MHA* 中的 Q,K,V 矩阵其实是通过 *Linear* 层实现，通过线性表示在表示能力上也有一定的限制，为了改变这个问题，使用 *FeedForward* 层增加非线性表达信息。

$$h_c = [h_{\tau_{t-1}}, \mathcal{D}_t] \quad (9)$$

$$h'_c = \text{MHA}(h_c, H_t, H_t). \quad (10)$$

$$p_i = \text{Softmax} \left( C \cdot \tanh \left( \frac{(h'_c)^T H_t}{\sqrt{d_h}} \right) \right)_i \quad (11)$$

修改前后的 Decoder 结果如图 7 所示。在 *MHA* 层和 *Compatibility* 层中间添加 *FeedForward* 层以增加非线性表达，并使用残差链接充分利用 embedding 信息。

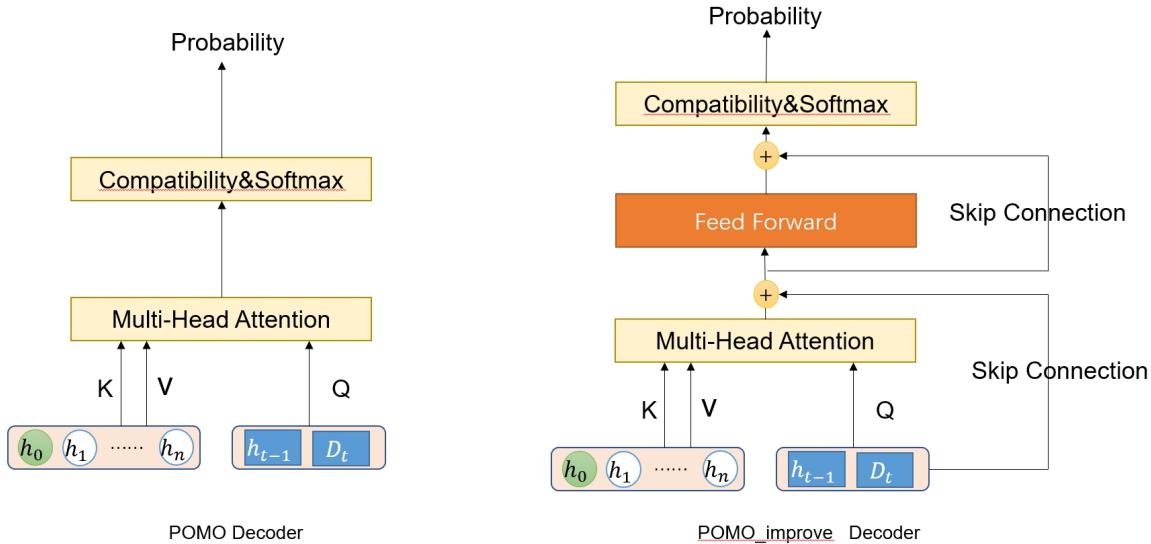


图 7. Decoder 对比

## 5 实验结果分析

### 5.1 数据集介绍

本次实验所采用的数据集为 TSPLIB 和 CVRPLIB 这两个数据集。依据问题规模的差异，CVRPLIB 被进一步细分为 Set-X 与 Set-XXL。具体细节可参见表 2 以及表 3。TSPLIB 是由一系列规模小于 1000 的 TSP 问题所构成，涵盖了丰富多样的情形。而 CVRPLIB Set-X 是由一系列规模小于 1000 的 CVRP 问题组合而成，CVRPLIB Set-XXL 则着重考虑了大规模的问题，其中涵盖了 3000、4000、6000、7000 这四种规模的 CVRP 问题，为实验提供了更全面的问题样本和研究视角。

表 2. TSPLIB 和 CVRPLIB Set-X

Dataset	$N \leq 200$	$200 < N \leq 500$	$500 < N \leq 1000$	Total
TSPLIB	29 instances	13 instances	6 instances	48 instances
CVRPLIB Set-X	22 instances	46 instances	32 instances	100 instances

表 3. CVRPLIB Set-XXL

CVRPLIB Set-XXL	A1	A2	L1	L2
Problem_size	6000	7000	3000	4000

## 5.2 实验结果可视化

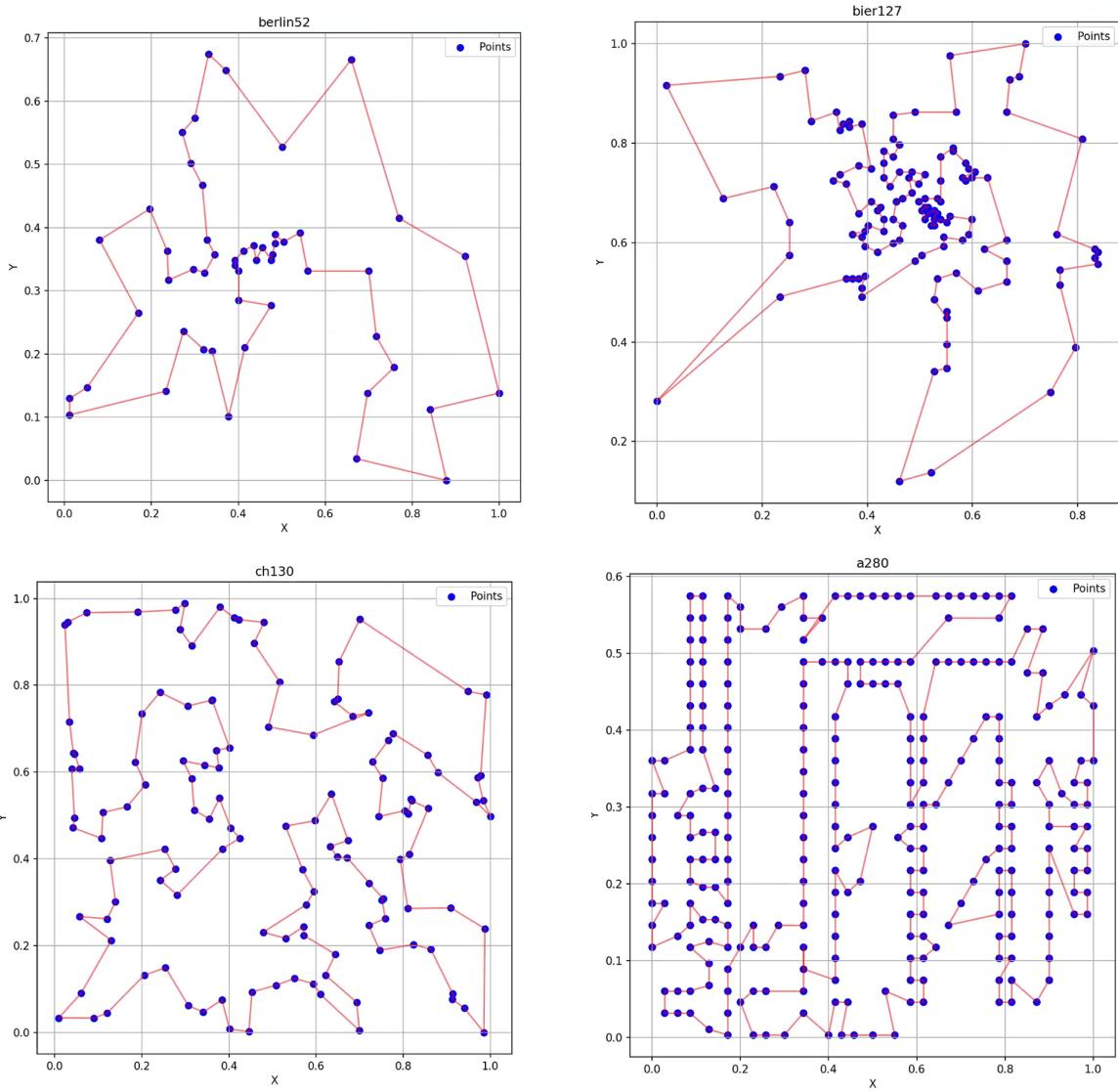


图 8. TSPLIB 部分求解方案可视化

如图 8 所示是 TSPLIB 部分求解方案的可视化结果，红色线条表示的求解路径，总体来看，随着问题规模的增加，路径的复杂性和求解难度也在增加。求解算法在小规模问题上表现较好，能够找到较为平滑的路径。

图 9 和图 10 所示是 CVRPLIB Set-X 和 CVRPLIB Set-XXL 部分求解方案的可视化结果，每一次回到仓库节点就用另外一种颜色的线进行绘制，对于节点数量较少，分布规则的数据集，路径相对较为简单。而随着节点数量的增加，路径变得更加复杂，出现了更多的弯曲和转折，这反映了在处理大规模问题时，找到一条高效的路径需要更复杂的算法和策略。

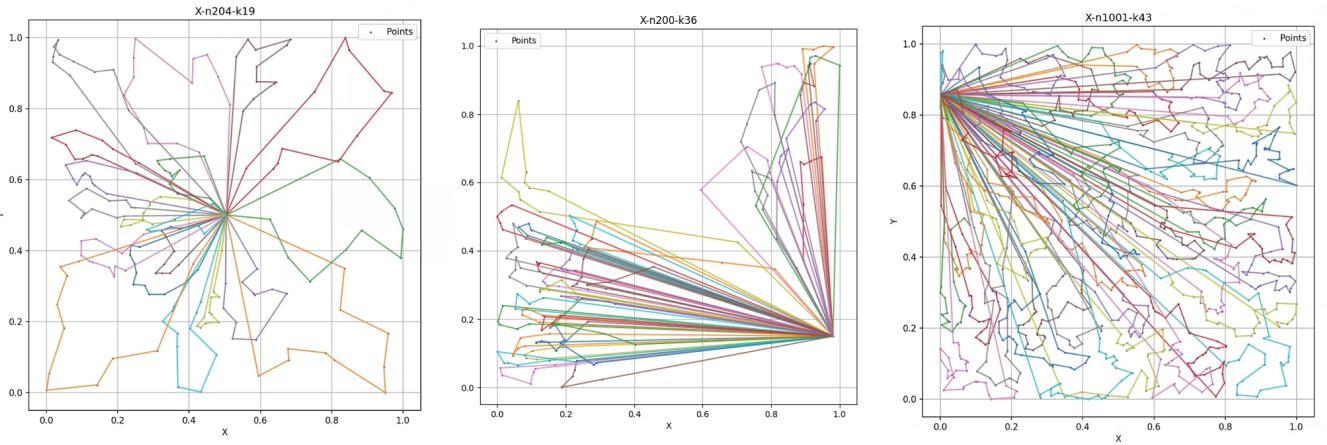


图 9. CVRPLIB Set-X 部分求解方案可视化

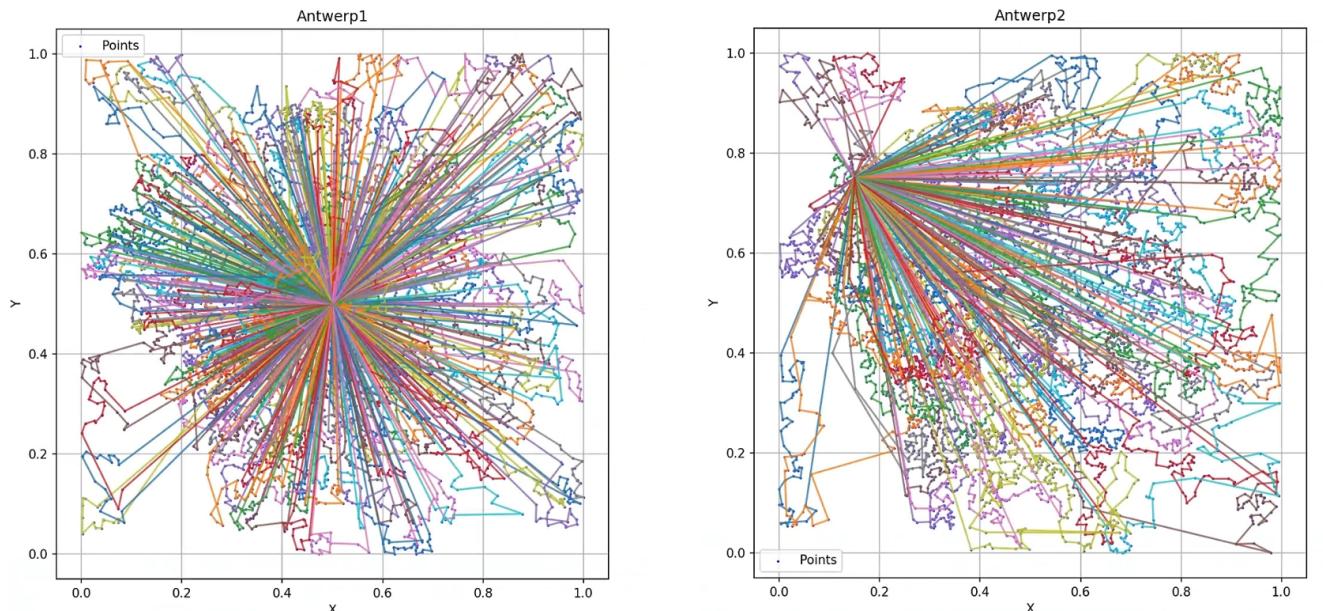


图 10. CVRPLIB Set-XXL 部分求解方案可视化

### 5.3 实验结果

实验结果如表4,5,6所示，可以看到在规模较小的 TSP 和 CVRP 问题上，改进后的 ELG 结果虽然有所提升，但是提升效果有限，原因可能是在规模较小的问题中，embedding 的信息密度较低，原本的 Decoder 结构已经能够较好地进行解码。在规模较大时，可以看到改进后的 ELG 效果有显著提升，提升的百分点基本在 5 以上，说明改进后的 Decoder 在解码 embedding 时效果确实更好。

表 4. 在 TSPLIB 上的实验结果

Method	$N \leq 200$	$200 < N \leq 500$	$500 < N \leq 1000$	Total
POMO	2.02%	15.25%	31.68%	9.31%
ELG	1.18%	4.34%	8.91%	3.12%
ELG-improve	<b>1.03%</b>	<b>4.33%</b>	<b>8.04%</b>	<b>2.90%</b>

表 5. 在 CVRPLIB Set-X 上的实验结果

Method	$N \leq 200$	$200 < N \leq 500$	$500 < N \leq 1000$	Total
POMO	6.90%	15.04%	40.81%	21.49%
ELG	4.51%	5.52%	7.80%	6.03%
ELG-improve	<b>3.85%</b>	<b>5.33%</b>	<b>7.06%</b>	<b>5.56%</b>

表 6. 在 CVRPLIB Set-XXL 上的实验结果

Method	A1	A2	L1	L2
POMO	112.27%	159.22%	75.30%	78.16%
ELG	14.04%	25.52%	16.45%	23.26%
ELG-improve	<b>9.01%</b>	<b>16.85%</b>	<b>10.00%</b>	<b>18.07%</b>

## 6 总结与展望

在本次复现工作中，为了提升 VRP 求解器的实用性，深入研究与复现一种集成 Global policy 和 Local policy 的方法——ELG，取得了一定的成果。在方法层面，ELG 创新性地结合了 POMO 作为 Global policy 关注全局信息与自行设计的 Local policy 聚焦当前节点邻域信息的优势，并通过距离惩罚项等手段优化了 Global policy，同时在训练过程中采用合理的预训练和联合训练策略梯度方法，有效提升了模型的泛化能力，然而原本的 ELG 的网络结构仍有不足，具体表现在大规模问题上，为了解决这个问题，引入残差连接和前馈层中的非线性表达。在实验方面，选用 TSPLIB 和 CVRPLIB 数据集进行测试，实验结果表明，改进后的 ELG 在大规模问题上表现出显著优势，相比原始方法在结果提升上可达 5 个百分点以上，证明了对 Decoder 结构改进的有效性，而在小规模问题上虽提升有限，但也验证了方法的可行性。

然而，研究过程中仍存在一些不足。在训练效率方面，尽管改进后 ELG 取得了较好的结果，但训练时间较长，尤其是在处理大规模数据集和复杂网络结构时，计算资源的消耗较为严重。这可能会限制其在一些对实时性要求较高的实际应用场景中的应用。在模型复杂度上，对 Decoder 结构进行了改进，但整体模型仍然相对复杂。针对这些不足，未来的研究可以从以下几个方向展开。在算法优化方面，可以探索更高效的训练算法和优化策略，或者轻量化网络结构。

## 参考文献

- [1] Rethinking light decoder-based solvers for vehicle routing problems.
- [2] Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [3] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.
- [4] Dimitri P Bertsekas. Dynamic programming and optimal control. belmont. *MA: Athena Scientific*, 2000.
- [5] Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.
- [6] Hanni Cheng, Haosi Zheng, Ya Cong, Weihao Jiang, and Shiliang Pu. Select and optimize: Learning to solve large-scale tsp instances. In *International Conference on Artificial Intelligence and Statistics*, pages 1219–1231. PMLR, 2023.
- [7] Wu Deng, Junjie Xu, and Huimin Zhao. An improved ant colony optimization algorithm based on hybrid strategies for scheduling problem. *IEEE access*, 7:20281–20292, 2019.
- [8] Darko Drakulic, Sofia Michel, Florian Mai, Arnaud Sors, and Jean-Marc Andreoli. Bq-nco: Bisimulation quotienting for generalizable neural combinatorial optimization. *arXiv preprint arXiv:2301.03313*, 2023.
- [9] Paola Festa. A brief introduction to exact, approximation, and heuristic algorithms for solving hard combinatorial optimization problems. In *2014 16th International Conference on Transparent Optical Networks (ICTON)*, pages 1–20. IEEE, 2014.
- [10] A Hanif Halim and IJAoCMiE Ismail. Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem. *Archives of Computational Methods in Engineering*, 26:367–380, 2019.
- [11] Dorit S Hochba. Approximation algorithms for np-hard problems. *ACM Sigact News*, 28(2):40–52, 1997.
- [12] Benjamin Hudson, Qingbiao Li, Matthew Malencia, and Amanda Prorok. Graph neural network guided local search for the traveling salesperson problem. *arXiv preprint arXiv:2110.05291*, 2021.
- [13] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

- [14] Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. Deep policy dynamic programming for vehicle routing problems. In *International conference on integration of constraint programming, artificial intelligence, and operations research*, pages 190–213. Springer, 2022.
- [15] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.
- [16] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International conference on learning representations*, 2019.
- [17] Fu Luo, Xi Lin, Fei Liu, Qingfu Zhang, and Zhenkun Wang. Neural combinatorial optimization with heavy decoder: Toward large scale generalization. *Advances in Neural Information Processing Systems*, 36:8845–8864, 2023.
- [18] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.
- [19] Zhiqing Sun and Yiming Yang. Difusco: Graph-based diffusion solvers for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36:3706–3731, 2023.
- [20] Eu Jin Teoh, Huajin Tang, and Kay Chen Tan. A columnar competitive model with simulated annealing for solving combinatorial optimization problems. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 3254–3259. IEEE, 2006.
- [21] Vijay V Vazirani and Vijay V Vazirani. Introduction to lp-duality. *Approximation Algorithms*, pages 93–107, 2003.
- [22] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.
- [23] J Wesley Barnes and MANUEL LAGUNA. Solving the multiple-machine weighted flow time problem using tabu search. *IIE transactions*, 25(2):121–128, 1993.
- [24] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.
- [25] Yiwen Zhong, Juan Lin, Lijin Wang, and Hui Zhang. Discrete comprehensive learning particle swarm optimization algorithm with metropolis acceptance criterion for traveling salesman problem. *Swarm and Evolutionary Computation*, 42:77–88, 2018.