A GAME-THEORETIC APPROACH FOR IMPROVING GENERALIZATION ABILITY OF TSP SOLVERS

Anonymous authors
Paper under double-blind review

ABSTRACT

In this paper, we introduce a two-player zero-sum framework between a trainable *Solver* and a *Data Generator* to improve the generalization ability of deep learning-based solvers for Traveling Salesman Problems (TSP). Grounded in *Policy Space Response Oracle* (PSRO) methods, our two-player framework outputs a population of best-responding Solvers, over which we can mix and output a combined model that achieves the least exploitability against the Generator, and thereby the most generalizable performance on different TSP tasks. We conduct experiments on a variety of TSP instances with different types and sizes. Results suggest that our Solvers achieve the state-of-the-art performance even on tasks the Solver never meets, whilst the performance of other deep learning-based Solvers drops sharply due to over-fitting. To demonstrate the principle of our framework, we study the learning outcome of the proposed two-player game and demonstrate that the exploitability of the Solver population decreases during training, and it eventually approximates the Nash equilibrium along with the Generator.

1 INTRODUCTION

Deep learning for solving combinatorial optimization problems has recently attracted enormous attention due to its ability to capture complex improvement heuristics from training over millions of problem instances (Dai et al., 2017). Additionally, due to the efficiency of the forward computation of neural networks, deep learning based techniques are particularly efficient in comparison to traditional methods when performing inference on large-scale problems. As a consequence, it is promising direction to study training deep learning-based solvers offline and later to implement solvers online.

The generalization ability of a solver concerns its performance on a variety of different data distributions. Most previous works only train their models on data from uniform distribution, however, overfitting to the uniform distribution can cause poor generalization ability. As a direct intuition, improvement on the generalization ability can be obtained by exploring the data distribution where the model performs poorly. In this work, we tackle the generalization problem by introducing a novel two-player game framework: player one, the trainable Solver, aims to train solvers to perform well on distributions chosen by player two, and player two, the Data Generator, aims to generate distributions in which Solver can be challenged. With regard to the policy space of Data Generator, it is possible that Data Generator has an infinite-sized number of choices since there are an enormous amount of candidate distributions. In this view, previous work focuses solely on adopting the uniform distribution, which unfortunately is only one of the many policies available to player two.

Without loss of generality, we take the typical problem in combinatorial optimisation—Travelling Salesman Problem (TSP)—which is widely used for many real-world applications. For our Solvers we utilise a standard deep learning-based solver as our base solver. Our framework is solver agnostic, it can be applied to improve the generalization performance of any existing solvers. For the Data Generator, we develop a learning-to-attacking technique under the two-player framework by adding perturbations on top of uniformly generated data where the induced task instance can challenge the current Solver and make it perform poorly; as such, the generator can learn to find the weaknesses of the Solver [1].

Overall, our contributions are as follows:

[1] See how easily the current TSP solvers can be exploited in Appendix A.5.

- We study the generalization ability of TSP solvers from a game-theoretical perspective, and propose a two-player game framework to train effective Solvers which can incorporate, with minimal changes, any exsiting deep learning-based Solver.
- We propose a learning-to-attack method by adding *learnable* perturbations on the data distribution of problem instance so that the solver can be exploited to perform poorly.
- We introduce a mixing-model by combining the population of Solvers so that we can make full use of the obtained solver population to attain the state-of-the-art results.
- We study the exploitability of obtained strategies during training to offer insights about how the solver population develop over time. The results show that the obtained strategies under our framework are asymptotically approximating the Nash Equilibrium.

2 Related Work

Deep Learning for Combinatorial Optimization. Deep learning or reinforcement learning based methods have achieved notable progress in various combinatorial optimization problems, such as Traveling Salesman Problem (Vaswani et al., 2017; Lu et al., 2019; Kool et al., 2018; 2021), Capacitated Vehicle Routing Problem (Hottung et al., 2020; Wu et al., 2021), graph matching problems (Yu et al., 2019; 2021). However, these works only focus on the data from uniform distribution, which ignore the generalization ability to unseen instances.

Meta-Game Analysis. In meta-game analysis (Wellman, 2006; Yang & Wang, 2020), traditional solution concepts (e.g., Nash equilibrium) can be computed in a more scalable manner. PSRO (Lanctot et al., 2017) generalises Double Oracle (McMahan et al., 2003) by introducing RL to obtain an approximate best response. In games with high degree of non-transitivity Czarnecki et al. (2020) such as Chess (Sanjaya et al., 2021), PSRO methods prove to be an efficient approach to prevent from learning strategic cycles. PSRO$_{rN}$ (Balduzzi et al., 2019) and Diverse-PSRO (Nieves et al., 2021; Liu et al., 2021) incorporated diversity seeking into PSRO and Pipeline-PSRO (McAleer et al., 2020) aims to improve training efficiency by training multiple best responses in parallel.

3 Notations and Preliminaries

Normal Form Game (NFG) - A tuple (Π, U^{Π}, n) where n is the number of players, $\Pi = (\Pi_1, \Pi_2, ..., \Pi_n)$ is the joint policy set and $U^{\Pi} = (U_1^{\Pi}, U_2^{\Pi}, ...U_n^{\Pi}) : \Pi \to \mathcal{R}^n$ is the utility matrix for each joint policy. A game is symmetric if all players have the same policy set ($\Pi_i = \Pi_j, i \neq j$) and same payoff structures, such that players are interchangeable.

Best Response - The strategy which attains the best expected performance against a fixed opponent strategy. $\sigma_i^* = \text{br}(\Pi_{-i}, \sigma_{-i})$ is the best response to σ_{-i} if:

$$U_i^{\Pi}(\sigma_i^*, \sigma_{-i}) \geq U_i^{\Pi}(\sigma_i, \sigma_{-i}), \forall i, \sigma_i \neq \sigma_i^*$$

Nash Equilibrium - A strategy profile $\sigma^* = (\sigma_1^*, \sigma_2^*, ..., \sigma_n^*)$ such that:

$$U_i^{\Pi}(\sigma_i^*, \sigma_{-i}^*) \geq U_i^{\Pi}(\sigma_i, \sigma_{-i}^*), \forall i, \sigma_i \neq \sigma_i^*$$

Intuitively, no player has an incentive to deviate from their current strategy if all players are playing their respective Nash equilibrium strategy.

Exploitability - In a two-player zero-sum game, the average exploitability of a strategy profile $\sigma = (\sigma_1, \sigma_2)$ is defined as follows (Davis et al., 2014):

$$\text{exploitability}(\sigma) = \frac{1}{2}\big(U_1^{\Pi}(\text{br}(\sigma_2), \sigma_2\big) + U_2^{\Pi}\big(\sigma_1, \text{br}(\sigma_1)\big) \tag{1}$$

Instance - An individual sample of a combinatorial optimization problem. Hereafter, we denote an instance by $\mathcal{I}$ which comes from some distribution $\mathbf{P}_{\mathcal{I}}$.

Optimal gap - Measures the quality of a Solver compared to an optimal Oracle. Given an Instance $\mathcal{I}$ and a Solver $S : \{\mathcal{I}\} \to \mathbf{R}$, the optimal gap is defined as:

$$g(S, \mathcal{I}, \text{Oracle}) = \frac{S(\mathcal{I}) - \text{Oracle}(\mathcal{I})}{\text{Oracle}(\mathcal{I})} \tag{2}$$

Algorithm 1 Two-Player Framework for Combinatorial Optimization

Input: Initial joint policy sets for Solver Selector and Data Generator as Π. Compute utilities U^{Π} for joint $\pi \in \Pi$. Initialize meta-strategies $\sigma_i = \text{UNIFORM}(\Pi_i)$
while epoch e in $\{1, 2, ...\}$ **do**
  Construct mixing distribution $\pi_{\text{mix}} = \sum_i \sigma^i_{\text{DG}} \pi^i_{\text{DG}}$ and train the oracle for Solver Selector $S^{'}$ with gradient in Eq. 6
  for many episodes **do**
    Sample $S \sim \Pi_{\text{SS}}$ according to σ_{SS}
    Train Oracle $\mathbf{P}^{'}_{\mathcal{I}} = \text{br}(S)$ with gradient in Eq. 11
  end for
  Update policy set:$\Pi \leftarrow \Pi \cup \{(S^{'}, \mathbf{P}^{'}_{\mathcal{I}})\}$
  Compute missing entries in U^{Π} from Π and the meta-strategy σ from U^{Π}
end while
Output meta-strategy σ_{SS} and policy set Π_{SS} to obtain mixing model by Eq. 12 or Eq. 13.

where Oracle($\mathcal{I}$) gives the true optimal value of the Instance. Furthermore, the *expected* optimal gap of an Instance distribution $\mathbf{P}_{\mathcal{I}}$ and an Oracle is defined as:

$$G(S, \mathbf{P}_{\mathcal{I}}, \text{Oracle}) = \mathbf{E}_{\mathcal{I} \sim \mathbf{P}_{\mathcal{I}}} g(S, \mathcal{I}, \text{Oracle}). \tag{3}$$

4 Our Method

In this section, we present our method for solving the Traveling Salesman Problem (TSP) at the meta-level. Let there be two players in the meta-game, one is the Solver Selector (SS) and the other is the Data Generator (DG). $\Pi_{\text{SS}} = \{S_i | i = 1, 2, ...\}$ is the policy set of candidate Solvers for the Solver Selector, and $\Pi_{\text{DG}} = \Pi_N \times \Pi_C = \{\mathbf{P}_{\mathcal{I},i} = (\mathbf{P}_{N,i}, \mathbf{P}_{C,i}) | i = 1, 2, ...\}$ is the policy set for the Data Generator, where Π_N is the policy set of the problem scale (i.e. the number of nodes that need to be generated) and Π_C is the policy set used to generate two-dimensional coordinates. Therefore, an Instance distribution $\mathbf{P}_{\mathcal{I}} \in \Pi_{\text{DG}}$ comprises of two parts: $\mathbf{P}_{\mathcal{I}} = (\mathbf{P}_N, \mathbf{P}_C)$ where $\mathbf{P}_N$ is the distribution for the number of nodes N contained in each Instance, and $\mathbf{P}_C$ is a two-dimensional distribution for coordinate positioning.

Formally, we have a two player zero-sum asymmetric NFG $(\Pi, \mathbf{U}^{\Pi}, 2)$ where $\Pi = (\Pi_{\text{SS}}, \Pi_{\text{DG}})$, $\mathbf{U}^{\Pi} : \Pi \rightarrow \mathbf{R}^{|\Pi_{\text{SS}}| \times |\Pi_{\text{DG}}|}$, $\mathbf{U}^{\Pi_{\text{SS}}}(\pi) = G(\pi, \text{Oracle})$ is the expected optimal gap under the joint policy $\pi = (S, \mathbf{P}_{\mathcal{I}}) \in \Pi$ as defined in Eq. 3 and $\mathbf{U}^{\Pi_{\text{DG}}}(\pi) = -\mathbf{U}^{\Pi_{\text{SS}}}(\pi)$. Given $\mathbf{U}^{\Pi}$, we can determine a Nash Equilibrium $\sigma^* = (\sigma^*_{\text{SS}}, \sigma^*_{\text{DG}})$ as the meta-strategy which satisfies:

$$\min_{\sigma_{\text{SS}} \in \Delta(\Pi_{\text{SS}})} \max_{\sigma_{\text{DG}} \in \Delta(\Pi_{\text{DG}})} \text{E}_{\pi \sim (\sigma_{\text{SS}}, \sigma_{\text{DG}})} G\big(\pi, \text{Oracle}\big). \tag{4}$$

We follow the PSRO framework as follows: at each iteration given the policy sets Π_{SS} and Π_{DG} and the meta-strategy $\sigma = (\sigma_{\text{SS}}, \sigma_{\text{DG}})$ we train two Oracles:

- $S^{'}$ represents the Solver Selector,a best response to the Data Generator's meta strategy σ_{DG}.
- $\mathbf{P}^{'}_{\mathcal{I}}$ is the Data Generator which learns a data distribution where σ_{SS} performs poorly.

Given these oracles, we update the joint policy set $\Pi^{'} = \Pi \cup (S^{'}, \mathbf{P}^{'}_{\mathcal{I}})$ and the meta-game $\mathbf{U}^{\Pi^{'}}$ according to $\Pi^{'}$. This expansion of the joint policy set has a dual purpose in that it aids in finding the difficult to find Instance distributions whilst also improving the ability of the Solver Selector to solve said distributions. In line with our objective, this process leads to a population of powerful Solvers which have diverse abilities on various distributions, and how we successfully combine these individual Solvers is discussed in Sec. 4.4. The general algorithm framework can be seen in Alg. 1.

The formulation above leaves us with four algorithm components to address: **1)** How to obtain the meta-strategy σ; **2)** How to train Oracles for both players; **3)** How to evaluate the utilities U^{Π}; **4)** How to combine the the population of Solvers. In the following parts, we will describe the flow of the algorithm as visualised in Fig. 1, where each section shown in purple represents the answer to the above four questions.

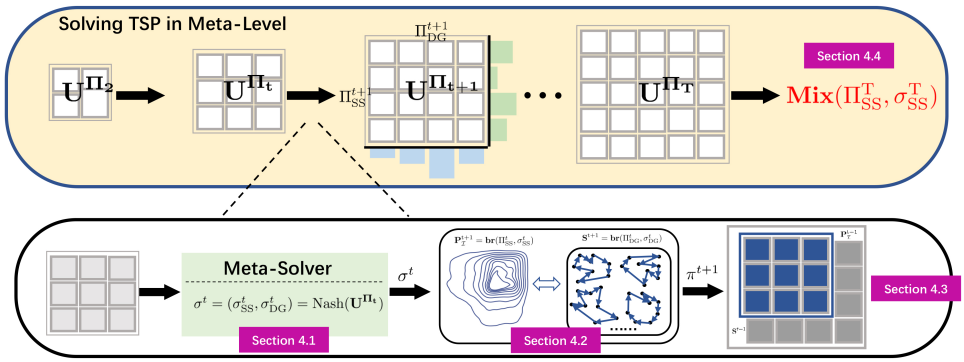


Figure 1: Pipeline of solving combinatorial optimization problems in meta-level. At PSRO loop t, we first use meta-Solver to compute the meta-strategy σ_t given the meta-table $\mathbf{U}^{\Pi_t}$ and then training best response $(\mathbf{S}^{t+1}, \mathbf{P}_{\mathcal{I}}^{t+1})$ based on current policy set and meta-strategy (Π^t, σ^t). Finally we get a new meta-table $\mathbf{U}^{\Pi_{t+1}}$ according to the new obtained policy and algorithm process iterates to the next loop.

4.1 Meta-Strategy Solvers

In this paper, we use the NE of the meta game as the meta-strategy solver as we believe for a two player zero-sum game the NE is sufficient, but various meta-strategy solvers can be used w.r.t the corresponding meta-game constructed by the specific combinatorial optimization problem.

4.2 Oracle Training

We now provide the higher-level details for training a best-response Oracle in the TSP setting, with more detailed derivations presented in Appendix A.1. Here we represent the trainable Solvers in Π_{SS} as S_θ and the trainable Instance distributions in Π_{DG} as $\mathbf{P}_{\mathcal{I},\gamma} = (\mathbf{P}_{N,\gamma_N}, \mathbf{P}_{C,\gamma_C})$ where θ and γ are the trainable parameters.

Solver Oracle. Given the Data Generator meta-strategy σ_{DG}, the Oracle training objective for the Solver is:

$$\min_\theta L_{\text{SS}}(\theta) = \mathbf{E}_{\mathbf{P}_{\mathcal{I}} \sim \sigma_{\text{DG}}} G(S_\theta, \mathbf{P}_{\mathcal{I}}, \text{Oracle}). \tag{5}$$

The gradient of this objective is:

$$\nabla_\theta L_{\text{SS}}(\theta) \quad = \mathbf{E}_{\mathbf{P}_{\mathcal{I}} \sim \sigma_{\text{DG}}} \mathbf{E}_{N \sim \mathbf{P}_N} \mathbf{E}_{x_1,\ldots,x_N \sim \prod_{i=1}^N \mathbf{P}_C} \frac{\nabla_\theta S_\theta(x_1, \ldots, x_N)}{\text{Oracle}(x_1, \ldots, x_N)}.^2 \tag{6}$$

Data Generator Oracle. Given the Solver Selector meta-strategy σ_{SS}, the Oracle training objective for the Data Generator is:

$$\max L_{\text{DG}}(\gamma) = \mathbf{E}_{S \sim \sigma_{\text{SS}}} G(S, \mathbf{P}_{\mathcal{I},\gamma}, \text{Oracle}). \tag{7}$$

To derive the gradient for the Data Generator Oracle, we first note the following: For Π_N, we fix the problem scale $\mathcal{N} = \{N_1, N_2, \ldots\}$, and let $\mathbf{P}_{N,\gamma_N} \in \Pi_N$ be a parameterized discrete distribution over $\mathcal{N}$. Here we directly use a learnable probability vector γ_N (implement by a softmax function) to represent $\mathbf{P}_{N,\gamma_N}$.

As the goal of the Data Generator is to find a suitable distribution that the current Solver finds difficult to solve, we design an approach that directly attacks the Solver by adding noise to given instances[3]. We achieve this by utilising an *attacked* distribution where Instances sampled from a uniform distribution are perturbed by Gaussian noise. Formally, we start by sampling $\mathcal{I} \sim \text{U}(0, 1)$, then we use a neural network parameterised attack generator f_{γ_C} to generate the variance of a Gaussian distribution:

$$\Sigma = f_{\gamma_C}(\mathcal{I}) \tag{8}$$

where the shape of Σ is the same as $\mathcal{I}$, that is, if $\mathcal{I}$ contains N two dimension coordinates then the variance matrix will be $\Sigma \in \text{R}^{N \times 2}$, and we attack an Instance $\mathcal{I}$ additively via $\tilde{\mathcal{I}}_{i,j} = \mathcal{I}_{i,j} + \epsilon$ where

[2]We omit the calling of 'Oracle' in the implementation because the goal of the two objectives are coincident.

[3]Demonstrations of these attacks are shown in Appendix A.5

$\epsilon \sim \mathrm{N}(0, \Sigma_{i,j})$. We denote the final attacked distribution by $\mathbf{P}_{C,\gamma_C}$, and our objective therefore is to find the optimal parameters $\gamma^* = (\gamma_C^*, \gamma_N^*)$.

The gradient w.r.t. γ_C is:

$$\nabla_{\gamma_C} L_{\text{DG}}(\gamma) = \mathbf{E}_{S\sim\sigma_{\text{SS}}} \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^{N}\mathbf{P}_{C,\gamma_C}} \Big[\nabla_{\gamma_C} \big(\sum_{i=1}^{N} \log \mathbf{P}_{C,\gamma_C}(x_i) \big) g\big(S, (x_1, ...x_N), \text{Oracle}\big) \Big]. \tag{9}$$

Details relating to the extra computation required for the log-probability in Eq. 9 is left to Appendix A.2.

The gradient w.r.t. γ_N is:

$$\nabla_{\gamma_N} L_{\text{DG}}(\gamma) = \mathbf{E}_{S\sim\sigma_{\text{SS}}} \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \nabla_{\gamma_N} (\log \mathbf{P}_{N,\gamma_N}(N)) \cdot \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^{N}\mathbf{P}_{C,\gamma_C}} g\big(S, (x_1, ...x_N\big), \text{Oracle}). \tag{10}$$ [4]

Overall, we have that the gradient of the Data Generator Oracle is:

$$\nabla_{\gamma} L_{\text{DG}}(\gamma) = \begin{pmatrix} \nabla_{\gamma_C} L_{\text{DG}}(\gamma) \\ \nabla_{\gamma_N} L_{\text{DG}}(\gamma) \end{pmatrix} \tag{11}$$

4.3 Evaluation

Given the joint policy set Π, we can compute the elements in matrix U^{Π} by approximating the expected optimal gap defined in Eq. 3:

$$u_{S,\mathbf{P}_{\mathcal{I}}} = G(S, \mathbf{P}_{\mathcal{I}}, \text{Oracle}) = \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} g(S, \mathcal{I}, \text{Oracle}) \approx \frac{1}{M} \sum_{i=1}^{M} g(S, \mathcal{I}_i, \text{Oracle}).$$

where $S \in \Pi_{SS}, \mathbf{P}_{\mathcal{I}} \in \Pi_{DG}$.

4.4 Combining the Solver population

At the end of training we are left with a diverse population of Solvers designed for different distributions, which we suspect may be combined to generate a powerful general Solver. There already exist several works on how to mix policies, such as Q-Mixing (Smith et al., 2020; 2021), however we instead choose to combine the Solvers based on the meta-strategy. As we use the Nash equilibrium as our meta-solver, we can guarantee that our combined Solver has a given *conservative* level of performance under the assumption that these Instance distributions can be generated by the Data Generator's policy set. To some extent, we suggest that the conservative nature of the Nash equilibrium is itself in accordance with the meaning of generalization ability. Additionally, in contrast to Q-mixing which only supports value-based methods, our mixing-method makes no prior assumptions on the type of RL algorithm in use, and can be used for both value-based and policy-based methods.

Formally, for value-based RL methods, we can weight the corresponding Q values to get the mixed Q-value of the combined model:

$$Q_{\text{mix}}(s, a) = \sum_{\pi\in\Pi_{\text{SS}}} \sigma_{\text{SS}}^*(\pi) Q_{\pi}(s, a) \tag{12}$$

For policy-based RL methods, we directly obtain the mixed policy probability by:

$$\pi_{\text{mix}}(a|s) = \sum_{\pi\in\Pi_{\text{SS}}} \sigma_{\text{SS}}^*(\pi) \pi(a|s) \tag{13}$$

5 Experiments

In the following section, we present our results on TSP Instances of size $n = 20, 50, 100$. In contrast to previous work which fail to show generalisation ability due to training and testing on the same distribution (uniform), we demonstrate performance on distributions that are unseen during training. We detail the basic settings below, with specific training settings listed in Appendix A.3.

[4] We omit the calling of 'Oracle' in the implementation because the goal of the two objectives are coincident.

Table 1: Our model vs baselines. The gap % is w.r.t. the best value across all methods.

	Method	$n = 20$ Obj.	Gap	Time	$n = 50$ Obj.	Gap	Time	$n = 100$ Obj.	Gap	Time
TSP	Concorde	3.43	0.00%	(6s)	4.99	0.00%	(1m)	6.20	0.00%	(3m)
	LKH3	3.43	0.00%	(2s)	4.99	0.00%	(27s)	6.20	0.00%	(3m)
	Gurobi	3.43	0.00%	(1s)	4.99	0.00%	(19s)	6.20	0.00%	(4m)
	AM(gr.)	3.45	0.58%	(2s)	5.12	2.61%	(4s)	6.71	8.23%	(8s)
	LIH(T=1000)	3.69	7.72%	(16s)	5.07	1.72%	(33s)	6.72	8.48%	(63s)
	LIH(FS)(T=1000)	**3.43**	**0.12%**	(18s)	**5.06**	**1.68%**	(34s)	**6.47**	**4.43%**	(67s)
	LIH(FT)(T=1000)	**3.43**	**0.04%**	(50s)	**5.07**	**1.70%**	(64s)	**6.45**	**4.00%**	(2m)
	AM(sampling)	3.43	0.11%	(6s)	5.03	0.95%	(29s)	7.22	16.45%	(2m)
	LIH(T=3000)	3.62	5.54%	(46s)	5.03	0.92%	(95s)	6.58	6.24%	(3m)
	Att-GCRN+MCTS	3.43	**0.00%**	(-)	5.09	2.16%	(-)	6.65	7.32%	(-)
	DPDP(bs=10K)	3.43	**0.00%**	(5s)	5.03	1.00%	(3m)	6.86	10.66%	(12m)
	LIH(FS)(T=3000)	**3.43**	0.04%	(54s)	**5.03**	**0.88%**	(100s)	**6.37**	**2.77%**	(3m)
	LIH(FT)(T=3000)	**3.43**	**0.00%**	(2m)	**5.03**	**0.89%**	(3m)	**6.36**	**2.50%**	(6m)

Table 2: Results on TSPlib Instances. The underlined and bold figures mean achieving the best results among all methods (including OR-Tools) and all deep learning-based methods respectively.

Instance	Opt.	OR-Tools	AM (N=1,280)	AM (N=5,000)	LIH (T=3,000)	**LIH(FS)** (T=3,000)	**LIH(FT)** (T=3,000)
eil51	426	436	435	434	438	**429**	**429**
pr124	59,030	62,519	62,750	61,996	66,010	**61,645**	**61,645**
rd100	7,910	8,189	8,180	8,048	7,915	**8,036**	8,160
pr76	108,159	111,104	111,598	111,924	109,668	109,418	**108,495**
kroB150	26,130	27,572	28,894	28,864	31,407	**27,418**	**27,418**
u159	42,080	45,778	45,394	44,581	51,327	**43,376**	**43,376**
berlin52	7,542	7,945	9,759	9,831	8,020	7,653	**7,544**
eil101	629	664	656	656	658	**642**	656
kroC100	20,749	21,583	22,586	22,896	25,343	**21,079**	21,255
eil76	538	561	558	557	575	**548**	**548**
kroB100	22,141	23,006	24,340	23,987	26,563	**22,855**	23,677
kroE100	22,068	22,598	22,895	22,716	26,903	**22,532**	22,898
bier127	118,282	122,733	130,513	128,150	142,707	**127,520**	**127,520**
Avg. Gap (%)	0	3.46	42.96	36.86	17.12	**5.13**	5.49

5.1 Experimental Settings

Data normalization We only consider Instances within $[0, 1] \times [0, 1]$ which are normalised via min-max normalization:

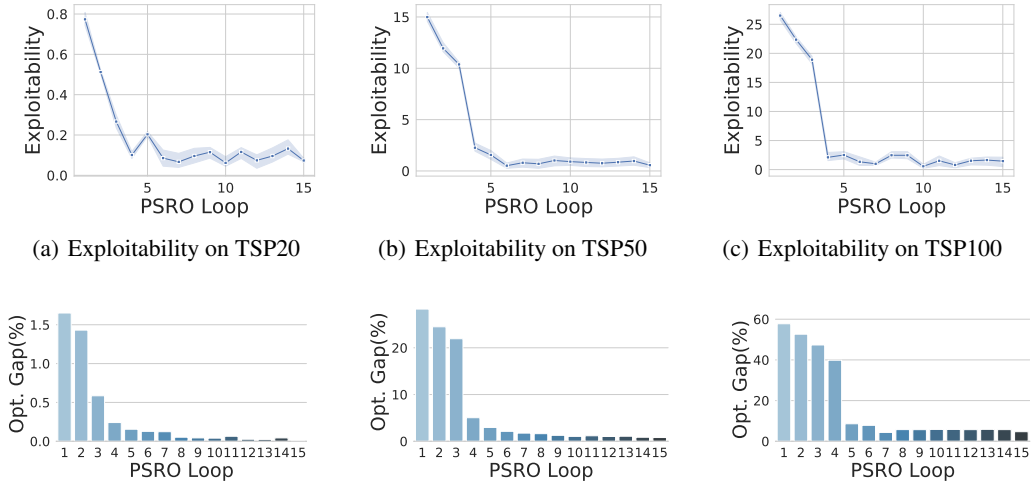
$$\mathcal{I}_{norm} = \text{Norm}(\mathcal{I}) = \frac{\mathcal{I} - \min\{\mathcal{I}\}}{\max\{\mathcal{I}\} - \min\{\mathcal{I}\}} \tag{14}$$

where $\min\{\mathcal{I}\}$ is the minimum scalar coordinate value in the TSP Instance and $\max\{\mathcal{I}\}$ is the max coordinate value.

Data generation We generate data by randomly sampling $x \in \mathrm{R}^2$ from the unit square, and sampling $y \in \mathrm{R}^2$ from $\mathrm{N}(\mathbf{0}, \Sigma)$ where $\Sigma \in \mathrm{R}^{2\times 2}$ is a diagonal matrix whose elements are sampled from $[0, \lambda]$ and $\lambda \sim \mathrm{U}(0, 1)$. Next, a two-dimensional coordinate is generated by $z = x + y$, and we can get any scale n of TSP by performing this sampling n times. We sample 1000 normalised (via Eq. 14) Instances which makes up 10 groups of data generated by different λ values, and report the relevant results on the generated datasets in Table 1.

Baselines. In this work, our base RL model comes from (Wu et al., 2021) which we denote as **LIH**, and we follow two different training paradigms: training from scratch **LIH(FS)** and fine-tuning **LIH(FT)**, with specific settings available in Appendix A.3. We compare our model with Gurobi (Gurobi Optimization, LLC, 2021), Concorde, LKH3 (Helsgaun, 2017) and the following deep learning-based methods: AM (Kool et al., 2018), (Wu et al., 2021), Att-GCRN+MCTS (Fu et al., 2020) [5] and DPDP (Kool et al., 2021) on generated data. On the real world instances from TSPLib (Reinelt, 1991), we compare the results with known best results, Or-tools (Perron & Furnon), AM (Kool et al., 2018) and LIH (Wu et al., 2021). All experiments are trained and executed with one single GPU (RTX3090) and CPU (i9-10900KF).

[5] We don't report the time because the implementation is using C which is different from others



(a) Exploitability on TSP20 (b) Exploitability on TSP50 (c) Exploitability on TSP100

(d) Opt. gap of combining Solvers trained on TSP20 (e) Opt. gap of combining Solvers trained on TSP50 (f) Opt. gap of combining Solvers trained on TSP100

Figure 2: Exploitability and performance of our model as the PSRO training goes on

5.2 RESULTS

Results on Generated Data We first evaluate how **LIH(FS)** and **LIH(FT)** perform on the unseen test data. In Table 1, we can see that the performance of deep-learning based methods trained on a uniform distribution degrades when dealing with Instances from an unseen distribution. On the other hand, our model obtained from the PSRO framework performs well out-of-distribution, and achieves state-of-the-art results among the deep-learning methods. Notably, we do not tune any hyperparameters in the original RL Solver, which suggests these improvements are based solely on the different training paradigm introduced by our method. However, due to the use of a mixing-strategy, the time consumed grows linearly compared with the base Solver due to the extra feed-forward computation, which can be seen as a trade-off between solution quality and runtime.

Results on Real-World Problems. We also test our **LIH** variants on real TSP problems from TSPLib (Reinelt, 1991), and report these results in Table 2. We maintain the settings from (Wu et al., 2021) with T= 3000, and the results reported for OR-Tools (Perron & Furnon) and **LIH** are directly taken from (Wu et al., 2021).

From the results in Table 2, we show that our model has the best performance among deep-learning based methods in the majority of Instances from TSPLib Reinelt (1991), and for the subsection: eil51, pr124, rd100, pr76, kroB150, u159, eil101, kroC100, eil76, kroB100, kroE100, bier127, our method is able to outperform OR-Tools.

6 DISCUSSIONS

In this section, we provide more insightful analysis into properties of the meta-games, the population Solvers and the effects of different initial solutions provided to the model.

6.1 META GAME ANALYSIS

To demonstrate the Game-Theoretic rationality of our method, we provide an analysis of the exploitability (Eq. 1) of our learned populations. Specifically, we train for 15 PSRO iterations on TSP20, TSP50, TSP100 and compute the exploitability of the obtained meta-strategy at each PSRO iteration. Results are shown in Fig. 2, with the visible decrease in exploitability demonstrating the validity of both our framework and the usage of the Nash equilibrium as the meta-solver. In particular, as training goes on, we generate strategies which approximately monotonically improve towards the Nash Equilibrium.

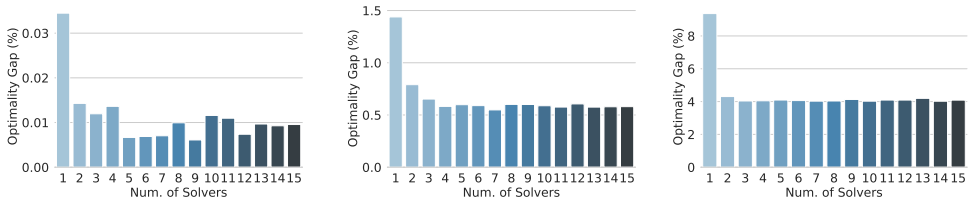


Figure 3: Optimality gap of mixing-Solver with different combined numbers on TSP20, 50, 100 (from left to right).

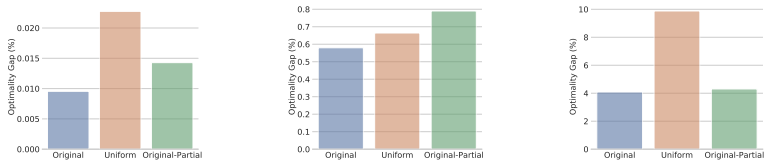


Figure 4: Optimality gap of mixing-Solver with different combined numbers and weights on TSP20, 50, 100 (from left to right)

6.2 Usage of a population of Solvers

In this section we discuss how restricting the number of Solvers available to be mixed-over, and how the choice of mixing weights may impact the final results. All results are obtained by **LIH(FS)** (T=1000), and we use Instances generated in the same manner as Section 5.

We first investigate the impact of the number of Solvers used in the combined-model. The Solvers obtained by PSRO are ranked according to their corresponding density in the meta-strategy, and we combine the top-k Solvers showing the performance in Fig. 3. Results show that our mixing method improves the performance significantly over the single most powerful Solver. In practice, it's preferable to use fewer Solvers in the combined-Solver as this reduces the amount of resources required. In this paper, we set the the probability threshold of 0.99 to choose solvers (1-3 solvers usually) to balance the performance and computational cost.

Furthermore, we investigate the impacts of mixing-weight. Here we consider three scenarios:

- Original: the whole population of Solvers combined by the Nash equilibrium meta-strategy.
- Uniform: the whole population of Solvers combined by a uniform meta-strategy.
- Original-Partial: the two most powerful Solvers as judged by the meta-strategy (i.e. they have the largest amount of density in the meta-strategy), combined by their respective normalized meta-strategy probabilities.

Fig. 4 shows the comparisons between the different cases listed above. We can see the 'Original' setting achieves the best results, which shows the theoretical stability of the Nash equilibrium meta-solver as described in Section. 4.4. In the case of the 'Uniform' setting, the performance degenerates as n increases, which we suspect is due to the equal importance given to all Solvers, even if some of them are particularly weak. For the 'Original-Partial' setting, the use of only 2 Solvers violates our original framework, specifically leading to a reduction in Solver diversity and a poor ability to deal with unseen problems.

7 Conclusions

Building on the framework of PSRO, in this paper, we propose the first game-theoretic solution to improving the generalization ability for any neural network-type TSP Solvers. On both randomly-generated and real-world TSP instances, we show that the Solvers trained under our two-player framework demonstrate the state-of-the-art generalization performance when compared to a series of strong TSP solution baselines. In principle, our proposed two-player game enables to improve the generalisation of the Solver population by decreasing its exploitability against an adaptive data Generator, which gets increasingly stronger during training. In future, we believe applying such a game-theoretic framework on other types of combinatorial optimization problems such as Vehicle Routing Problems will have an enormous economic impact.

REFERENCES

David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. In *International Conference on Machine Learning*, pp. 434–443. PMLR, 2019.

Wojciech Marian Czarnecki, Gauthier Gidel, Brendan Tracey, Karl Tuyls, Shayegan Omidshafiei, David Balduzzi, and Max Jaderberg. Real world games look like spinning tops. *arXiv preprint arXiv:2004.09468*, 2020.

Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

Trevor Davis, Neil Burch, and Michael Bowling. Using response functions to measure strategy strength. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

Zhang-Hua Fu, Kai-Bin Qiu, and Hongyuan Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. *arXiv preprint arXiv:2012.10658*, 2020.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL `https://www.gurobi.com`.

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

André Hottung, Bhanu Bhandari, and Kevin Tierney. Learning a latent search space for routing problems using variational autoencoders. In *International Conference on Learning Representations*, 2020.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. Deep policy dynamic programming for vehicle routing problems. *arXiv preprint arXiv:2102.11756*, 2021.

Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. *arXiv preprint arXiv:1711.00832*, 2017.

Xiangyu Liu, Hangtian Jia, Ying Wen, Yaodong Yang, Yujing Hu, Yingfeng Chen, Changjie Fan, and Zhipeng Hu. Unifying behavioral and response diversity for open-ended learning in zero-sum games. *arXiv preprint arXiv:2106.04958*, 2021.

Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2019.

Stephen McAleer, John Lanier, Roy Fox, and Pierre Baldi. Pipeline psro: A scalable approach for finding approximate nash equilibria in large games. *arXiv preprint arXiv:2006.08555*, 2020.

H Brendan McMahan, Geoffrey J Gordon, and Avrim Blum. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 536–543, 2003.

Nicolas Perez Nieves, Yaodong Yang, Oliver Slumbers, David Henry Mguni, and Jun Wang. Modelling behavioural diversity for learning in open-ended games. *arXiv preprint arXiv:2103.07927*, 2021.

Laurent Perron and Vincent Furnon. Or-tools. URL `https://developers.google.com/optimization/`.

Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4): 376–384, 1991.

Ricky Sanjaya, Jun Wang, and Yaodong Yang. Measuring the non-transitivity in chess, 2021.

Max Olan Smith, Thomas Anthony, Yongzhao Wang, and Michael P Wellman. Learning to play against any mixture of opponents. *arXiv preprint arXiv:2009.14180*, 2020.

Max Olan Smith, Thomas Anthony, and Michael P Wellman. Iterative empirical game solving via single policy best response. *arXiv preprint arXiv:2106.01901*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Michael P Wellman. Methods for empirical game-theoretic analysis. In *AAAI*, pp. 1552–1556, 2006.

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems.. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

Yaodong Yang and Jun Wang. An overview of multi-agent reinforcement learning from game theoretical perspective. *arXiv preprint arXiv:2011.00583*, 2020.

Tianshu Yu, Runzhong Wang, Junchi Yan, and Baoxin Li. Learning deep graph matching with channel-independent embedding and hungarian attention. In *International conference on learning representations*, 2019.

Tianshu Yu, Runzhong Wang, Junchi Yan, and Baoxin Li. Deep latent graph matching. In *International Conference on Machine Learning*, pp. 12187–12197. PMLR, 2021.

A Appendix

A.1 Oracle Training

In the Algorithm 1, we need train two oracles: $S^{'}$ and $\mathbf{P}_{\mathcal{I}}^{'}$ as a new policy to be added to the corresponding policy set. Here we will provide a specific derivation for training the oracle in our combinatorial optimization problems setting.

Taken the formula from Eq. 6, the gradient is apparent to get:

$$\begin{aligned}\nabla_\theta L_{\text{SS}}(\theta) &= \nabla_\theta \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} g(S_\theta, \mathcal{I}, \text{Oracle}) \\ &= \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} \nabla_\theta g(S_\theta, \mathcal{I}, \text{Oracle}) \\ &= \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} \frac{\nabla_\theta S_\theta(\mathcal{I})}{\text{Oracle}(\mathcal{I})} \\ &= \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{N\sim\mathbf{P}_N} \mathbf{E}_{x_1,...,x_N\sim\prod_{i=1}^N \mathbf{P}_C} \frac{\nabla_\theta S_\theta(x_1,...,x_N)}{\text{Oracle}(x_1,...,x_N)}.\end{aligned} \tag{15}$$

Also for Eq. 11, the computation of this gradient is:

$$\begin{aligned}\nabla_\gamma L_{\text{DG}}(\gamma) &= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \nabla_\gamma \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} g(S, \mathcal{I}, \text{Oracle}) \\ &= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \int_{\mathcal{I}} \nabla_\gamma \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) g(S, \mathcal{I}, \text{Oracle}) \mathbf{d}\mathcal{I} \\ &= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \int_{\mathcal{I}} \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) \frac{\nabla_\gamma \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I})}{\mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I})} g(S, \mathcal{I}, \text{Oracle}) \mathbf{d}\mathcal{I} \\ &= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} \nabla_\gamma \log \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) g(S, \mathcal{I}, \text{Oracle}).\end{aligned} \tag{16}$$

Furthermore, we can take a expansion on $\mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} \nabla_\gamma \log \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I})$ in the last line w.r.t. $\gamma = (\gamma_C, \gamma_N)$:

$$\mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} \nabla_\gamma \log \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) = \begin{pmatrix} \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^N \mathbf{P}_{C,\gamma_C}} \nabla_{\gamma_C} (\sum_{i=1}^N \log \mathbf{P}_{C,\gamma_C}(x_i)) \\ \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \nabla_{\gamma_N} \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^N \mathbf{P}_{C,\gamma_C}} \log \mathbf{P}_{N,\gamma_N}(N) \end{pmatrix}$$

After taking the above formula into Eq. 16, we complete the derivation of gradients about the Data Generator.

A.2 COMPUTATION OF LOG-PROBABILITY

An extra computation is needed for the log-probability in Eq. 9 and we do this in the following way: Assuming it's independent between each dimension in a two-dimension coordinate, we only show the one-dimension case without loss of generality.

Formally, there are two random variables $X \sim \mathrm{U}(0,1)$ and $Y \sim \mathrm{N}(0,\sigma^2)$, and we are to compute the probability density function of random variable $Z = X + Y$. We get:

$$\mathrm{P}(Z \leq z) = \mathrm{P}(X + Y \leq z) = \int_0^1 dx \int_{-\infty}^{z-x} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{y^2}{2\sigma^2}) dy$$

and we have:

$$\mathrm{p}(z) = \frac{d\mathrm{P}(Z \leq z)}{dz} = \int_0^1 \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(z-x)^2}{2\sigma^2}) \mathrm{d}x.$$

All we need to do is to approximate this integration. Various methods can be used do so. In this work, we handle this simple integration by Monte Carlo sampling by sampling 10000 samples within $[0,1]$ to make a rough approximation. After obtaining the approximated probability, we can easily get the log-probability due to the independent assumption.

A.3 DETAILED EXPERIMENTAL SETTINGS

Hyperparameters. We don't propose any specific RL Solver in this work since our method is a unified framework to suit any previous models. So in this paper, we use **LIH** as our base model. All settings about the RL Solver is same as the original paper.

For the settings of Data Generator, we initialize the γ_N randomly and use a simple three-layer neural networks to represent the attack generator f_{γ_C} in Eq. 8. We also use a Sigmoid function as the last layer to scale the variation within $[0,1]$ and an additional scalar $\lambda \in [0,1]$ to make a further limit within $[0,\lambda]$. Here we set $\lambda = \frac{1}{3}$ because of the '68-95-99.7 rule' which is a famous principle in statistics. It not only guarantees each point within [0,1] can reach any other point after adding a gaussian perturbation, but also makes few changes to the structure of original Instances after normalization in Eq. 14.

All parameters in our framework except for those in the RL Solver are updated by Adam (Kingma & Ba, 2014) optimizer with specific learning rate settings and the overall configuration of this neural networks is shown in Table 3.

Table 3: Configuration of Attack Neural Networks

Module	DESCRIPTION
First Layer	dim=2 with ReLU activation
Second Layer	dim=128 with ReLU activation
Output Layer	dim=2 with Sigmoid activation
Optimizer	Adam with lr=0.05, lr_decay=0.95
l_2 norm	weight_decay=0.01
Epochs	40 for TSP20 TSP50 and TSP100

During the training at each PSRO loop, we choose the attack generator where the current Solver Selector performs worst for the next PSRO loop. Similarly, we choose the Solver with best performance on the mixing distribution constructed by the current Data Generator. Specifically, we generate a validation set by sampling 1000 Instances from the distribution constructed by the Data Generator's policy set and its meta-strategy. We then test each epoch's model on this dataset and select the best one as the model to be trained in the next PSRO loop.

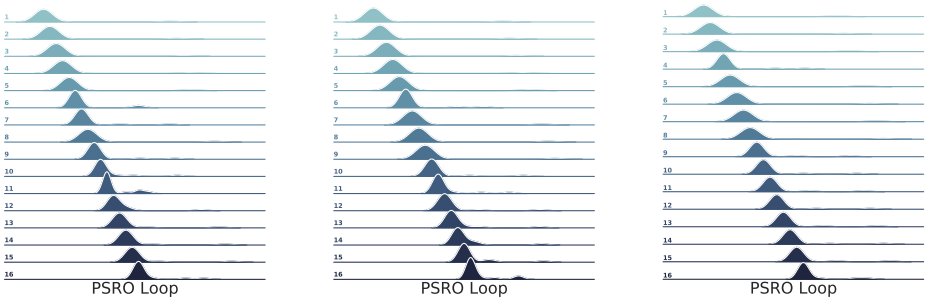
Fine-tune version. We also provide a fine-tune version under our training framework. Specifically, we pick the model as a warm start which has pretrained on the uniform distribution and continue to train them under the framework of PSRO, which can be seen as the fine-tune process to overcome the weakness of the current model. We call this version of model **LIH(FT)** in the following. Respectively, we denote the version of model that trains from scratch as **LIH(FS)**. For practical use, we often use the fine-tune model rather than the model train from scratch because of the limit of time and computational resource, and in this work, we test the fine-tune model trained on TSP100 to solve the Instances from TSPLIB (Reinelt, 1991)).

Setup in meta-level. Under the framework of PSRO, we train oracles for Solver Selector and Data Generator at each PSRO loop. During training **LIH(FS)**, we set the same training epochs for RL Solver: 40 epochs (per PSRO loop) for TSP20, TSP50 and TSP100 and we train 7 PSRO loop in each case.. When we train the fine-tune version, **LIH(FT)**, we use the model in the last epoch of training period in original paper as our pretrained model (for (Wu et al., 2021), we use the model trained after 200 epochs.) Then we train 10 epochs for TSP100 in each PSRO loop and train 8 PSRO loop. During the training, the Solver inherit the parameters of last PSRO loop and continue to train in the new PSRO loop. Noticing that we obtain a population of Solvers by 280 epochs of training, we train 280 epochs for **LIH** rather than 200 epochs in its original paper to guarantee the fair comparison.

Mixing-model. After getting a population of Solvers, we use the mixing policy obtained by Eq. 12 or 13 to combine these Solver. Considering we need to get each Solver's policy during each decision step, we need to execute forward propagation for each Solver so the running time will grow linearly if there are no implementation-level tricks. As a consequence, we only use the Solvers whose probabilities are accumulated no less than 0.99 because of Solver Selector's sparse meta-strategy.

A.4 Meta Strategy in Different PSRO Loops

We visualize the meta strategy in every PSRO loops in Fig. 5. Results show that at each loop, there exists the strongest Solver with a dominate meta strategy probability, leading to a quite sparse meta strate



(a) Meta-strategy of the population of Solvers trained on TSP20.

(b) Meta-strategy of the population of Solvers trained on TSP50.

(c) Meta-strategy of the population of Solvers trained on TSP100.

Figure 5: Meta strategy of the model population.

A.5 Weakness of Solvers

Under our framework, it's interesting to find some distributions where the Solvers (or methods) can perform poorly, revealing the weakness of the solver. It can also provide a rough judgement on the stability of a method. We are amazed to find that only using simple multi-layer neural networks, the same as that during training oracles for Data Generator, the methods show diverse performance, as shows in Appendix. 6 and 7. Therefore, it's reasonable to take this criterion into consideration when comparing different methods. However, there are few researches about the exploration about the weakness but we think it's quite important especially in realistic applications.

We demonstrate performance can be influenced a lot even by adding small gaussian perturbations in Fig. 6 and 7. We use the model trained in corresponding paper: training 200 epochs for LIH (Wu et al., 2021) and 100 epochs for AM (Kool et al., 2018) on TSP20, TSP50 and TSP100. Results show that our attack generator can learn a distribution where the well-trained model performs bad, which motivates us to employ such method to train oracles under the framework of PSRO.

A.6 Demonstration of Attack Distribution

We visualize the attack distribution obtained by each PSRO loop in Fig. 8. Specifically, Fig. 8(a), 8(c) and 8(e) are points which comprises 1000 Instances. Fig. 8(b), 8(d) and 8(f) are corresponding kernel density estimations.

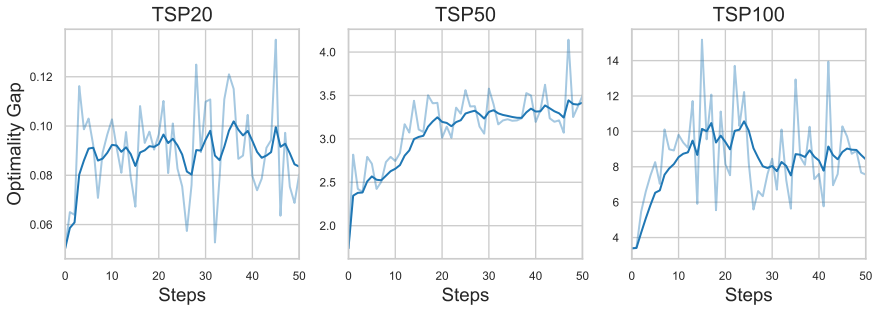


Figure 6: Training figure of attack generator for LIH (Wu et al., 2021).

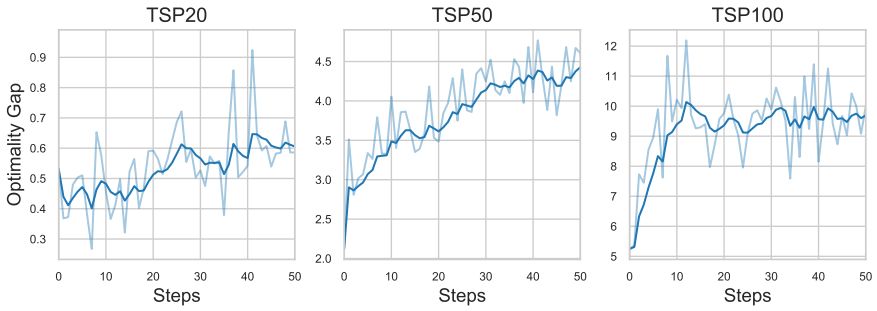
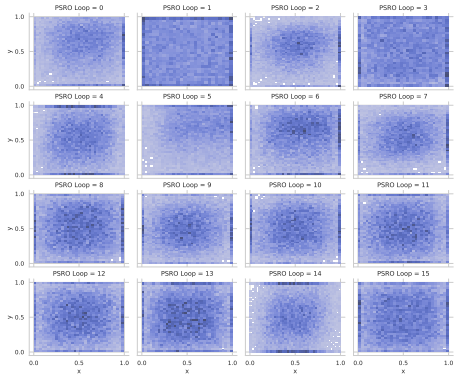
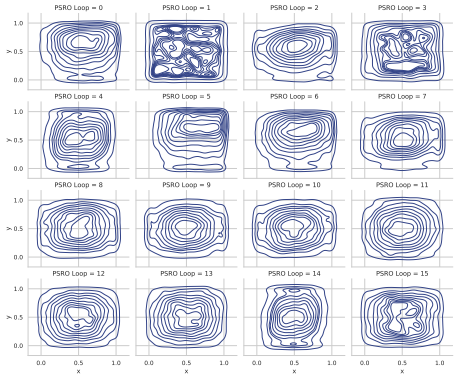


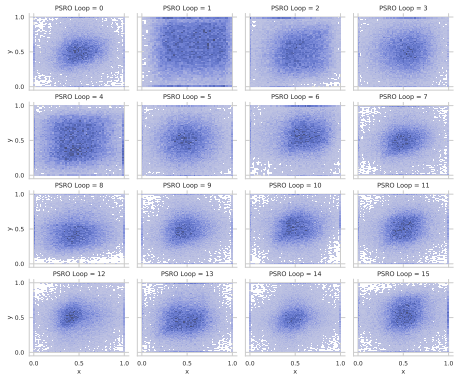
Figure 7: Training figure of attack generator for AM (Kool et al., 2018)



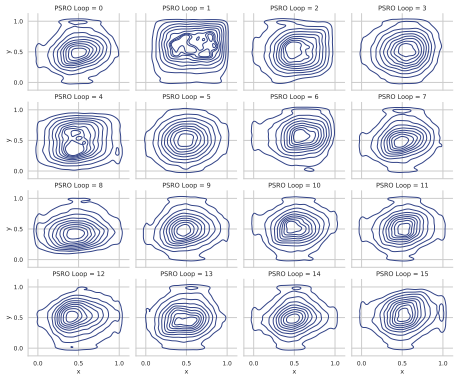
(a) Attack distribution generated by PSRO trained on TSP20.



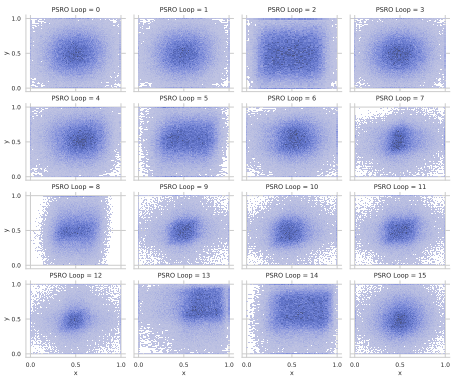
(b) Kernel density estimations of attack distribution generated by PSRO trained on TSP20.



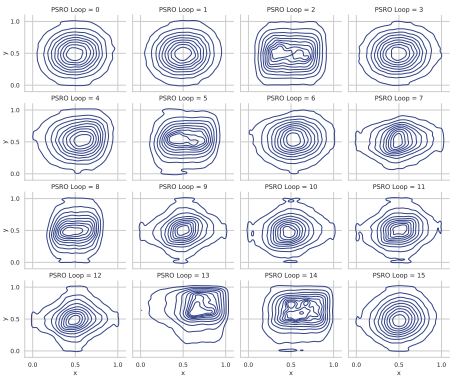
(c) Attack distribution generated by PSRO trained on TSP50.



(d) Kernel density estimations of attack distribution generated by PSRO trained on TSP50.



(e) Attack distribution generated by PSRO trained on TSP100.



(f) Kernel density estimations of attack distribution generated by PSRO trained on TSP100.

Figure 8: Attack distribution generated by PSRO