Learning the Travelling Salesperson Problem Requires Rethinking Generalization

Chaitanya K. Joshi[1], **Quentin Cappart**[2], **Louis-Martin Rousseau**[2], **Thomas Laurent**[3]
[1]University of Cambridge, UK
[2]Ecole Polytechnique de Montréal, Canada
[3]Loyola Marymount University, USA
chaitanya.joshi@cl.cam.ac.uk

Abstract

End-to-end training of neural network solvers for graph combinatorial optimization problems such as the Travelling Salesperson Problem (TSP) have seen a surge of interest recently, but remain intractable and inefficient beyond graphs with few hundreds of nodes. While state-of-the-art learning-driven approaches for TSP perform closely to classical solvers when trained on trivially small sizes, they are unable to generalize the learnt policy to larger instances at practical scales. This work presents an end-to-end *neural combinatorial optimization* pipeline that unifies several recent papers in order to identify the inductive biases, model architectures and learning algorithms that promote generalization to instances larger than those seen in training. Our controlled experiments provide the first principled investigation into such *zero-shot* generalization, revealing that extrapolating beyond training data requires rethinking the neural combinatorial optimization pipeline, from network layers and learning paradigms to evaluation protocols. Additionally, we analyze recent advances in deep learning for routing problems through the lens of our pipeline and provide new directions to stimulate future research.[1]

1 Introduction

NP-hard combinatorial optimization problems are the family of integer constrained optimization problems which are intractable to solve optimally at large scales. Robust approximation algorithms to popular problems have immense practical applications and are the backbone of modern industries. Among combinatorial problems, the 2D Euclidean Travelling Salesperson Problem (TSP) has been the most intensely studied NP-hard graph problem in the Operations Research (OR) community, with applications in logistics, genetics and scheduling [44]. TSP is intractable to solve optimally above thousands of nodes for modern computers [3]. In practice, the Concorde TSP solver [2] uses linear programming with carefully handcrafted heuristics to find solutions up to tens of thousands of nodes, but with prohibitive execution times.[2] Besides, the development of problem-specific OR solvers such as Concorde for novel or under-studied problems arising in scientific discovery [61] or computer architecture [50] requires significant time and specialized knowledge.

An alternate approach by the Machine Learning community is to develop generic learning algorithms which can be trained to solve *any* combinatorial problem directly from problem instances themselves [67, 6, 7]. Using classical problems such as TSP, Minimum Vertex Cover and Boolean Satisfiability as benchmarks, recent *end-to-end* approaches [39, 60, 46] leverage advances in graph representation learning [40, 24, 65, 5] and have shown competitive performance with OR solvers on trivially small problem instances up to few hundreds of nodes. Once trained, approximate solvers based on Graph Neural Networks (GNNs) have significantly favorable time complexity than their OR counterparts, making them highly desirable for real-time decision-making problems such as TSP and the associated class of Vehicle Routing Problems (VRPs).

[1]Code and datasets: github.com/chaitjo/learning-tsp
[2]The largest TSP solved by Concorde to date has 109,399 nodes with running time of 7.5 months.

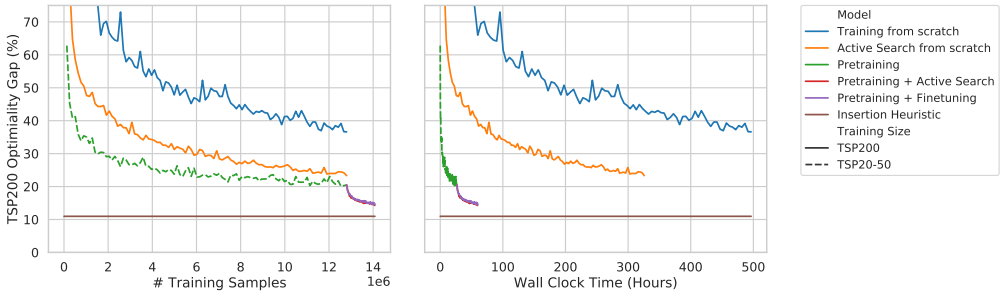


Figure 1: **Computational challenges of learning large scale TSP.** We compare three identical autoregressive GNN-based models trained on 12.8 Million TSP instances via reinforcement learning. We plot average optimality gap to the Concorde solver on 1,280 held-out TSP200 instances vs. number of training samples (left) and wall clock time (right) during the learning process. Training on large TSP200 from scratch is intractable and sample inefficient. Active Search [6], which learns to directly overfit to the 1,280 held-out samples, further demonstrates the computational challenge of memorizing very few TSP200 instances. Comparatively, learning efficiently from trivial TSP20-TSP50 allows models to better generalize to TSP200 in a zero-shot manner, indicating positive knowledge transfer from small to large graphs. Performance can further improve via rapid finetuning on 1.28 Million TSP200 instances or by Active Search. Within our computational budget, a simple non-learnt *furthest insertion* heuristic still outperforms all models. Precise experimental setup is described in **Appendix A**.

Motivation Scaling end-to-end approaches to practical and real-world instances is still an open question [7] as the training phase of state-of-the-art models on large graphs is extremely time-consuming. For graphs larger than few hundreds of nodes, the gap between GNN-based solvers and simple non-learnt heuristics is especially evident for routing problems like TSP [42, 37].

As an illustration, Figure 1 presents the computational challenge of learning TSP on 200-node graphs (TSP200) in terms of both sample efficiency and wall clock time. Surprisingly, it is difficult to outperform a simple insertion heuristic when directly training on 12.8 Million TSP200 samples for 500 hours on university-scale hardware.

We advocate for an alternative to expensive large-scale training: learning efficiently from trivially small TSP and transferring the learnt policy to larger graphs in a *zero-shot* fashion or via fast finetuning. Thus, identifying promising inductive biases, architectures and learning paradigms that enable such zero-shot generalization to large and more complex instances is a key concern for training practical solvers for real-world problems.

Contributions Towards end-to-end learning of *scale-invariant* TSP solvers, we unify several state-of-the-art architectures and learning paradigms [54, 42, 17, 37] into one experimental pipeline and provide the first principled investigation on zero-shot generalization to large instances. Our findings suggest that learning scale-invariant TSP solvers requires rethinking the status quo of neural combinatorial optimization to explicitly account for generalization:

- The prevalent evaluation paradigm overshadows models' poor generalization capabilities by measuring performance on fixed or trivially small TSP sizes.
- Generalization performance of GNN aggregation functions and normalization schemes benefits from explicit redesigns which account for shifting graph distributions, and can be further boosted by enforcing regularities such as constant graph diameters when defining problems using graphs.
- Autoregressive decoding enforces a sequential inductive bias which improves generalization over non-autoregressive models, but is costlier in terms of inference time.
- Models trained with expert supervision are more amenable to post-hoc search, while reinforcement learning approaches scale better with more computation as they do not rely on labelled data.

Our framework and datasets are available online[3]. Additionally, we use our pipeline to characterize the recent state-of-the-art in deep learning for routing problems and provide new directions for future research [35].

[3] `https://github.com/chaitjo/learning-tsp`

2 Related Work

Neural Combinatorial Optimization In a recent survey, Bengio et al. [7] identified three broad approaches to leveraging machine learning for combinatorial optimization problems: learning alongside optimization algorithms [23, 12, 13], learning to configure optimization algorithms [68, 19], and end-to-end learning to approximately solve optimization problems, *a.k.a.* neural combinatorial optimization [67, 6].

State-of-the-art end-to-end approaches for TSP use Graph Neural Networks (GNNs) [40, 24, 65, 5] and *sequence-to-sequence* learning [62] to construct approximate solutions directly from problem instances. Architectures for TSP can be classified as: (1) autoregressive approaches, which build solutions in a step-by-step fashion [39, 17, 42, 47, 43, 55]; and (2) non-autoregressive models, which produce the solution in one shot [54, 53, 37, 21, 41]. Models can be trained to imitate optimal solvers via supervised learning or by minimizing the length of TSP tours via reinforcement learning [38].

Other classical problems tackled by similar architectures include Vehicle Routing [52, 14], Maximum Cut [39], Minimum Vertex Cover [46], Boolean Satisfiability [60, 77], and Graph Coloring [29]. Using TSP as an illustration, we present a unified pipeline for characterizing neural combinatorial optimization architectures in Section 3.

Notably, TSP has emerged as a challenging testbed for neural combinatorial optimization. Whereas generalization to problem instances larger and more complex than those seen in training has at least partially been demonstrated on non-sequential problems such as SAT, MaxCut, and MVC [46, 60], the same architectures do not show strong generalization for TSP [42, 37].

Combinatorial Optimization and GNNs From the perspective of graph representation learning, algorithmic and combinatorial problems have recently been used to characterize the expressive power of GNNs [58, 11]. An emerging line of work on learning to execute graph algorithms [66, 64] has lead to the development of provably more expressive GNNs [15] and improved understanding of their generalization capability [74, 75]. Towards tackling realistic and large-scale combinatorial problems, this paper aims to quantify the limitations of prevalent GNN architectures and learning paradigms via zero-shot generalization to problems larger than those seen during training.

Novel Applications Advances on classical combinatorial problems have shown promising results in downstream applications to novel or under-studied optimization problems in the physical sciences [25, 61] and computer architecture [49, 56, 50], where the development of exact solvers is expensive and intractable. For example, autoregressive architectures provide a strong inductive bias for device placement optimization problems [51, 80], while non-autoregressive models [9] are competitive with autoregressive approaches [33, 78] for molecule generation tasks.

3 Neural Combinatorial Optimization Pipeline

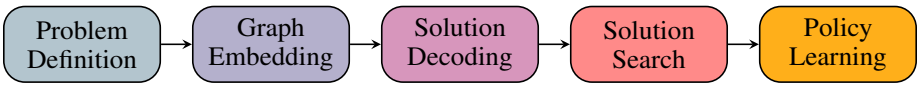
NP-hard problems can be formulated as sequential decision making tasks on graphs due to their highly structured nature. Towards a controlled study of neural combinatorial optimization for TSP, we unify recent ideas [54, 42, 17, 37] via a five stage end-to-end pipeline illustrated in Figure 2. Our discussion focuses on TSP, but the pipeline presented is generic and can be extended to characterize modern architectures for several NP-hard graph problems.

3.1 Problem Definition

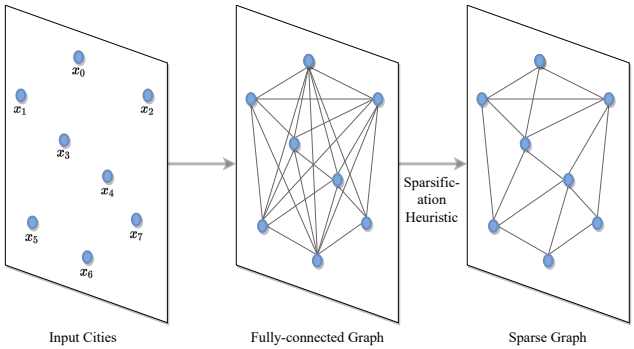
The 2D Euclidean TSP is defined as follows: *"Given a set of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"* Formally, given a fully-connected input graph of n cities (nodes) in the two dimensional unit square $S = \{x_i\}_{i=1}^{n}$ where each $x_i \in [0, 1]^2$, we aim to find a permutation of the nodes π, termed a tour, that visits each node once and has the minimum total length, defined as:

$$L(\pi|s) = \|x_{\pi_n} - x_{\pi_1}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi_i} - x_{\pi_{i+1}}\|_2, \tag{1}$$

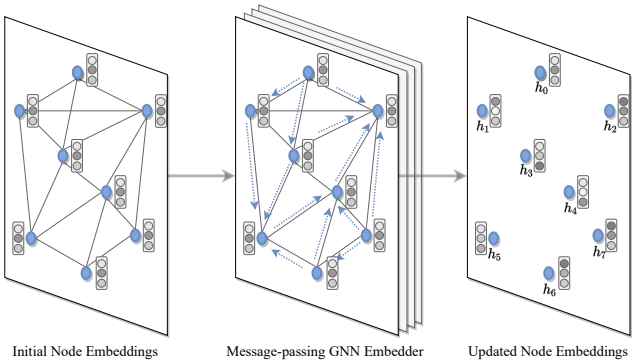
where $\|\cdot\|_2$ denotes the ℓ_2 norm.



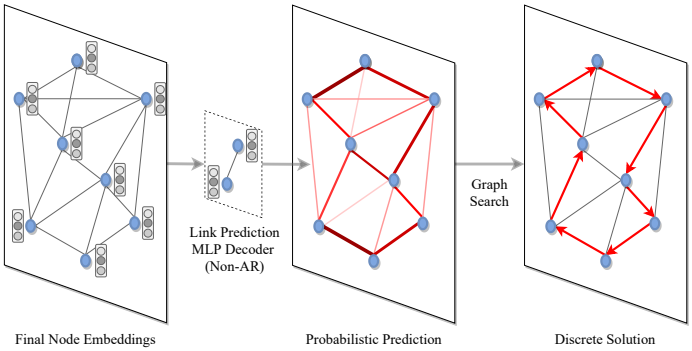
(a) Neural combinatorial optimization pipeline in stages.



(b) **Problem Definition:** TSP is formulated via a fully-connected graph of cities/nodes. The graph can be sparsified via heuristics such as k-nearest neighbors.



(c) **Graph Embedding:** Embeddings for each graph node are obtained using a Graph Neural Network encoder, which builds local structural features.



(d) **Solution Decoding & Search:** Probabilities are assigned to each node for belonging to the solution set, either independent of one-another (*i.e.* Non-autoregressive decoding) or conditionally through graph traversal (*i.e.* Autoregressive decoding). The predicted probabilities are converted into discrete decisions through classical graph search techniques such as greedy search or beam search.

Figure 2: **End-to-end neural combinatorial optimization pipeline:** The entire model in trained end-to-end via imitating an optimal solver (*i.e.* supervised learning) or through minimizing a cost function (*i.e.* reinforcement learning).

Graph Sparsification Classically, TSP is defined on fully-connected graphs, see Figure 2(b). Graph sparsification heuristics based on k-nearest neighbors aim to reduce TSP graphs, enabling models to scale up to large instances where pairwise computation for all nodes is intractable [39] or learn faster by reducing the search space [37]. Notably, problem-specific graph reduction techniques have proven effective for out-of-distribution generalization to larger graphs for other NP-hard problems such as MVC and SAT [46].

Fixed size vs. variable size graphs Most work on learning for TSP has focused on training with a fixed graph size [42, 37], likely due to ease of implementation. Learning from multiple graph sizes naturally enables better generalization within training size ranges, but its impact on generalization to larger TSP instances remains to be analyzed.

3.2 Graph Embedding

A Graph Neural Network (GNN) encoder computes d-dimensional representations for each node in the input TSP graph, see Figure 2(c). At each layer, nodes gather features from their neighbors to represent local graph structure via recursive message passing [24]. Stacking L layers allows the network to build representations from the L-hop neighborhood of each node. Let h_i^ℓ and e_{ij}^ℓ denote respectively the node and edge feature at layer ℓ associated with node i and edge ij. We define the feature at the next layer via an *anisotropic* message passing scheme using an edge gating mechanism [8]:

$$h_i^{\ell+1} = h_i^\ell + \text{ReLU}\Big(\textsc{Norm}\Big(U^\ell h_i^\ell + \textsc{Aggr}_{j\in\mathcal{N}_i}\Big(\sigma(e_{ij}^\ell)\odot V^\ell h_j^\ell\Big)\Big)\Big)\,, \tag{2}$$

$$e_{ij}^{\ell+1} = e_{ij}^\ell + \text{ReLU}\Big(\textsc{Norm}\Big(A^\ell e_{ij}^\ell + B^\ell h_i^\ell + C^\ell h_j^\ell\Big)\Big), \tag{3}$$

where $U^\ell, V^\ell, A^\ell, B^\ell, C^\ell \in \mathbb{R}^{d\times d}$ are learnable parameters, NORM denotes the normalization layer (BatchNorm [32], LayerNorm [4]), AGGR represents the neighborhood aggregation function (SUM, MEAN or MAX), σ is the sigmoid function, and $\odot$ is the Hadamard product. As inputs $h_i^{\ell=0}$ and $e_{ij}^{\ell=0}$, we use d-dimensional linear projections of the node coordinate x_i and the euclidean distance $\|x_i - x_j\|_2$, respectively.

Anisotropic Aggregation We make the aggregation function anisotropic or directional via a dense attention mechanism which scales the neighborhood features $h_j, \forall j \in \mathcal{N}_i$, using edge gates $\sigma(e_{ij})$. Anisotropic and attention-based GNNs such as Graph Attention Networks [65], Transformers [63, 34], and Gated Graph ConvNets [8] have been shown to outperform isotropic Graph ConvNets [40] across several challenging domains [18], including TSP [42, 37].

3.3 Solution Decoding

Non-autoregressive Decoding (NAR) Consider TSP as a link prediction task: each edge may belong/not belong to the optimal TSP solution independent of one another [54]. We define the edge predictor as a two layer MLP on the node embeddings produced by the final GNN encoder layer L, following Joshi et al. [37], see Figure 2(d). For adjacent nodes i and j, we compute the unnormalized edge logits:

$$\hat{p}_{ij} = W_2\Big(\text{ReLU}\ \big(W_1\big(\ [\ h_G,\ h_i^L,\ h_j^L\ ]\ \big)\big)\Big), \quad \text{where}\ \ h_G = \frac{1}{n}\sum_{i=0}^{n} h_i^L, \tag{4}$$

$W_1 \in \mathbb{R}^{3d\times d}, W_2 \in \mathbb{R}^{d\times 2}$, and $[\cdot,\cdot,\cdot]$ is the concatenation operator. The logits $\hat{p}_{ij}$ are converted to probabilities over each edge p_{ij} via a softmax.

Autoregressive Decoding (AR) Although NAR decoders are fast as they produce predictions in one shot, they ignore the sequential ordering of TSP tours. Autoregressive decoders, based on attention [17, 42] or recurrent neural networks [67, 47], explicitly model this sequential inductive bias through step-by-step graph traversal. We follow the attention decoder from Kool et al. [42], which starts from a random node and outputs a probability distribution over its neighbors at each step. Greedy search is used to perform the traversal over n time steps and masking enforces constraints such as not visiting previously visited nodes.

At time step t at node i, the decoder builds a context $\hat{h}_i^C$ for the partial tour $\pi'_{t'}$, generated at time $t' < t$, by packing together the graph embedding h_G and the embeddings of the first and last node

in the partial tour: $\hat{h}_i^C = W_C \left[h_G,\ h^L_{\pi'_{t-1}},\ h^L_{\pi'_1} \right]$, where $W_C \in \mathbb{R}^{3d \times d}$ and learned placeholders are used for $h^L_{\pi'_{t-1}}$ and $h^L_{\pi'_1}$ at $t = 1$. The context $\hat{h}_i^C$ is then refined via a standard Multi-Head Attention (MHA) operation [63] over the node embeddings:

$$h_i^C = \text{MHA}\Big(Q = \hat{h}_i^C, K = \{h_1^L, \dots, h_n^L\}, V = \{h_1^L, \dots, h_n^L\}\Big), \tag{5}$$

where Q, K, V are inputs to the M-headed MHA ($M = 8$). The unnormalized logits for each edge e_{ij} are computed via a final attention mechanism between the context h_i^C and the embedding h_j:

$$\hat{p}_{ij} = \begin{cases} C \cdot \tanh\left(\frac{\left(W_Q h_i^C\right)^T \cdot \left(W_K h_j^L\right)}{\sqrt{d}} \right) & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{6}$$

The $\tanh$ is used to maintain the value of the logits within $[-C, C]$ ($C = 10$) [6]. The logits $\hat{p}_{ij}$ at the current node i are converted to probabilities p_{ij} via a softmax over all edges.

Inductive Biases NAR approaches, which make predictions over edges independently of one-another, have shown strong out-of-distribution generalization for non-sequential problems such as SAT and MVC [46]. On the other hand, AR decoders come with the sequential/tour constraint built-in and are the default choice for routing problems [42]. Although both approaches have shown close to optimal performance on fixed and small TSP sizes under different experimental settings, it is important to fairly compare which inductive biases are most useful for generalization.

3.4 Solution Search

Greedy Search For AR decoding, the predicted probabilities at node i are used to select the edge to travel along at the current step via sampling from the probability distribution p_i or greedily selecting the most probable edge p_{ij}, *i.e.* greedy search. Since NAR decoders directly output probabilities over all edges independent of one-another, we can obtain valid TSP tours using greedy search to traverse the graph starting from a random node and masking previously visited nodes. Thus, the probability of a partial tour π' can be formulated as $p(\pi') = \prod_{j' \sim i' \in \pi'} p_{i'j'}$, where each node j' follows node i'.

Beam Search and Sampling During inference, we can increase the capacity of greedy search via limited width breadth-first beam search, which maintains the b most probable tours during decoding. Similarly, we can sample b solutions from the learnt policy and select the shortest tour among them. Naturally, searching longer, with more sophisticated techniques, or sampling more solutions allows trading off run time for solution quality. However, it has been noted that using large b for search/sampling or local search during inference may overshadow an architecture's inability to generalize [20]. To better understand generalization, we focus on using greedy search and beam search/sampling with small $b = 128$.

3.5 Policy Learning

Supervised Learning Models can be trained end-to-end via imitating an optimal solver at each step (*i.e.* supervised learning). For models with NAR decoders, the edge predictions are linked to the ground-truth TSP tour by minimizing the binary cross-entropy loss for each edge [37]. For AR architectures, at each step, we minimize the cross-entropy loss between the predicted probability distribution over all edges leaving the current node and the next node from the groundtruth tour, following Vinyals et al. [67]. We use teacher-forcing to stabilize training [70].

Reinforcement Learning Reinforcement learning is a elegant alternative in the absence of groundtruth solutions, as is often the case for understudied combinatorial problems. Models can be trained by minimizing problem-specific cost functions (the tour length in the case of TSP) via policy gradient algorithms [6, 42] or Q-Learning [39]. We focus on policy gradient methods due to their simplicity, and define the loss for an instance s parameterized by the model θ as $\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}\left[L(\pi)\right]$, the expectation of the tour length $L(\pi)$, where $p_\theta(\pi|s)$ is the probability distribution from which we sample to obtain the tour $\pi|s$. We use the REINFORCE gradient estimator [69] to minimize $\mathcal{L}$:

$$\nabla\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}\left[\left(L(\pi) - b(s)\right) \nabla \log p_\theta(\pi|s)\right], \tag{7}$$

where the baseline $b(s)$ reduces gradient variance. Our experiments compare standard critic network baselines [6, 17] and the greedy rollout baseline proposed by Kool et al. [42].

4 Experimental Setup

We design controlled experiments to probe the unified pipeline described in Section 3 in order to identify inductive biases, architectures and learning paradigms that promote zero-shot generalization. We focus on learning efficiently from small problem instances (TSP20-50) and measure generalization to a wider range of sizes, including large instances which are intractable to learn from (*e.g.* TSP200). Each experiment starts with a 'base' model configuration and ablates the impact of a specific component of the five-stage pipeline. We aim to fairly compare state-of-the-art ideas in terms of model capacity and training data, and expect models with good inductive biases for TSP to: (1) learn trivially small TSPs without hundreds of millions of training samples and model parameters; and (2) generalize reasonably well across smaller and larger instances than those seen in training.

To quantify 'good' generalization, we additionally evaluate our models against a simple, non-learnt *furthest insertion* heuristic baseline, which constructively builds a partial tour π' by inserting node i between tour nodes $j_1, j_2 \in \pi'$ such that the distance from node i to its nearest node j_1 is maximized. Kool et al. [42] provide a detailed description of insertion heuristic baselines.

Training Datasets We perform ablation studies of each component of the pipeline by training on variable TSP20-50 graphs for rapid experimentation. We also compare to learning from fixed graph sizes up to TSP100. Each TSP instance consist of n nodes sampled uniformly in the unit square $S = \{x_i\}_{i=1}^{n}$ and $x_i \in [0, 1]^2$. In the supervised learning paradigm, we generate a training set of 1,280,000 TSP samples and groundtruth tours using the optimal Concorde solver as an oracle. Models are trained using the Adam optimizer for 10 epochs with a batch size of 128 and a fixed learning rate $1e-4$. For reinforcement learning, models are trained for 100 epochs on 128,000 TSP samples which are randomly generated for each epoch (without optimal solutions) with the same batch size and learning rate. Thus, both learning paradigms see 12,800,000 TSP samples in total. Considering that TSP20-50 are trivial in terms of complexity as they can be solved by simpler non-learnt heuristics, training good solvers at this scale should ideally not require billions of instances.

Model Hyperparameters For models with AR decoders, we use 3 GNN encoder layers followed by the attention decoder head, setting hidden dimension $d = 128$. For NAR models, we use the same hidden dimension and opt for 4 GNN encoder layers followed by the edge predictor. This results in approximately 350,000 trainable parameters for each model, irrespective of decoder type. Unless specified, most experiments use our best model configuration: AR decoding scheme and Graph ConvNet encoder with MAX aggregation and BatchNorm (with batch statistics). All models are trained via supervised learning except when comparing learning paradigms.

Evaluation We compare models on a held-out test set of 25,600 TSPs, consisting of 1,280 samples each of TSP10, TSP20, . . . , TSP200. Our evaluation metric is the optimality gap *w.r.t.* the Concorde solver, *i.e.* the average percentage ratio of predicted tour lengths relative to optimal tour lengths. To compare design choices among identical models, we plot line graphs of the optimality gap as TSP size increases (along with a 99%-ile confidence interval) using beam search with a width of 128. Compared to previous work which evaluated on fixed problem sizes, our protocol identifies not only those models that perform well on training sizes, but also those that generalize better than non-learnt heuristics for large instances which are intractable to train on.

5 Results

5.1 Does learning from variable sizes help generalization?

We train five identical models on fully connected graphs of instances from TSP20, TSP50, TSP100, TSP200 and variable TSP20-50. The line plots of optimality gap across TSP sizes in Figure 3 indicates that learning from variable TSP sizes helps models retain performance across the range of graph sizes seen during training (TSP20-50). Variable graph training compared to training solely on the maximum sized instances (TSP50) leads to marginal gains on small instances but, somewhat counter-intuitively, does not enable better generalization to larger problems. Learning from small TSP20 is unable to generalize to large sizes while TSP100 models generalize poorly to trivially easy sizes, suggesting that the prevalent protocol of evaluation on training sizes [42, 37] overshadows brittle out-of-distribution performance.

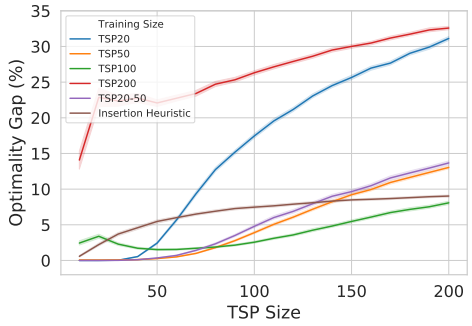


Figure 3: **Learning from various TSP sizes.** The prevalent protocol of evaluation on training sizes overshadows brittle out-of-distribution performance to larger and smaller graphs.

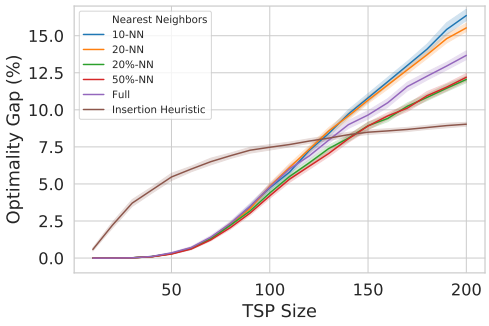


Figure 4: **Impact of graph sparsification.** Maintaining a constant graph diameter across TSP sizes leads to better generalization on larger problems than using full graphs.

Training on TSP200 graphs is intractable within our computational budget, see Figure 1. TSP100 is the only model which generalizes better to large TSP200 than the non-learnt baseline. However, training on TSP100 can also be prohibitively expensive: one epoch takes approximately 8 hours (TSP100) vs. 2 hours (TSP20-50) (details in Appendix B). For rapid experimentation, we train efficiently on variable TSP20-50 for the rest of our study.

5.2 What is the best graph sparsification heuristic?

Figure 4 compares full graph training to the following heuristics: (1) **Fixed node degree** across graph sizes, via connecting each node in TSPn to its k-nearest neighbors, enabling GNN encoder layers to specialize to constant degree k; and (2) **Fixed graph diameter** across graph sizes, via connecting each node in TSPn to its $n \times k\%$-nearest neighbors, ensuring that the same number of message passing steps are required to diffuse information across both small and large graphs.

Although both sparsification techniques lead to faster convergence on training instance sizes (not shown), we find that only approach (2) leads to better generalization on larger problems than using full graphs. Consequently, all further experiments use approach (2) to operate on sparse 20%-nearest neighbors graphs. Our results also suggest that developing more principled problem definition and graph reduction techniques beyond simple k-nearest neighbors for augmenting learning-based approaches may be a promising direction.

5.3 What is the relationship between GNN aggregation functions and normalization layers?

In Figure 5, we compare identical models with anisotropic SUM, MEAN and MAX aggregation functions. As baselines, we consider the Transformer encoder on full graphs [17, 42] as well as a structure-agnostic MLP on each node, which can be instantiated by not using any aggregation function in Eq.(2), *i.e.* $h_i^{\ell+1} = h_i^{\ell} + \text{ReLU}\left(\text{NORM}\left(U^{\ell} h_i^{\ell}\right)\right)$.

We find that the choice of GNN aggregation function does not have an impact when evaluating models within the training size range TSP20-50. As we tackle larger graphs, GNNs with aggregation functions that are agnostic to node degree (MEAN and MAX) are able to outperform Transformers and MLPs. Importantly, the theoretically more expressive SUM aggregator [73] generalizes worse than structure-agnostic MLPs, as it cannot handle the distribution shift in node degree and neighborhood statistics across graph sizes, leading to unstable or exploding node embeddings [66]. We use the MAX aggregator in further experiments, as it generalizes well for both AR and NAR decoders.

We also experiment with the following normalization schemes: (1) standard BatchNorm which learns mean and variance from training data, as well as (2) BatchNorm with batch statistics; and (3) LayerNorm, which normalizes at the embedding dimension instead of across the batch. Figure 6 indicates that BatchNorm with batch statistics and LayerNorm are able to better account for changing statistics across different graph sizes. Standard BatchNorm generalizes worse than not doing any normalization, thus our other experiments use BatchNorm with batch statistics.

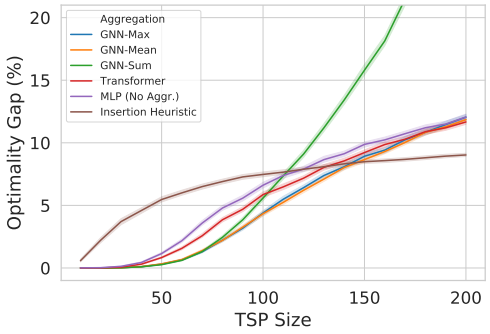


Figure 5: **Impact of GNN aggregation functions.** For larger graphs, aggregators that are agnostic to node degree (MEAN, MAX) are able to outperform theoretically more expressive aggregators.

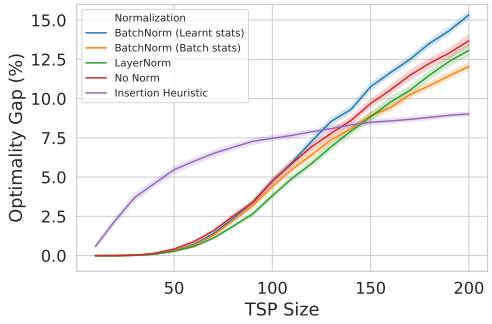


Figure 6: **Impact of normalization schemes.** Modifying BatchNorm to account for changing graph statistics across graph sizes leads to better generalization.

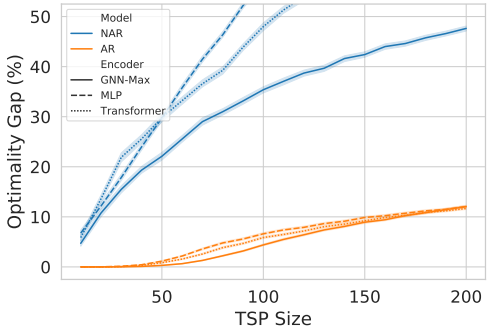


Figure 7: **Comparing AR and NAR decoders.** Sequential AR decoding is a powerful inductive bias for TSP as it enables significantly better generalization, even in the absence of graph structure (MLP encoders).

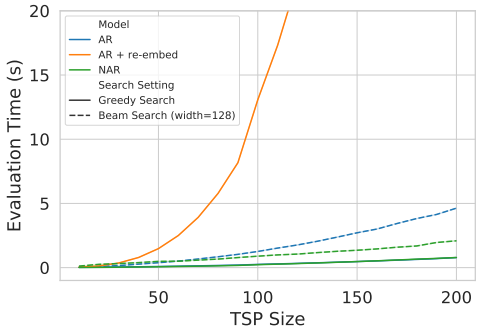


Figure 8: **Inference time for various decoders.** One-shot NAR decoding is significantly faster than sequential AR, especially when re-embedding the graph at each decoding step [39].

We further dissect the relationship between graph representations and normalization in Appendix D, confirming that poor performance on large graphs can be explained by unstable representations due to the choice of aggregation and normalization schemes. Using MAX aggregators and BatchNorm with batch statistics are temporary hacks to overcome the failure of the current architectural components. Overall, our results suggest that inference beyond training sizes will require the development of expressive GNN mechanisms that are able to leverage global graph topology [22] while being invariant to distribution shifts in terms of node degree and other graph statistics [45].

5.4 Which decoder has a better inductive bias for TSP?

Figure 7 compares NAR and AR decoders for identical models. To isolate the impact of the decoder's inductive bias without the inductive bias imposed by GNNs, we also show Transformer encoders on full graphs as well as structure-agnostic MLPs. Within our experimental setup, AR decoders are able to fit the training data as well as generalize significantly better than NAR decoders, indicating that sequential decoding is powerful for TSP even without graph information.

Conversely, NAR architectures are a poor inductive bias as they require significantly more computation to perform competitively to AR decoders. For instance, recent models [54, 37] used more than 30 GNN layers with over 10 Million parameters. We believe that such overparameterized networks are able to memorize all patterns for small TSP training sizes [79], but the learnt policy is unable to generalize beyond training graph sizes. At the same time, when compared fairly within the same experimental settings, NAR decoders are significantly faster than AR decoders described in Section 3.3 as well as those which re-embed the graph at each decoding step [39], see Figure 8.

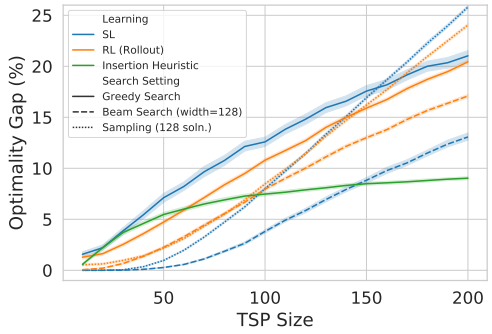


Figure 9: **Comparing solution search settings.** Under greedy decoding, RL demonstrates better performance and generalization. Conversely, SL models improve over their RL counterparts when performing beam search or sampling.

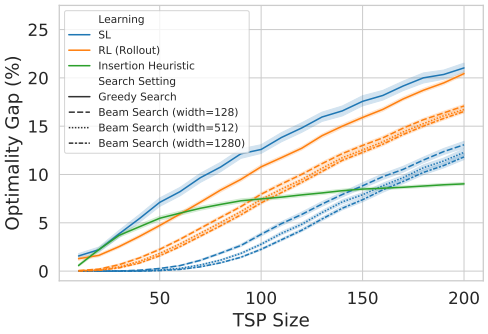


Figure 10: **Impact of increasing beam width.** Teacher-forcing during SL leads to poor generalization under greedy decoding, but makes the probability distribution more amenable to beam search.

5.5 How do learning paradigms impact the search phase?

Identical models are trained via supervised learning (SL) and reinforcement learning (RL)[4]. Figure 9 illustrates that, when using greedy decoding during inference, RL models perform better on the training size as well as on larger graphs. Conversely, SL models improve over their RL counterparts when performing beam search or sampling.

In Appendix C, we find that the rollout baseline, which encourages better greedy behaviour, leads to the model making very confident predictions about selecting the next node at each decoding step, even out of training size range. In contrast, SL models are trained with teacher forcing, *i.e.* imitating the optimal solver at each step instead of using their own prediction. This results in less confident predictions and poor greedy decoding, but makes the probability distribution more amenable to beam search and sampling, as shown in Figure 10. Our results advocate for tighter coupling between the training and inference phase of learning-driven TSP solvers, mirroring recent findings in generative models for text [27].

5.6 Which learning paradigm scales better?

Our experiments till this point have focused on isolating the impact of various pipeline components on zero-shot generalization under limited computation. At the same time, recent results in natural language processing have highlighted the power of large scale pre-training for effective transfer learning [57]. To better understand the impact of learning paradigms when scaling computation, we double the model parameters (up to 750,000) and train on tens times more data (12.8M samples) for AR architectures. We monitor optimality gap on the training size range (TSP20-50) as well as a larger size (TSP100) vs. the number of training samples.

In Figure 11, we see that increasing model capacity leads to better learning. Notably, RL models, which train on unique randomly generated samples throughout, are able to keep improving their performance within as well as outside of training size range as they see more samples. On the other hand, SL is bottlenecked by the need for optimal groundtruth solutions: SL models iterate over the same 1.28M unique labelled samples and stop improving at a point. Beyond favorable inductive biases, distributed and sample-efficient RL algorithms [59] may be a key ingredient for learning from and scaling up to larger TSPs beyond tens of nodes.

6 Recent Case Studies and Future Work

Since the initial publication of this work [36], deep learning for routing problems has received considerable attention from the research community [71, 43, 21, 16, 41, 72, 48, 55, 30]. In this section and in an associated blogpost [35], we highlight recent advances, characterize them using

[4]For RL, we show the greedy rollout baseline. Critic baseline results are available in Appendix E

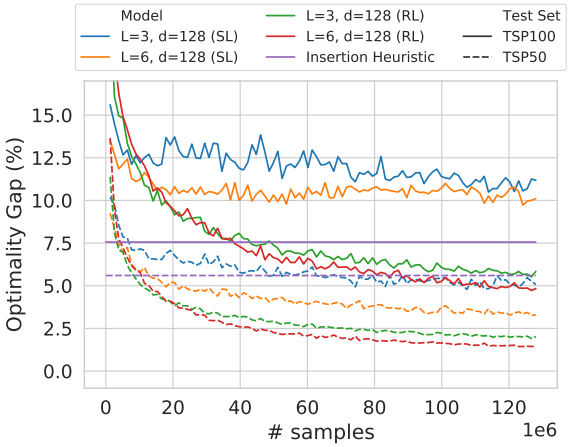


Figure 11: **Scaling computation and parameters for SL and RL-trained models.** All models are trained on TSP20-50. We plot optimality gap on 1,280 held-out samples of both TSP50 (performance on training size) and TSP100 (out-of-distribution generalization) under greedy decoding. Note that SL models are less amenable than RL models to greedy search. RL models are able to keep improving their performance within as well as outside of training size range with more data. On the other hand, SL performance is bottlenecked by the need for optimal groundtruth solutions.

the unified pipeline presented in Figure 2, and provide future research directions, with a focus on improving generalization to large-scale and real-world instances. As a reminder, the unified neural combinatorial optimization pipeline consists of: (1) Problem Definition → (2) Graph Embedding → (3) Solution Decoding → (4) Solution Search → (5) Policy Learning.

Leveraging equivariance and symmetries The autoregressive Attention Model [42] sequentially constructs TSP tours as permutations of cities, but does not consider the underlying symmetries of routing problems.

Kwon et al. [43] consider invariance to the starting city in constructive heuristics: They propose to train the Attention Model with a new reinforcement learning algorithm (innovating on box 5 in Figure 2(a)) which exploits the existence of multiple optimal tour permutations. Similarly, Ouyang, Wang, et al. [55] consider invariance with respect to rotations, reflections, and translations (Euclidean symmetry group) of the input cities: They propose a constructive approach similar to Attention Model while ensuring invariance by performing data augmentation during the problem definition stage (Figure 2(a), box 1) and using relative coordinates during graph encoding (Figure 2(a), box 2). Their approach shows particularly strong results on zero-shot generalization from random instances to the real-world TSPLib benchmark suite.

Future work may follow the Geometric Deep Learning blueprint [10] by designing models that respect the symmetries and inductive biases that govern the data. As routing problems are embedded in euclidean coordinates and the routes are cyclical, incorporating these contraints directly into the architectures or learning paradigms may be a principled approach to improving generalization to large-scale instances greater than those seen during training.

Improved graph search algorithms Several papers have proposed to improve the one-shot non-autoregressive approach of Joshi et al. [37] by retaining the same GNN encoder (Figure 2(a), box 2) while replacing the graph search component of the pipeline (Figure 2(a), box 4) with more powerful and flexible algorithms, *e.g.* Dynamic Programming [41] or Monte-Carlo Tree Search (MCTS) [21].

Notably, the GNN + MCTS framework of Fu et al. [21] shows that the NAR approach can generalize to TSPs with up to 1000 nodes. They ensure that the predictions of the GNN encoder generalize from small to large TSP by updating the problem definition (Figure 2(a), box 1): large problem instances are represented as many smaller sub-graphs which are of the same size as the training graphs for the GNN, and then merge the GNN edge predictions before performing MCTS.

Overall, this line of work suggests that stronger coupling between the design of both the neural and symbolic/search components of models is essential for out-of-distribution generalization.

Learning within local search heuristics Recent work has explored an alternative to constructive AR and NAR decoding schemes which involves learning to iteratively improve (sub-optimal) solutions or learning to perform local search [71, 16, 72, 48, 30]. Since deep learning is used to guide decisions within classical search algorithms (which are designed to work regardless of problem scale), this approach implicitly leads to better zero-shot generalization to larger problem instances compared to constructive approaches studied in our work. In particular, NeuroLKH [72] uses GNNs to improve the Lin-Kernighan-Helsgaun algorithm and demonstrates strong zero-shot generalization to TSP with 5000 nodes as well as across TSPLib instances.

A limitation of this line of work is the need for hand-designed local search heuristics, which may be missing for understudied problems. On the other hand, constructive approaches are comparatively easier to adapt to new problems by enforcing constraints during the solution decoding and search procedure (Figure 2(a), box 4).

Learning Paradigms that promote generalization Future work could look at novel learning paradigms which explicitly focus on generalization beyond supervised and reinforcement learning. For *e.g.*, this work explored zero-shot generalization to larger problems, but the logical next step is to fine-tune the model on a small number of larger problem instances [28]. Thus, it will be interesting to explore fine-tuning/generalization as a meta-learning problem, wherein the goal is to train model parameters specifically for fast adaptation and fine-tuning to new data distributions and problem sizes.

Another interesting direction could explore tackling understudied routing problems with challenging constraints via multi-task pre-training on well-known routing problems such as TSP and CVPR, followed by problem-specific finetuning. Similar to language modelling as a pre-training objective in NLP [57], the goal of pre-training for routing would be to learn generally useful neural network representations that can transfer well to novel routing problems.

7 Conclusion

Learning-driven solvers for combinatorial problems such as the Travelling Salesperson Problem have shown promising results for trivially small instances up to a few hundred nodes. However, scaling fully *end-to-end* deep learning approaches to real-world instances is still an open question as training on large graphs is extremely time-consuming and challenging to learn from.

This paper advocates for an alternative to expensive large-scale training: training models efficiently on trivially small TSP and transferring the learnt policy to larger graphs in a *zero-shot* fashion or via fast fine-tuning. Thus, identifying promising inductive biases, architectures and learning paradigms that enable such zero-shot generalization to large and more complex instances is a key concern for tackling real-world combinatorial problems.

We perform the first principled investigation into zero-shot generalization for learning large scale TSP, unifying state-of-the-art architectures and learning paradigms into one experimental pipeline for neural combinatorial optimization. Our findings suggest that key design choices such as GNN layers, normalization schemes, graph sparsification, and learning paradigms need to be explicitly re-designed to consider out-of-distribution generalization. Additionally, we use our unified pipeline to characterize recent advances in deep learning for routing problems and provide new directions to stimulate future research.

Acknowledgements

We would like to thank R. Anand, X. Bresson, V. Dwivedi, A. Ferber, E. Khalil, W. Kool, R. Levie, A. Prouvost, P. Veličković and the anonymous reviewers for helpful comments and discussions.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint*, 2016. (Cited on page 18)

[2] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde tsp solver, 2006. (Cited on page 1)

[3] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The traveling salesman problem: a computational study*. 2006. (Cited on page 1)

[4] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint*, 2016. (Cited on page 5)

[5] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint*, 2018. (Cited on page 1, 3)

[6] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. In *ICLR*, 2017. (Cited on page 1, 2, 3, 6, 16, 20)

[7] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020. (Cited on page 1, 2, 3)

[8] X. Bresson and T. Laurent. An experimental study of neural networks for variable graphs. In *ICLR Workshop*, 2018. (Cited on page 5)

[9] X. Bresson and T. Laurent. A two-step graph convolutional decoder for molecule generation. In *NeurIPS Workshop on Machine Learning and the Physical Sciences*, 2019. (Cited on page 3)

[10] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint*, 2021. (Cited on page 11)

[11] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković. Combinatorial optimization and reasoning with graph neural networks. In *IJCAI*, 2021. (Cited on page 3)

[12] Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *AAAI*, 2019. (Cited on page 3)

[13] F. Chalumeau, I. Coulon, Q. Cappart, and L.-M. Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. In *CPAIOR*, 2021. (Cited on page 3)

[14] X. Chen and Y. Tian. Learning to perform local rewriting for combinatorial optimization. In *NeurIPS*, 2019. (Cited on page 3)

[15] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. Principal neighbourhood aggregation for graph nets. In *NeurIPS*, 2020. (Cited on page 3)

[16] P. R. d. O. da Costa, J. Rhuggenaath, Y. Zhang, and A. Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In *Asian Conference on Machine Learning*, 2020. (Cited on page 10, 12)

[17] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the tsp by policy gradient. In *CPAIOR*, 2018. (Cited on page 2, 3, 5, 6, 8, 20)

[18] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *arXiv preprint*, 2020. (Cited on page 5)

[19] A. Ferber, B. Wilder, B. Dilkina, and M. Tambe. Mipaal: Mixed integer program as a layer. In *AAAI*, 2020. (Cited on page 3)

[20] A. François, Q. Cappart, and L.-M. Rousseau. How to evaluate machine learning approaches for combinatorial optimization: Application to the travelling salesman problem. *arXiv preprint*, 2019. (Cited on page 6)

[21] Z.-H. Fu, K.-B. Qiu, and H. Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. In *AAAI*, 2021. (Cited on page 3, 10, 11)

[22] V. K. Garg, S. Jegelka, and T. Jaakkola. Generalization and representational limits of graph neural networks. In *ICML*, 2020. (Cited on page 9)

[23] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *NeurIPS*, 2019. (Cited on page 3)

[24] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017. (Cited on page 1, 3, 5)

[25] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 2018. (Cited on page 3)

[26] A. Hermans, L. Beyer, and B. Leibe. In defense of the triplet loss for person re-identification. *arXiv preprint*, 2017. (Cited on page 18)

[27] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *ICLR*, 2020. (Cited on page 10)

[28] A. Hottung, Y.-D. Kwon, and K. Tierney. Efficient active search for combinatorial optimization problems. *arXiv preprint*, 2021. (Cited on page 12)

[29] J. Huang, M. Patwary, and G. Diamos. Coloring big graphs with alphagozero. *arXiv preprint*, 2019. (Cited on page 3)

[30] B. Hudson, Q. Li, M. Malencia, and A. Prorok. Graph neural network guided local search for the traveling salesperson problem. *arXiv preprint*, 2021. (Cited on page 10, 12)

[31] G. O. Inc. Gurobi optimizer reference manual. *URL* `http://www.gurobi.com`, 2015. (Cited on page 21)

[32] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint*, 2015. (Cited on page 5)

[33] W. Jin, R. Barzilay, and T. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *ICML*, 2018. (Cited on page 3)

[34] C. Joshi. Transformers are graph neural networks. *The Gradient*, 2020. (Cited on page 5)

[35] C. K. Joshi and R. Anand. Recent advances in deep learning for routing problems. In *ICLR Blog Track*, 2022. (Cited on page 2, 10)

[36] C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent. Learning tsp requires rethinking generalization. In *International Conference on Principles and Practice of Constraint Programming*, 2021. (Cited on page 10)

[37] C. K. Joshi, T. Laurent, and X. Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint*, 2019. (Cited on page 2, 3, 5, 6, 7, 9, 11, 20)

[38] C. K. Joshi, T. Laurent, and X. Bresson. On learning paradigms for the travelling salesman problem. *NeurIPS Graph Representation Learning Workshop*, 2019. (Cited on page 3)

[39] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *NeurIPS*, 2017. (Cited on page 1, 3, 5, 6, 9, 16, 17)

[40] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017. (Cited on page 1, 3, 5)

[41] W. Kool, H. van Hoof, J. Gromicho, and M. Welling. Deep policy dynamic programming for vehicle routing problems. *arXiv preprint*, 2021. (Cited on page 3, 10, 11)

[42] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *ICLR*, 2019. (Cited on page 2, 3, 5, 6, 7, 8, 11, 16, 17, 20)

[43] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min. Pomo: Policy optimization with multiple optima for reinforcement learning. In *NeurIPS*, 2020. (Cited on page 3, 10, 11)

[44] J. K. Lenstra and A. R. Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 1975. (Cited on page 1)

[45] R. Levie, M. M. Bronstein, and G. Kutyniok. Transferability of spectral graph convolutional neural networks. *arXiv preprint*, 2019. (Cited on page 9)

[46] Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *NeurIPS*, 2018. (Cited on page 1, 3, 5, 6, 16)

[47] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. In *AAAI Workshop on Deep Learning on Graphs*, 2020. (Cited on page 3, 5)

[48] Y. Ma, J. Li, Z. Cao, W. Song, L. Zhang, Z. Chen, and J. Tang. Learning to iteratively solve routing problems with dual-aspect collaborative transformer. In *NeurIPS*, 2021. (Cited on page 10, 12)

[49] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *ACM Special Interest Group on Data Communication*, 2019. (Cited on page 3)

[50] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 2021. (Cited on page 1, 3)

[51] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. In *ICML*, 2017. (Cited on page 3)

[52] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. In *NeurIPS*, 2018. (Cited on page 3)

[53] A. Nowak, D. Folqué, and J. B. Estrach. Divide and conquer networks. In *ICLR*, 2018. (Cited on page 3)

[54] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint*, 2017. (Cited on page 2, 3, 5, 9, 20)

[55] W. Ouyang, Y. Wang, P. Weng, and S. Han. Generalization in deep rl for tsp problems via equivariance and local search. *arXiv preprint*, 2021. (Cited on page 3, 10, 11)

[56] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals. Regal: Transfer learning for fast optimization of computation graphs. *arXiv preprint*, 2019. (Cited on page 3)

[57] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 2020. (Cited on page 10, 12)

[58] R. Sato, M. Yamada, and H. Kashima. Approximation ratios of graph neural networks for combinatorial problems. In *NeurIPS*, 2019. (Cited on page 3)

[59] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017. (Cited on page 10)

[60] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a sat solver from single-bit supervision. In *ICLR*, 2019. (Cited on page 1, 3, 16)

[61] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 2020. (Cited on page 1, 3)

[62] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014. (Cited on page 3)

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017. (Cited on page 5, 6)

[64] P. Veličković and C. Blundell. Neural algorithmic reasoning. *Patterns*, 2021. (Cited on page 3)

[65] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. *ICLR*, 2018. (Cited on page 1, 3, 5)

[66] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. Neural execution of graph algorithms. In *ICLR*, 2020. (Cited on page 3, 8)

[67] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *NeurIPS*, 2015. (Cited on page 1, 3, 5, 6)

[68] B. Wilder, B. Dilkina, and M. Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI*, 2019. (Cited on page 3)

[69] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992. (Cited on page 6)

[70] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989. (Cited on page 6)

[71] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim. Learning improvement heuristics for solving routing problem. *IEEE Transactions on Neural Networks and Learning Systems*, 2021. (Cited on page 10, 12)

[72] L. Xin, W. Song, Z. Cao, and J. Zhang. Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. In *NeurIPS*, 2021. (Cited on page 10, 12)

[73] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019. (Cited on page 8)

[74] K. Xu, J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka. What can neural networks reason about? In *ICLR*, 2019. (Cited on page 3)

[75] K. Xu, J. Li, M. Zhang, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *ICLR*, 2020. (Cited on page 3)

[76] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*, 2018. (Cited on page 19)

[77] E. Yolcu and B. Poczos. Learning local search heuristics for boolean satisfiability. In *NeurIPS*, 2019. (Cited on page 3)

[78] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS*, 2018. (Cited on page 3)

[79] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. In *ICLR*, 2017. (Cited on page 9, 20)

[80] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. C. Ma, Q. Xu, M. Zhong, H. Liu, A. Goldie, A. Mirhoseini, et al. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint*, 2019. (Cited on page 3)

A Additional Context for Figure 1

Experimental Setup In Figure 1, we illustrate the computational challenges of learning large scale TSP by comparing three identical models trained on 12.8 Million TSP instances via reinforcement learning. Our experimental setup largely follows Section 4. All models use identical configurations: autoregressive decoding and Graph ConvNet encoder with MAX aggregation and LayerNorm. The TSP20-50 model is trained using the greedy rollout baseline [42] and the Adam optimizer with batch size 128 and learning rate $1e-4$. Direct training, active search and finetuning on TSP200 samples is done using learning rate $1e-5$, as we found larger learning rates to be unstable. During active search and finetuning, we use an exponential moving average baseline, as recommended by Bello et al. [6].

Furthest Insertion Baseline We characterize 'good' generalization across our experiments by the well-known *furthest insertion* heuristic, which constructively builds a solution/partial tour π' by inserting node i between tour nodes $j_1, j_2 \in \pi'$ such that the distance from node i to its nearest tour node j_1 is maximized.

We motivate our work by showing that learning from large TSP200 is intractable on university-scale hardware, and that efficient pre-training on trivial TSP20-50 enables models to better generalize to TSP200 in a zero-shot manner. Within our computational budget, furthest insertion still outperforms our best models. At the same time, we are not claiming that it is *impossible* to outperform insertion heuristics with current approaches: reinforcement learning-driven approaches will only continue to improve performance with more computation and training data. We want to use simple non-learnt baselines to motivate the development of better architectures, learning paradigms and evaluation protocols for neural combinatorial optimization.

Routing Problems and Generalization It is worth mentioning why we chose to study TSP in particular. Firstly, TSP has stood the test of time in terms of relevance and continues to serve as an engine of discovery for general purpose techniques in applied mathematics.

TSP and associated routing problems have also emerged as a challenging testbed for learning-driven approaches to combinatorial optimization. Whereas generalization to problem instances larger and more complex than those seen in training has at least partially been demonstrated on non-sequential problems such as SAT, MaxCut, MVC [39, 46, 60][5], the same architectures do not show strong

[5] It is worth noting that classical algorithmic and symbolic components such as graph reduction, sophisticated tree search as well as post-hoc local search have been pivotal and complementary to GNNs in enabling such generalization.

generalization for TSP. For *e.g.*, furthest insertion outperforms or is competitive with state-of-the-art approaches for TSP above tens of nodes, see Figure D.1.(e, f) from Khalil et al. [39] or Figure 5 from Kool et al. [42], despite using more computation and data than our controlled study.

B Hardware and Timings

Fairly timing research code can be difficult due to differences in libraries used, hardware configurations and programmer skill. In Table 1, we report approximate total training time and inference time across TSP sizes for the model setup described in Section 4. All experiments were implemented in PyTorch and run on an Intel Xeon CPU E5-2690 v4 server and four Nvidia 1080Ti GPUs. Four experiments were run on the server at any given time (each using a single GPU). Training time may vary based on server load, thus we report the lowest training time across several runs in Table 1.

Table 1: Approximate training time (12.8M samples) and inference time (1,280 samples) across TSP sizes and search settings for SL and RL-trained models. *GS*: Greedy search, *BS128*: beam search with width 128, *S128*: sampling 128 solutions. RL training uses the rollout baseline and timing includes the time taken to update the baseline after each 128,000 samples.

Graph Size	**Training Time**		**Inference Time**		
	SL	**RL**	**GS**	**BS128**	**S128**
TSP20	4h 24m	8h 02m	2.62s	7.06s	63.37s
TSP20-50	9h 49m	15h 47m	-	-	-
TSP50	16h 11m	40h 29m	7.45s	29.09s	86.48s
TSP100	68h 34m	108h 30m	19.04s	98.26s	180.30s
TSP200	-	495h 55m	54.88s	372.09s	479.37s

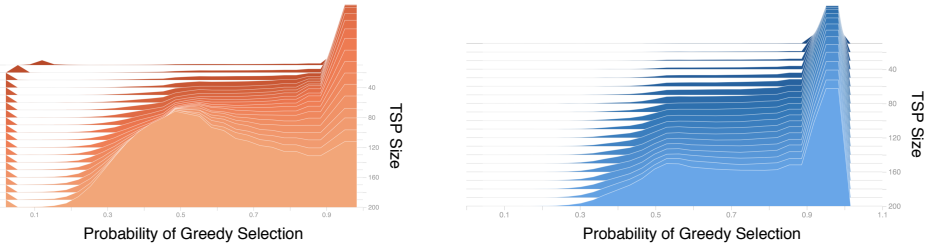
C Learning Paradigms and Amenity to Search

Figure 10 demonstrate that SL models are more amenable to beam search and sampling, but are outperformed by RL-rollout models under greedy search. In Figure 12, we investigate the impact of learning paradigms on probability distributions by plotting histograms of the probabilities of greedy selections during inference across TSP sizes for identical models trained with SL and RL. We find that the rollout baseline, which encourages better greedy behaviour, leads to the model making very confident predictions about selecting the next node at each decoding step, even beyond training size range. In contrast, SL models are trained with teacher forcing, *i.e.* imitating the optimal solver at each step instead of using their own prediction. This results in less confident predictions and poor greedy decoding, but makes the probability distribution more amenable to beam search and sampling techniques.

We understand this phenomenon as follows: More confident predictions (Figure 12b) do not automatically imply better solutions. However, sampling repeatedly or maintaining the top-b most probable solutions from such distributions is likely to contain very similar tours. On the other hand, less sharp distributions (Figure 12a) are likely to yield more diverse tours with increasing b. This may result in comparatively better optimality gap, especially for TSP sizes larger than those seen in training.

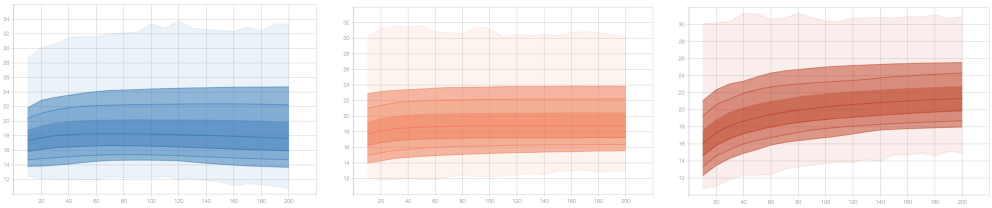
D Visualizing Node and Graph Embedding Spaces

Our results in Section 5.3 suggest that inference beyond training sizes requires the development of GNN architectures and normalization layers that are both expressive as well as invariant to distribution shifts. We explore how node and graph embeddings for TSP graphs evolve across training distribution (TSP20-50) and beyond (up to TSP200) through visualizing the statistics of the embedding spaces. Intuitively, constructing TSP tours involves decisions which are not just locally optimal, but also optimal *w.r.t* some global graph structure. Thus, node embeddings represent *local* information while graph embeddings, which are conventionally computed as the mean of node embeddings, provide *global* structural information.



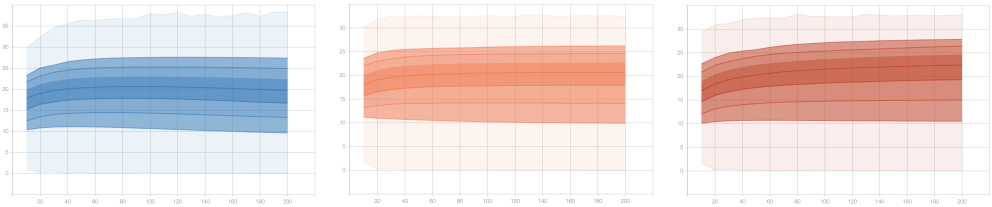
(a) Supervised Learning (b) Reinforcement Learning

Figure 12: Histograms of greedy selection probabilities (x-axis) across TSP sizes (y-axis).



(a) *GNN-Sum* (b) *GNN-Max* (c) *GNN-Max + learnt BN*

Figure 13: Distribution plots of **node** embedding ℓ_2 norms (y-axis) across TSP sizes (x-axis).



(a) *GNN-Sum* (b) *GNN-Max* (c) *GNN-Max + learnt BN*

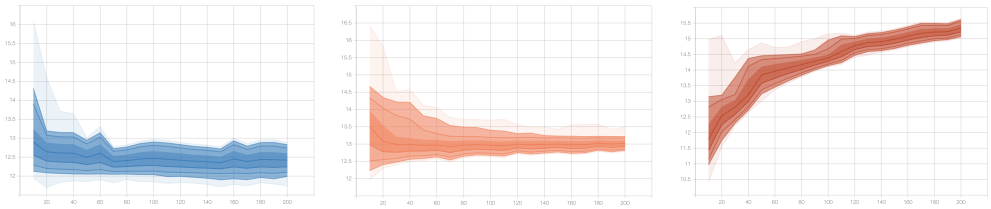
Figure 14: Distribution plots of **node** embedding pair-wise distances (y-axis) across TSP sizes (x-axis).

We utilize distribution plots to study the variation in embedding statistics[6] of three identical models: (1) ***GNN-Max***, which represents our best model configuration from Section 5: autoregressive decoding, Graph ConvNet encoder with MAX aggregation and BatchNorm with batch statistics; (2) ***GNN-Sum***, which uses SUM aggregation for the Graph ConvNet and shows comparatively poor generalization beyond training size, see Figure 5; and (3) ***GNN-Max + learnt BN***, which uses standard BatchNorm, *i.e.* learns statistics from the training data, and also shows comparatively poor generalization, see Figure 6.

We draw upon work in learning embeddings for computer vision [26] to characterize embedding spaces across TSP sizes according to: (1) **magnitudes**, denoted by ℓ_2 norms, indicating whether embeddings are shrinking to one magnitude or expanding outwards as TSP size increases; and (2) **pair-wise distances**, which tells us how well-separated the embedding are, or whether they are pulled apart/towards each other as TSP size increases.

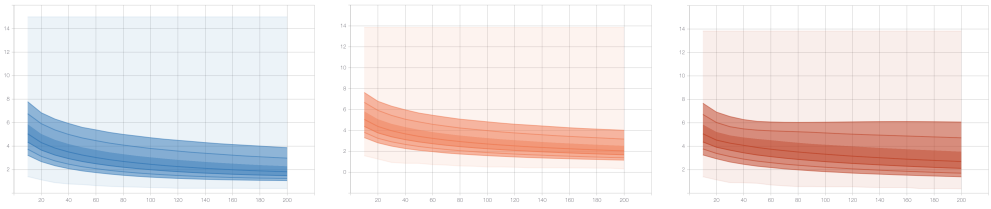
Node Embedding Space In Figures 13 and 14, we see that *GNN-Max* leads to the most stable node embedding norms and pair-wise distances (which are calculated at an intra-graph level) across TSP sizes. On the other hand, *GNN-Sum* and *GNN-Max + learnt BN* lead to fluctuating and monotonically increasing embedding norms as size increases, *e.g.* compare Figure 13b and Figure 13c. Clearly, maintaining similar distributions for node embeddings across graph sizes indicates that the GNN is building meaningful representations of local structure, or, at the very least, does not break down for large graphs. This enables better generalization, as the decoder has lower chances of encountering embeddings which are statistically different than those seen during training.

[6]Distribution plots show 0, 5, 50, 95, and 100-percentiles for embedding statistics at various TSP sizes, thus visualizing how the statistics changes with problem scale (implemented via TensorBoard [1]).



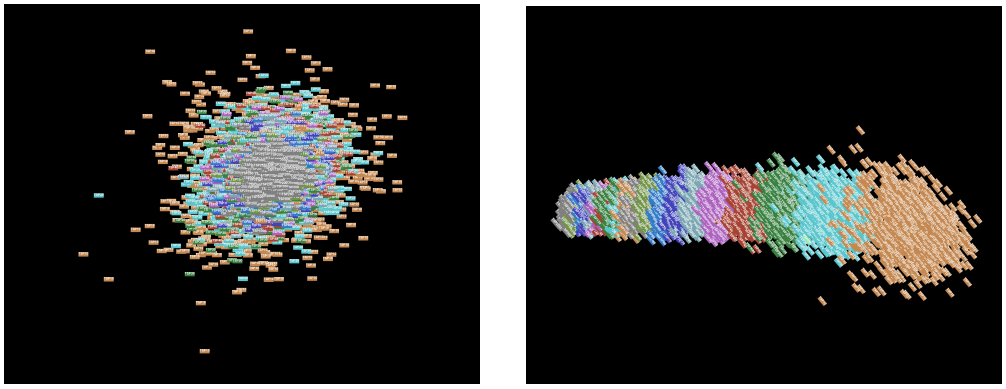
(a) *GNN-Sum* (b) *GNN-Max* (c) *GNN-Max + learnt BN*

Figure 15: Distribution plots of **graph** embedding ℓ_2 norms (y-axis) across TSP sizes (x-axis).



(a) *GNN-Sum* (b) *GNN-Max* (c) *GNN-Max + learnt BN*

Figure 16: Distribution plots of **graph** embedding pair-wise distances (y-axis) across TSP sizes (x-axis).



(a) *GNN-Max* (b) *GNN-Max + learnt BN*

Figure 17: 2D PCA of graph embedding spaces. Colors represent TSP instance sizes, *e.g.* orange: TSP10, teal: TSP20, pink: TSP50, dark grey: TSP200.

Graph Embedding Space Figures 15 and 16 indicate that the graph embedding space is shrinking towards a single magnitude and moving closer as graph size increases. Interestingly, with standard BatchNorm, the graph embedding magnitude monotonically increases with graph size to ranges beyond those for training graphs. On the other hand, using batch statistics for BatchNorm, as done in *GNN-Max* and *GNN-Sum*, leads to graph embedding magnitudes converging to a single value which is within the range of values for training graphs, thus enabling better generalization. *E.g.* compare Figure 15b and Figure 15c.

We can further visualize this phenomenon through 2D Principal Component Analysis (PCA) plots of graph embedding spaces for *GNN-Max* and *GNN-Max + learnt BN* models, see Figures 17a and 17b. In both cases, the graph embeddings at larger sizes have very similar magnitudes and are extremely close to each other, indicating that the model is unable to differentiate among different graphs. Thus, decoders currently lack good global structural context. Investigating better graph embeddings through pooling methods [76] could be an interesting approach towards representing global graph structure beyond training sizes.

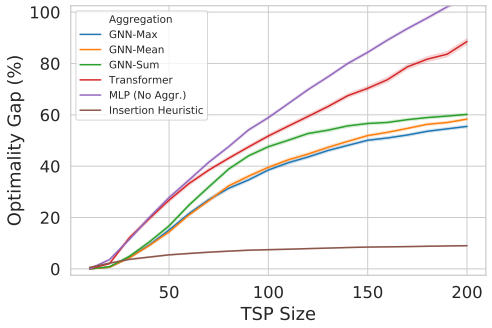


Figure 18: GNN aggregation functions (NAR decoder).

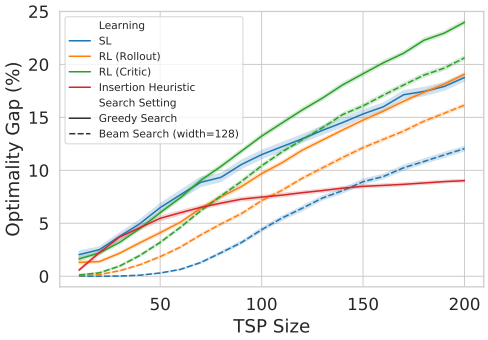


Figure 19: Comparing learning paradigms and solution search settings.

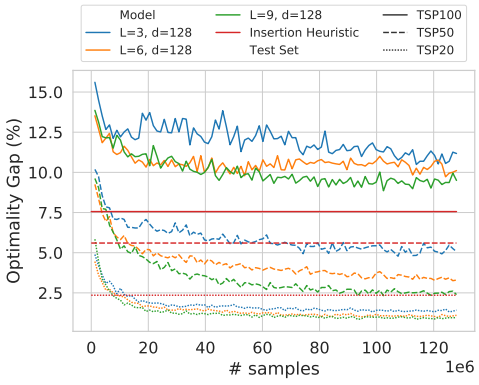


Figure 20: Scaling computation and model parameters for AR decoder.

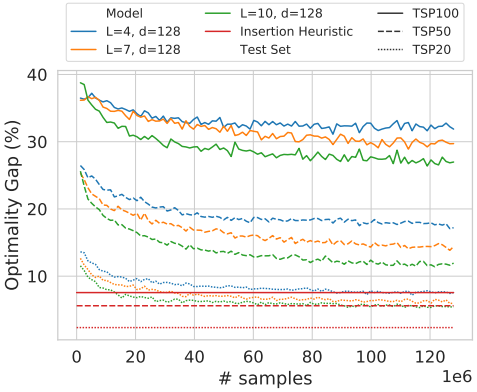


Figure 21: Scaling computation and model parameters for NAR decoder.

E Extra Results

NAR Decoders and Aggregation Functions In Section 5, we found that AR decoding provides a powerful sequential inductive bias for TSP and is able to generalizes well with both GNNs as well as structure-agnostic encoder architectures. This result may lead one to question the need for GNNs, altogether. Interestingly, Figure 18 illustrates a different trend for NAR architectures: GNN encoders generalize better than both Transformers and MLPs, indicating that leveraging graph structure is essential in the absence of the sequential inductive bias. (It is worth noting that, overall, all models with NAR decoders generalize poorly compared to AR architectures for our experimental setup.)

Critic baseline Figure 19 illustrates that, for identical models, the critic baseline [6, 17] is unable to match the performance of the rollout baseline [42] under both greedy and beam search settings. We did not explore tuning learning rates and hyperparameters for the critic network, opting to use the same settings as those for the actor. In general, getting actor-critic methods to work seems to require more parameter tuning than the rollout baseline.

Scaling computation for AR and NAR architectures In Figures 20 and 21, we present extended results for Section 5.6, where we scale model parameters and data. We observe that using larger models (up to 1.5 Million parameters) enables fitting the training dataset better. The impact of larger models is especially evident for NAR architectures. As previously noted, recent NAR-based models [54, 37] used more than 30 layers with over 10 Million parameters to outperform AR architectures on fixed TSP sizes. We believe that such overparameterized networks are able to memorize all patterns for small TSP training sizes [79], but the learnt policy is unable to generalize beyond training graph sizes as NAR decoding does not provide a useful inductive bias for TSP.

F Visualizing Model Predictions

As a final note, we present a visualization tool for generating model predictions and heatmaps of TSP instances, see Figures 22, 23. We advocate for the development of more principled approaches to neural combinatorial optimization, *e.g.* along with model predictions, visualizing the reduce costs for each edge (cheaply obtained using the Gurobi solver [31]) may help debug and improve learning-driven approaches in the future. Using reduce costs as supervision signals could also be an inexpensive alternative to running optimal solvers to create large labelled datasets.

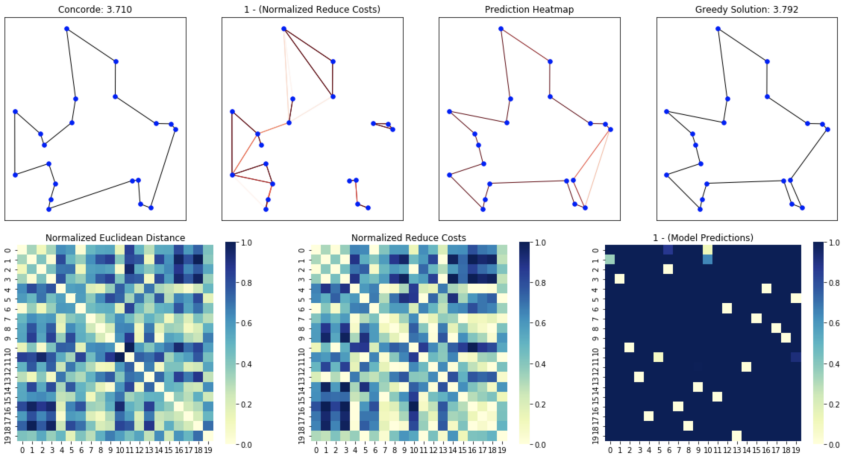


Figure 22: Prediction visualization for TSP20.

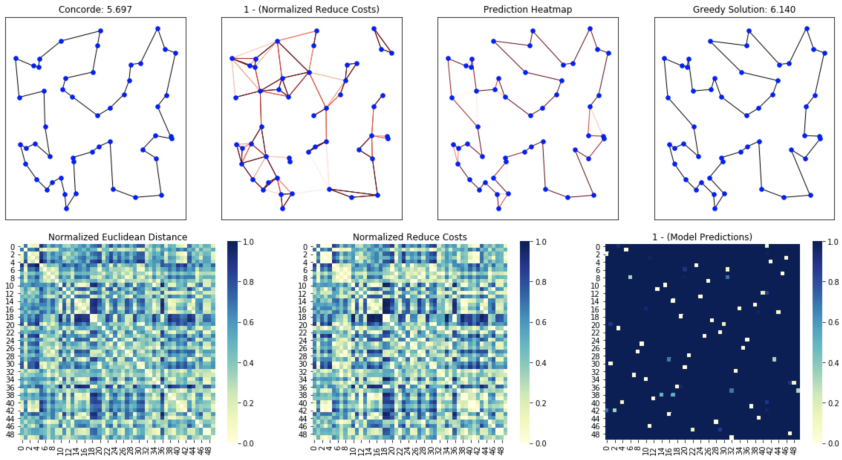


Figure 23: Prediction visualization for TSP50