Transformer encoder for TSP

Alessandro Bizzarri, Davide Caffagni

1 Introduction

The Travelling Salesman Problem (TSP) is a NP-hard combinatorial optimization problem aiming to find the shortest possible hamiltonian tour in a graph. Although it has been studied for years, there's no known exact solution for it. Recently, deep neural networks (DNN) have been proposed to address it, sometimes resulting in encouraging outcomes. In a typical DNN setup, an *encoder* network is in charge to extract a latent representation of the graph. This is then fed to a *decoder* that outputs a probability distribution from which the next node is sampled. Such an *autoregressive* approach could be cumbersome, either because it scales linearly with the size of the graph and it doesn't exploit well hardware acceleration. In this work, we question whether it is possible to design a neural network that can directly provide a valuable TSP solution without resorting to any autoregressive component.

2 Related works

2.1 Transformers

Vaswani et al. proposed a new approach to tackle sequence modeling by applying self-attention on the sequence itself in the Transformer architecture[11]. It begins with an encoder component which starts by performing multi-head attention on the inputs and extracts a set of *queries*, *keys* and *values* to have a complete understanding of the input sequence. The *keys* and *values* will be shared with the next component, the decoder, where they will be used in conjunction with the queries extracted in the decoder itself to perform cross attention and later to output the predictions for the next elements in the sequence. The Transformer was originally born for machine translation, but then they've been extended to computer vision and, recently, even to combinatorial optimization problems.

2.2 Transformer for TSP

Bresson et al. [5] showed that transformers can be used to address the TSP. Their model first embeds each node in the graph into a latent representation using a stack of self-attention blocks. Then, an autoregressive decoder queries the encoded graph with masked cross-attention to output a probability distribution from which the next node will be selected.

Their encoder differs from the original one because it substitutes layer normalization [3] with batch normalization [7]. Additionally, they replace the first self-attention block in each decoder layer with a cross-attention one. Here, they extract from the encoder output the embedding of the last selected node, which acts as a *query*. On the other hand, *keys* and *values* are obtained by linearly projecting each node in the current partial tour. The outcome of this layer will be in turn the *query* of the subsequent encoder-decoder cross-attention.

2.3 Optuna

Working with artificial intelligence could lead to many different experiments where each one tests a single hyperparameter while the others are either left untouched or modified without a specific reason. To avoid wasting time with the definition of hyperparameters, not always being sure that the ones used are the most optimal ones, we took advantage of Optuna [2], an automatic hyperparameter optimization software framework which allows the user to define a search space regarding the hyperparameters that are to be optimized with respect to a user defined objective function. Then, the user can choose one of the many multiple optimization algorithms already implemented, such as Grid Search, Random Search and, in our studies, Tree of Parzen Estimator proposed by Bergstra et al [4], which is an iterative process that uses history of evaluated hyperparameters to create a probabilistic model, which is used to suggest the next set of hyperparameters to evaluate the defined objective function. Moreover, the optimization can be easily implemented to run in parallel on multiple processes to allow for a more fine tuning of the architecture where each process will test a given number of trials and share between each process the acquired knowledge to be used in the Tree of Parzen Estimator. Finally, when the optimization has completed, we save locally the hyperparameters provided and proceed to train more thoroughly the network.

2.4 REINFORCE

When applying neural network to combinatorial optimization, *reinforcement learning* is a popular choice, primarily because it doesn't require a supervised signal, which may be either unknown or derived from an heuristic, and thus potentially sub-optimal. In this work, we resort to the REINFORCE [12] algorithm.

Given a policy π_θ, parametrized by θ, learning is done by *gradient ascent*

$$\theta_{t+1} \leftarrow \theta_t + \lambda \nabla_{\theta_t} J(\theta_t)$$

to maximize the *expected sum of discounted rewards*

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[r(\tau)].$$

In REINFORCE, the gradient is estimated by Monte Carlo sampling:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t), \tag{1}$$

where a_t and s_t are the *action* and *state* at time t respectively.

3 Proposed method

In this section we explain how we address the TSP using a transformer network. Two main design choices have been taken into account. First of all, we stick with the *attention* as the main building block of our solution. Indeed, this operator, inherently global, can potentially capture any relationship between any pair of nodes in a graph. At the same time, we focus on avoiding any autoregressive component, thus providing a model which outputs a complete tour using a single forward pass.

3.1 Architecture

Understanding our method is easier if one can visualize what's happening behind an attention layer, so here we provide a brief recap.

1. Given an input set $X \in \mathbb{R}^{n \times d}$, we linearly project it to three different vectors, commonly referred to as *query*, *key* and *value*:
   $Q = XW_Q$, $K = XW_K$, $V = XW_V$,
   with $W_Q, W_K \in \mathbb{R}^{d \times d_k}$, $W_V \in \mathbb{R}^{d \times d_v}$.
2. Then we compute the *attention matrix* as
   $A = \frac{QK^\top}{\sqrt{d_k}}$.
3. The output of the attention is finally
   $Attention(X) = softmax(AV)$

Conceptually, this projects a set of *tokens* to a different latent representation, in which each token is the outcome of a weighted linear combination. Specifically, these weights are computed dynamically and are proportional to the *similarities* among the tokens.

When Q, K and V come from the same input X, we talk about *self-attention*. Conversely, if Q comes from X, while K and V are linear projection of a different input, it is a *cross-attention*.

As depicted in Figure 1, our model can be seen as a transformer encoder network, in which each layer has an additional cross-attention between the self-attention and the feed-forward blocks. The self-attention is required to build a global representation of the graph. Note that we don't integrate the input with any positional encoding here, effectively treating it as a set rather than a sequence. This is only partially true for the TSP task, since the features of the graph nodes inherently define a spatial order. Anyway, the proposed method may be generalized to any problem requiring to turn a set into a sequence.

Given any positional encoding *pe* (i.e. such the one proposed in [11]), the extra cross-attention block exploits *pe* as query, while key and value come from the global context provided by the self-attention. From an high level point of view, we'are weighting each node in the graph by its similarities with positions. After applying *softmax*, the i^{th} row of an attention matrix A such computed is a probability distribution, giving the odds for the j^{th} node of a graph to be in the i^{th} position in a tour.

The attention matrix computed in the last layer of our model, $\hat{A}$, is used as inverse cost matrix for solving an instance of the linear sum assignment problem [10]. This avoids any node repetition, leading to a permutation of the graph. We then manually close the tour by appending the first selected node to the end.

3.2 Sinkhorn operator

Ideally, we'd like $\hat{A}$ to be a permutation matrix, thus obtaining a valid tour without the need to solve a linear sum assignment. While this can be deemed as an additional task to teach to the model, we found this extra complexity, together with the inherent hardness of the TSP, to be too difficult to learn within a reasonable time. We leave this to further developments.

Anyway, thanks to the Sinkhorn operator [1], we can still obtain a matrix which is *close* to a permutation one, while being able to back-propagate the gradient through it. Given a matrix X, the Sinhorn operator is defined as

$$\begin{aligned} S^0 &= exp(X/\tau) \\ S^l &= \mathcal{T}_c(\mathcal{T}_r(S^{l-1}(X))) \\ S(X) &= \lim_{l \to \infty} S^l(X)\ . \end{aligned} \tag{2}$$

That is, we repeatedly normalize a matrix row and column wise. This leads to a double-stochastic matrix. Mena et al., 2018 [9] showed that for $\tau \to 0^+$, $S(X)$ becomes a permutation matrix. Throughout our experiments, we found that τ has a dramatic impact on the overall performance of the network, which improves as τ gets smaller. Due to numerical stability issues, we end up fixing $\tau = 0.02$.

3.3 Loss function

The action space of the reinforcement learning setup for TSP is, as in [5], the set of the nodes in the graph $\{a_t : t = 0, 1, ..., n-1\}$. The agent receives a delayed reward only at the end of the episode, according to the

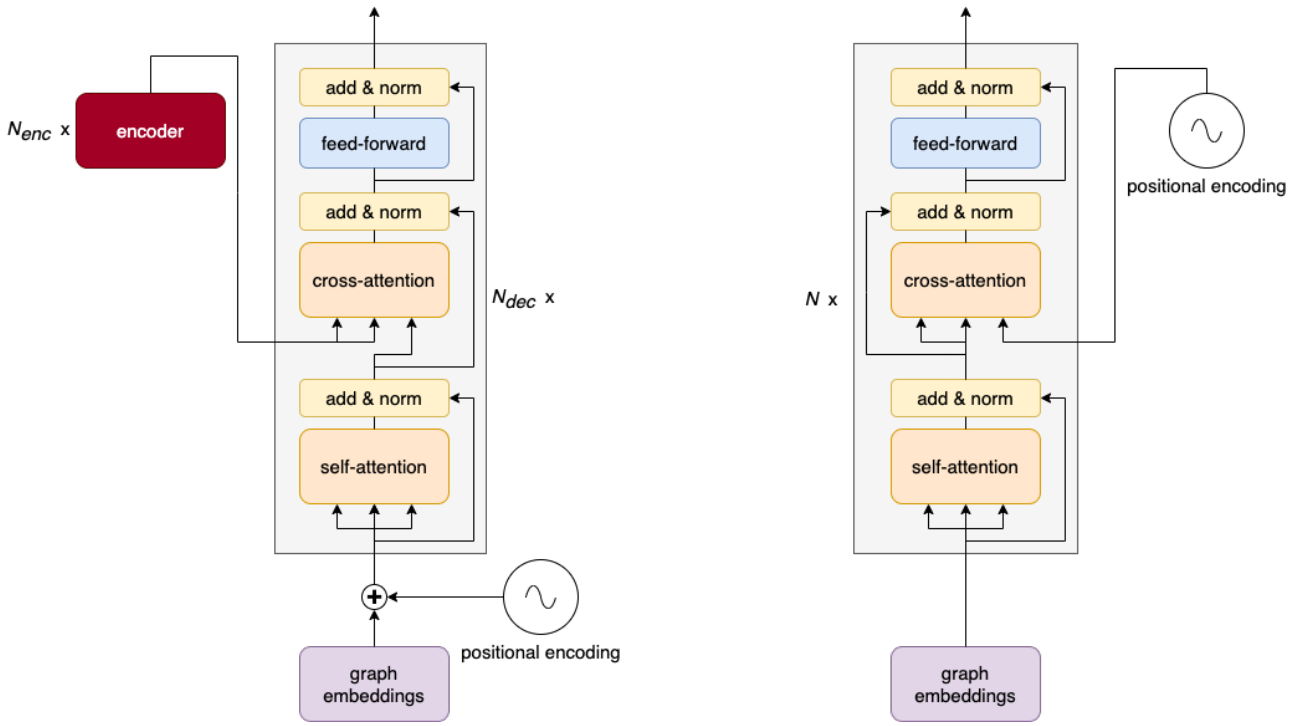

Figure 1: Comparison between the original transformer architecture (*left*) and our implementation (*right*).

chosen trajectory τ, i.e. a valid tour. Specifically, the reward of a trajectory is

$$r(\tau) = L(\hat{\tau}) - L(\tau) \ , \tag{3}$$

where $L(\tau)$ is the length of the tour given by τ, and $L(\hat{\tau})$ is a *baseline* reward. We've experimented two different baselines: one computed from the same model in inference mode, as in [5], the other obtained with the Christofides algorithm. It is worth noting that, while we were able to train the agent of Bresson et al. with either baselines, our network could only learn with the former.

We end up with the following loss function:

$$J(\theta) = \lambda J(\theta)_R + (1 - \lambda) J(\theta)_H \ , \tag{4}$$

where $J(\theta)_R$ is the REINFORCE loss, λ is a hyperparameter typically set to 0.6, and $J(\theta)_H$ is a regularization term to penalize actions (i.e. nodes) with high entropy. In particular, since our agent outputs a matrix $\hat{A}$ whose rows are probability distributions, we compute a weighted mean of the entropy of each row of $\hat{A}$, with weights linearly increased from 1 to n. This can be thought as asking the agent to be more and more confident in adding nodes to the partial tour built so far. Refer to Figure 2 to see the effects on the training.

3.4 Dataset

To tackle this problem we need to train the model on a great number of examples and we chose to generate them ourselves. This is done thanks to the use of *NetworkX* [6], a Python package to work with graphs. We implemented a simple script that generates as many graphs as we need with properties that fit our study. Each graph is generated with 50 nodes placed in a Cartesian space, all linked together by bi-directional edges and whose weights are represented by the distance of the two linked nodes. One consideration to be made is that for each graphs, the coordinates of the nodes are normalized to be inside the range [0,1] on both axis, this was done to introduce invariance with respect to scale to help the model generalize better as we can apply this normalization to every real life use-case. Finally, we generate distinct datasets for train, test and evaluation with respectively 100'000, 10'000 and 5'000 graphs.

4 Experiments

We compare our method against the one proposed in [5]. Both agents were trained with REINFORCE, using the Adam [8] optimizer with constant learning rate of 0.001. We too replaced the layer norm with batch norm. Unfortunately, due to hardware constraints, we trained the baseline with a maximum batch size of 256, rather than 512, as originally suggested.

4.1 Comparisons

To finally test the performances of our model we put a focus on the two main characteristic that can define the predictions made. How *good* they are and how much *time* did it take the model to make them. We compared our results to both one of the heuristics already established in the field, Christofides algorithm, and the architecture proposed by Bresson et al.. We start by focusing on the goodness of the predictions, which for

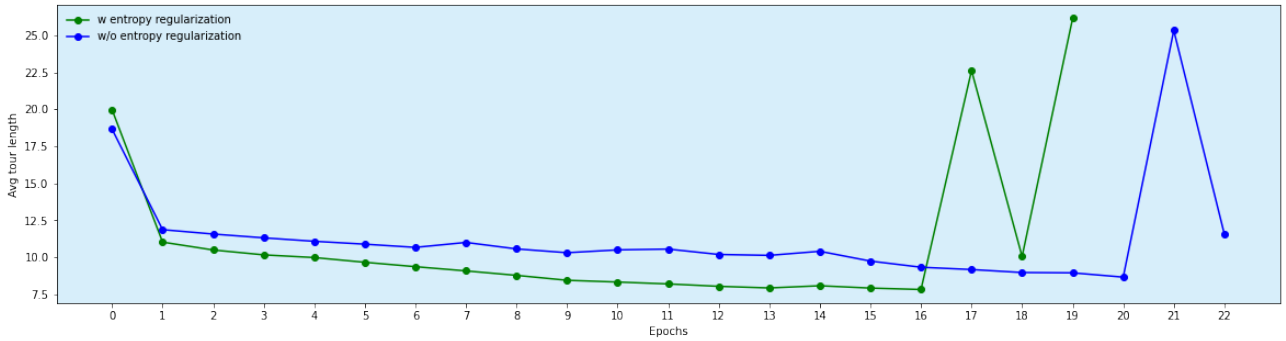

Figure 2: Impact of the entropy regularization term on training.

this problem is equivalent to how short is the final tour provided. As show in 3 and 1, our architecture performs worse compared to the other two approaches. About 18% worse than the heuristic taken into consideration. The other metric in question being time shows a different behaviour 4. We achieved an inference time which is nearly constant throughout the dataset while also being faster than the two other methods that also show some inconsistencies in the time required for the generation of the solution.

	tour length	time (ms)
Christofides	**6.41**	351.395
Bresson et al.	7.71	20.647
Ours	7.82	**1.964**

Table 1: Average performance comparison.

4.2 ILS

To further improve the results provided by our model, we perform Iterated Local Search on the outputted tours. Generally speaking, ILS is a modification of local search methods to avoid getting stuck in local optima. We show below the general idea of local search.

k	type	tour length	improvement (%)
5	s	7.81	0.095
	c	**7.70**	**1.618**
10	s	7.82	0
	c	7.82	0.031
20	s	7.82	0
	c	7.82	0

Table 2: Comparison between *standard* ILS (s) and our *custom* (c) implementation. The improvement is expressed wrt our best model.

Algorithm 1 Local Search

init $x \leftarrow$ initial solution
for $n_{iterations}$ **do**
  $x' \leftarrow perturb(x)$
  if $f(x') < f(x)$ **then**
    $x \leftarrow x'$
  end if
end for

As we can see, the local search can get stuck in local optima, minimum in this case, because the perturbation are always within a given range and always refer to the current best solution. ILS wants to solve this by adding to the local search the possibility of restarting the search from a completely different point. This is achieved by perturbing the solution by a far greater amount with respect to the perturbation done in the local search, while still performing the local search on the new staring point.

Algorithm 2 Iterated Local Search

init $x \leftarrow$ initial solution
for $n_{restarts}$ **do**
  $x' \leftarrow perturb_{heavy}(x)$
  for $n_{iterations}$ **do**
    $x'' \leftarrow perturb_{light}(x')$
    if $f(x'') < f(x)$ **then**
      $x \leftarrow x''$
    end if
  end for
end for

Finally, we implemented ILS by applying it on the output of our model to potentially improve the results. But implementing ILS as described above would mean completely ignoring why the model selected the given tour as at each iteration it would perturb randomically the solution. To take advantage of both components, we extract from the output of the model the worst nodes, defined as the nodes whose entropy was the highest computed from the attention matrix. This means that the network has low confidence when placing those nodes in the tour, so it could prove useful to perturb those indexes. Finally, as a good tradeoff between time and overall improvement, we simply swap

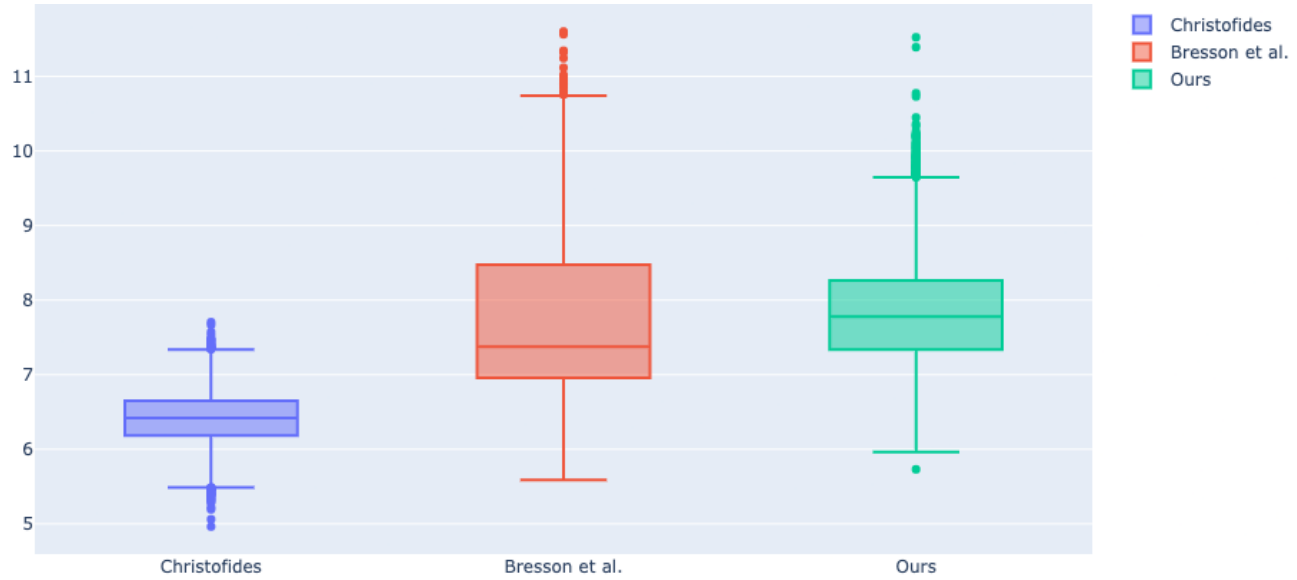

Figure 3: Tour length of the three compared methods.

Figure 4: Time required to solve each of the 10000 test samples.

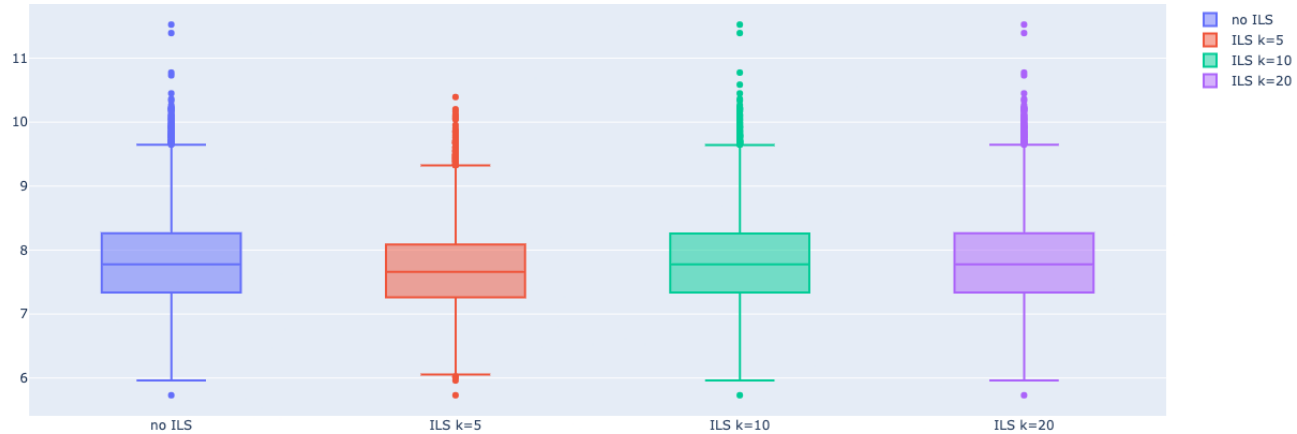

Figure 5: Tour length comparison with our ILS and different k.

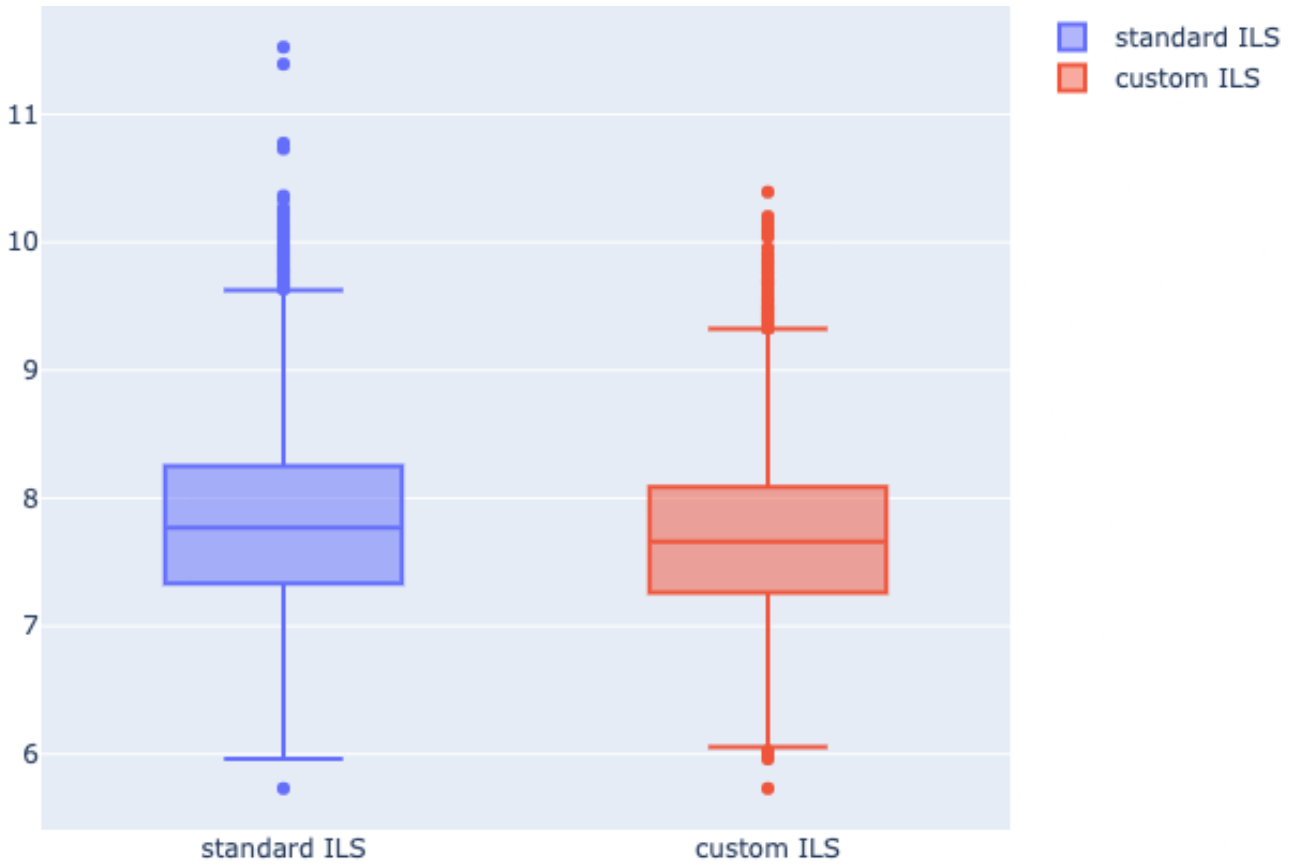

Figure 6: Tour length comparison between standard ILS and our implementation, both with k=5 .

two of the worst nodes in the sequence and check if the resulting tour is an improvement or not and we repeat it a number of times.

Algorithm 3 Our ILS

```
init x ← model output
worst nodes ← top-k nodes by entropy
for n_restarts do
    x' ← perturb(x, worst nodes)
    for n_iterations do
        x'' ← 2-opt(x', worst nodes)
        if f(x'') < f(x) then
            x ← x''
        end if
    end for
end for
```

5 Conclusions

In this work, we present a novel method to tackle the TSP problem. Differently from other neural approaches, our model doesn't require any autoregressive component. It proved to be faster than other methods, but, as a tradeoff, it performs worse. We hope to provide a valuable starting point for further developments.

References

[1] Ryan Prescott Adams and Richard S Zemel. "Ranking via sinkhorn propagation". In: *arXiv preprint arXiv:1106.1925* (2011).

[2] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[4] J.S. Bergstra, D. Yamins, and D.D. Cox. "Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms". In: *Python for Scientific Computing Conference* (Jan. 2013), pp. 1–7.

[5] Xavier Bresson and Thomas Laurent. "The transformer network for the traveling salesman problem". In: *arXiv preprint arXiv:2103.03012* (2021).

[6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[7] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[8] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[9] Gonzalo Mena et al. "Learning latent permutations with gumbel-sinkhorn networks". In: *arXiv preprint arXiv:1802.08665* (2018).

[10] $scipy.optimize.linear_sum_assignment x2014; SciPy v1.10.0 Manual - - - docs.scipy.org$. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html`. [Accessed 18-Feb-2023].

[11] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[12] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Reinforcement learning* (1992), pp. 5–32.