Learning TSP Requires Rethinking Generalization

Chaitanya K. Joshi[1], **Quentin Cappart**[2], **Louis-Martin Rousseau**[2],
Thomas Laurent[3], **and Xavier Bresson**[1]
[1] Nanyang Technological University, Singapore
[2] Ecole Polytechnique de Montréal, Montreal, Canada
[3] Loyola Marymount University, LA, USA
{chaitanya.joshi,xbresson}@ntu.edu.sg, tlaurent@lmu.edu
{quentin.cappart,louis-martin.rousseau}@polymtl.ca

Abstract

End-to-end training of neural network solvers for combinatorial problems such as the Travelling Salesman Problem is intractable and inefficient beyond a few hundreds of nodes. While state-of-the-art Machine Learning approaches perform closely to classical solvers for trivially small sizes, they are unable to generalize the learnt policy to larger instances of practical scales. Towards leveraging transfer learning to solve large-scale TSPs, this paper identifies 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 entire neural combinatorial optimization pipeline, from network layers and learning paradigms to evaluation protocols.[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 Salesman 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 [54]. TSP is intractable to solve optimally above thousands of nodes for modern computers [5]. In practice, the Concorde TSP solver [4] 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 [81, 76] or computer architecture [67, 21] 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 [82, 11]. Using classical problems such as TSP, Minimum Vertex Cover and Boolean Satisfiability as benchmarks, recent *end-to-end* approaches [47, 80, 57] leverage advances in graph representation learning [22, 49, 35] 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.

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

Preprint. Under review.

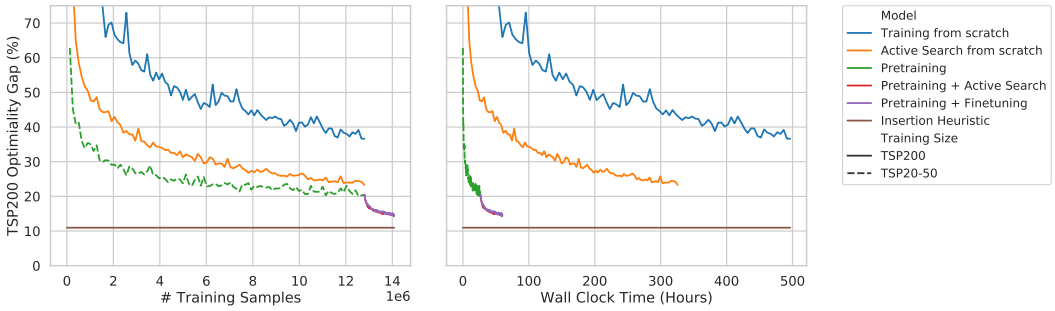


Figure 1: **Computational challenges of learning large scale TSP.** We compare three identical autoregressive GNN-based models trained on 12.8 Million TSP instances via reinforcement learning. We plot average optimality gap to the Concorde solver on 1,280 held-out TSP200 instances vs. number of training samples (left) and wall clock time (right) during the learning process. Training on large TSP200 from scratch is intractable and sample inefficient. Training directly on the 1,280 held-out samples via Active Search [10] further demonstrates the 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.

However, scaling to practical and real-world instances is still an open question [11] as the training phase of state-of-the-art approaches 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 [47, 50]. 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. This paper advocates 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.

The goal of this paper is two-fold: (1) Towards *end-to-end* learning of *scale-invariant* TSP solvers, we unify several state-of-the-art architectures and learning paradigms [71, 50, 23, 45] into one experimental pipeline and provide the first principled investigation on *zero-shot* generalization to large instances. (2) We open-source our framework and datasets to encourage the community to go beyond evaluating performance on fixed TSP sizes and study transfer learning for combinatorial problems. Our controlled experiments reveal that best practices regarding design choices such as GNN layers, normalization schemes, graph sparsification, and learning paradigms do not hold when evaluating generalization. In other words, learning to solve TSP at realistic scales will require rethinking the experimental as well as architectural status quo of neural combinatorial optimization to explicitly consider out-of-distribution generalization.

2 Related Work

In a recent survey, Bengio et al. [11] identified three broad approaches to leveraging machine learning for combinatorial optimization problems: learning alongside optimization algorithms [60, 31, 15], learning to configure optimization algorithms [93, 90, 26], and *end-to-end* learning to approximately solve optimization problems, *a.k.a.* neural combinatorial optimization [88, 10].

State-of-the-art end-to-end approaches for TSP use Graph Neural Networks [9] and *sequence-to-sequence* learning [83] 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 [47, 23, 50, 61]; and (2) non-autoregressive models, which produce the solution

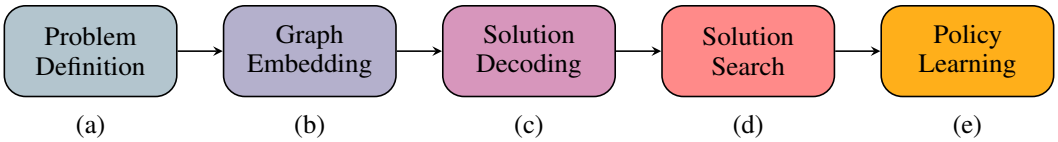


Figure 2: **End-to-end neural combinatorial optimization pipeline: (a)** The problem is formulated via a graph. **(b)** Embeddings for each graph node are obtained using a Graph Neural Network encoder. **(c)** Probabilities are assigned to each node for belonging to the solution set, either independent of one-another (*i.e.* Non-autoregressive decoding) or conditionally through graph traversal (*i.e.* Autoregressive decoding). **(d)** The predicted probabilities are converted into discrete decisions through classical graph search techniques such as greedy search or beam search. **(e)** 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).

in one shot [71, 70, 45]. Models can be trained to imitate optimal solvers via supervised learning or by minimizing the length of TSP tours via reinforcement learning [46]. Beyond TSP, other classical problems tackled by similar architectures include Vehicle Routing [68, 16], Maximum Cut [47], Minimum Vertex Cover [57], Boolean Satisfiability [80, 100], and Graph Coloring [41].

Advances on classical combinatorial problems have shown promising results in downstream applications to novel or under-studied optimization problems in the physical sciences [34, 69, 81] and computer architecture [63, 73, 66], where the development of exact solvers is expensive and intractable. For example, autoregressive architectures provide a strong inductive bias for device placement optimization problems [67, 103], while non-autoregressive models [13] are competitive with autoregressive approaches [44, 101] for molecule generation tasks.

3 Neural Combinatorial Optimization Pipeline

Many 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, we unify recent ideas [71, 50, 23, 45] via a five stage *end-to-end* pipeline illustrated in Figure 2. Our discussion focuses on the Travelling Salesman Problem (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. 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 [47] or learn faster by reducing the search space [45]. 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 [57].

3.2 Graph Embedding

A Graph Neural Network (GNN) encoder computes d-dimensional representations for each node in the input TSP graph. At each layer, nodes gather features from their neighbors to represent local graph structure via recursive message passing [32]. 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 [64, 12]:

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

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

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 [43], LayerNorm [7]), 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 GNNs [85, 12] have been shown to outperform simpler isotropic variants [49, 35] across several challenging domains, including TSP [25].

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 [71]. 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. [45]. For adjacent nodes i and j, we compute the unnormalized edge logits:

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

$W_1 \in \mathbb{R}^{3d\times d}, W_2 \in \mathbb{R}^{d\times 2}$, 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 [23, 50] or recurrent neural networks [88, 61], explicitly model this sequential inductive bias through step-by-step graph traversal.

We follow the attention decoder from Kool et al. [50], 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: $\hat{h}_i^C = W_C\ \Big[\ h_G,\ h_{\pi'_{t-1}}^L,\ h_{\pi'_1}^L\ \Big]$, where $W_C \in \mathbb{R}^{3d\times d}$ and learned placeholders are used for $h_{\pi'_{t-1}}^L$ and $h_{\pi'_1}^L$ at $t = 1$. The context $\hat{h}_i^C$ is then refined via a standard Multi-Head Attention (MHA) operation [84] over the node embeddings:

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

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 $\tanh$ function is used to maintain the value of the logits within $[-C, C]$ ($C = 10$) [10]. 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 [57]. On the other hand, AR decoders come with the sequential/tour constraint built-in and are the default choice for routing problems [50]. 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.

A recently proposed alternative [96] to constructive AR and NAR decoding schemes involves iteratively improving (sub-optimal) solutions via an attention mechanism in a fashion similar to local search heuristics.

3.4 Solution 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'.

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 [29, 97], 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 [28]. To better understand generalization, we focus on using greedy search and beam search/sampling with small $b = 128$.

3.5 Policy 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 [71, 45]. 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. [88]. We use teacher-forcing to stabilize training [95].

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 [10, 50] or Q-Learning [47]. 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 [94] 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 [10, 23] and the greedy rollout baseline proposed by Kool et al. [50].

4 Experiments

4.1 Controlled Experiment 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). 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 [50].

Training Datasets Our experiments focus on learning from variable TSP20-50 graphs. We also compare to training on fixed graph sizes TSP20, TSP50, TSP100, which have been the default choice in TSP literature. In the supervised learning paradigm, we generate a training set of 1,280,000 TSP samples and groundtruth tours using Concorde. Models are trained using the Adam optimizer [48] 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.

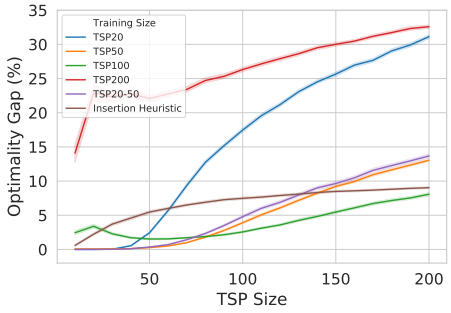


Figure 3: Learning from various TSP sizes.

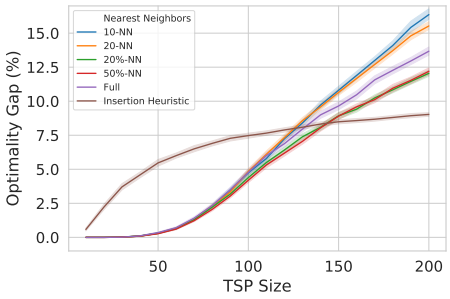


Figure 4: Impact of graph sparsification.

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 TSP samples, 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 models on fixed problem sizes [10, 23, 50], our evaluation 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.

4.2 Does learning from variable graphs 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 [50, 45] overshadows brittle out-of-distribution performance.

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, as noted by Deudon et al. [23], 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.

4.3 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 GNNs 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, 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

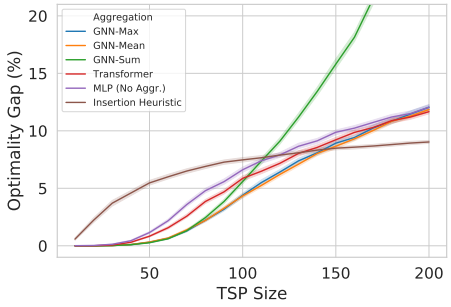


Figure 5: Impact of GNN aggregation functions.

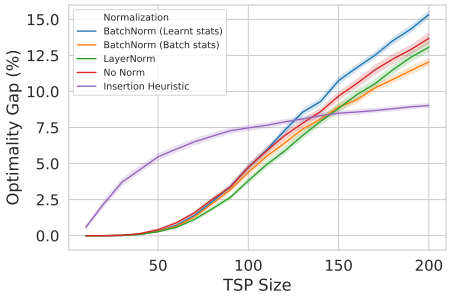


Figure 6: Impact of normalization schemes.

results also suggest that developing more principled graph reduction techniques beyond simple k-nearest neighbors for augmenting learning-based approaches may be a promising direction.

4.4 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 [23, 50] 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 [98] 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 [86]. We use the MAX aggregator in further experiments, as it generalizes well for both AR and NAR decoders (not shown).

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.

We further dissect the relationship between graph representations and normalization in Appendix F, 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 mechanisms that are both expressive as well as invariant to distribution shifts [19, 55].

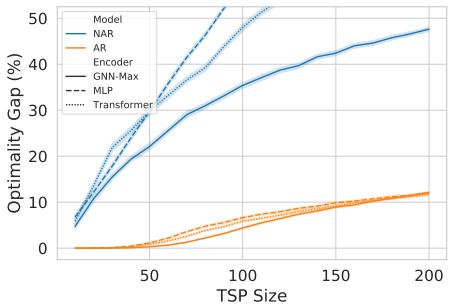


Figure 7: Comparing AR and NAR decoders.

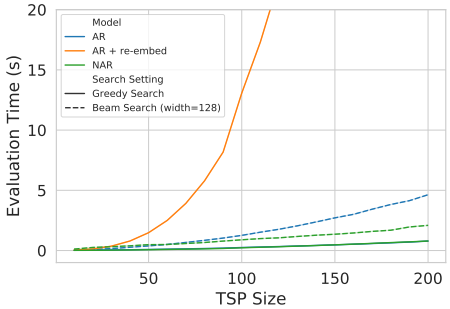


Figure 8: Inference time for various decoders.

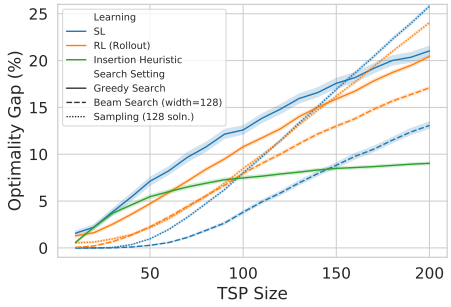


Figure 9: Comparing solution search settings.

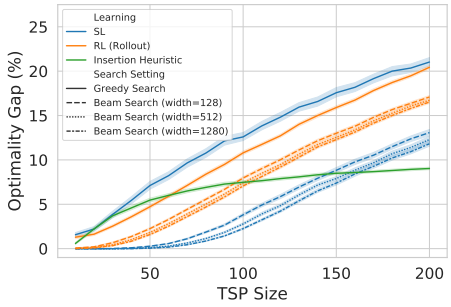


Figure 10: Impact of increasing beam width.

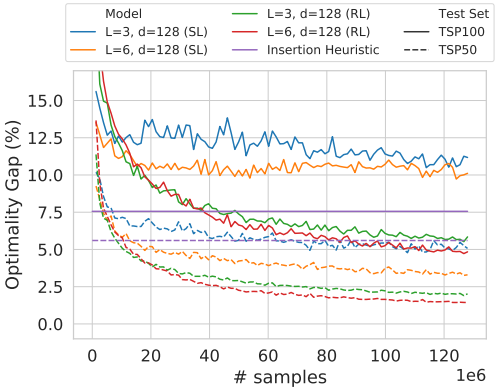


Figure 11: **Scaling computation and parameters for SL and RL-trained models.** All models are trained on TSP20-50 graphs. We plot optimality gap on 1,280 held-out samples of both TSP50 (performance on training size) and TSP100 (out-of-distribution generalization) under greedy decoding. Note that SL models are less amenable than RL models to greedy search.

4.5 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 work [71, 45] used more than 30 layers with over 10 Million parameters. We believe that such overparameterized networks are able to memorize all patterns for small TSP training sizes [102], but the learnt policy is unable to generalize beyond training graph sizes. At the same time, when compared fairly within the same codebase, 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 [47], see Figure 8.

4.6 How does the learning paradigms impact the search phase?

Identical models are trained via supervised learning (SL) and reinforcement learning (RL)[3]. 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 E, 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 techniques, as shown in Figures 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 [39, 92].

[3] We show only the greedy rollout baseline for clarity. Critic baseline results are available in Appendix D

4.7 Which learning paradigm scales better with computation and model parameters?

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 on natural language have highlighted the power of scale in effective transfer learning [75, 14].

To better understand the impact of learning paradigms when scaling compute, 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, more sample-efficient RL algorithms [79] may be a key ingredient for learning from larger TSPs beyond tens of nodes.

5 Conclusion

We perform the first principled investigation into transfer learning and *zero-shot* generalization for learning large scale TSP, unifying state-of-the-art architectures and learning paradigms into one experimental pipeline. Our findings suggest that learning *scale-invariant* TSP solvers requires rethinking the status quo of neural combinatorial optimization to explicitly accounting 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 costly in terms of inference time.
- Models trained with supervision are more ammenable to post-hoc search, while reinforcement learning approaches scale better with more computation as they do not rely on labelled data.

We hope that our unified experimental framework and codebase can serve as a foundation for future work exploring new architectures, transfer learning and meta-learning approaches for neural combinatorial optimization.

Broader Impact

Operations Research (OR) started in the first world war as an initiative to use mathematics and computer science to assist military planners in their decisions [30]. Today, combinatorial optimization algorithms developed in the OR community form the backbone of the most important modern industries including transportation, logistics, scheduling, finance and supply chains. However, designing powerful and robust optimization algorithms requires significant time and specialized knowledge, especially for understudied but high-impact problems arising in drug discovery [34], genomics [81], distributed systems [63], and circuit design [66].

In that respect, replacing domain experts and specialists with learning-based systems and data-driven approaches has had a 'democratizing' effect in the broad algorithm design community [52, 62]. Today, tech-savy generalists and software engineers can deploy state-of-the-art solutions for challenging problems in computer vision [33, 24, 36], natural language [65, 83, 84] and speech processing [72, 91]. We envision the long-term goal of *end-to-end* learning for combinatorial optimization to be similar: By making the modelling process intuitive for non-experts, learning-driven OR algorithms will be at the fingertips of more industries and can be personalized to individual users/use cases.

As an illustration, consider that you are a smartphone operating systems provider and want to optimize for battery life based on phone usage. Instead of investing in expensive hardware specialists and years of R&D, on-device models driven by reinforcement learning algorithms such as the ones studied

in this paper [10, 50] can learn to optimize battery life in a personalized way from each individual phone's usage patterns. However, unlike classical OR solvers which come with theoretical guarantees, our findings suggest that current learning-based approaches are not yet robust at handling situations beyond what they see in training and do not reliably scale to practical sizes. Continuing our analogy, reinforcement learning models could eventually end up draining battery life faster if the architectures and policies are not robust or generalizable to, *e.g.* low-cost and obscure smartphones which are often used by marginalized or low-income groups [77].

This paper presents an initial technical investigation of these questions on the classical Travelling Salesman Problem, with the broader goal of developing robust and reliable learning-driven solvers for *any* combinatorial problem. TSP has a rich history of serving as an engine of discovery for general purpose techniques in applied mathematics, including foundational work in Mixed Integer Programming [20], Branch-and-bound methods [59], heuristic search [1] and early neural networks [40, 3]. This line of work has already inspired novel reinforcement learning algorithms [50] and powerful Graph Neural Network architectures [45, 25]. Beyond combinatorial problems, designing statistical models that can generalize out-of-training-distribution is seen as an important challenge for the entire Machine Learning community [102, 27, 8, 18, 38, 51, 58]. In the near future, we hope TSP and combinatorial problems can serve as a challenging and practical benchmark for studying generalization in deep learning.

Acknowledgments

CJ would like to thank Vijay Prakash Dwivedi, Aaron Ferber, Elias Khalil, Wouter Kool, Ron Levie, Antoine Prouvost and Petar Veličković for helpful comments and discussions on a preliminary version of this work. XB is supported by NRF Fellowship NRFF2017-10.

References

[1] E. Aarts, E. H. Aarts, and J. K. Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[2] 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 arXiv:1603.04467*, 2016.

[3] B. Angeniol, G. D. L. C. Vaubois, and J.-Y. Le Texier. Self-organizing feature maps and the travelling salesman problem. *Neural Networks*, 1(4):289–293, 1988.

[4] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde tsp solver, 2006.

[5] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[6] F. Arabshahi, Z. Lu, S. Singh, and A. Anandkumar. Memory augmented recursive neural networks. *arXiv preprint arXiv:1911.01545*, 2019.

[7] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[8] D. Bahdanau, S. Murty, M. Noukhovitch, T. H. Nguyen, H. de Vries, and A. Courville. Systematic generalization: What is required and can it be learned? *arXiv preprint arXiv:1811.12889*, 2018.

[9] 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 arXiv:1806.01261*, 2018.

[10] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017.

[11] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

[12] X. Bresson and T. Laurent. An experimental study of neural networks for variable graphs. In *International Conference on Learning Representations*, 2018.

[13] 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.

[14] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[15] Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

[16] X. Chen and Y. Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6278–6289, 2019.

[17] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[18] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

[19] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. Principal neighbourhood aggregation for graph nets. *arXiv preprint arXiv:2004.05718*, 2020.

[20] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[21] J. Dean. The deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 8–14. IEEE, 2020.

[22] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[23] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the tsp by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.

[24] C. Dong, C. C. Loy, K. He, and X. Tang. Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence*, 38(2):295–307, 2015.

[25] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[26] A. Ferber, B. Wilder, B. Dilkina, and M. Tambe. Mipaal: Mixed integer program as a layer. In *AAAI Conference on Artificial Intelligence*, 2020.

[27] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[28] 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 arXiv:1909.13121*, 2019.

[29] Z.-H. Fu, K.-B. Qiu, M. Qiu, and H. Zha. Targeted sampling of enlarged neighborhood via monte carlo tree search for {tsp}, 2020.

[30] S. I. Gass and A. A. Assad. History of operations research. In *Transforming Research into Action*, pages 1–14. INFORMS, 2011.

[31] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*, 2019.

[32] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

[33] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[34] 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*, 4(2):268–276, 2018.

[35] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[36] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[37] A. Hermans, L. Beyer, and B. Leibe. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737*, 2017.

[38] F. Hill, A. Lampinen, R. Schneider, S. Clark, M. Botvinick, J. L. McClelland, and A. Santoro. Environmental drivers of systematicity and generalization in a situated agent. In *International Conference on Learning Representations*, 2020.

[39] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

[40] J. J. Hopfield and D. W. Tank. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[41] J. Huang, M. Patwary, and G. Diamos. Coloring big graphs with alphagozero. *arXiv preprint arXiv:1902.10162*, 2019.

[42] G. O. Inc. Gurobi optimizer reference manual. *URL http://www. gurobi. com*, 2015.

[43] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[44] W. Jin, R. Barzilay, and T. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, pages 2323–2332, 2018.

[45] C. K. Joshi, T. Laurent, and X. Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

[46] C. K. Joshi, T. Laurent, and X. Bresson. On learning paradigms for the travelling salesman problem. *arXiv preprint arXiv:1910.07210*, 2019.

[47] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[48] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[49] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[50] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[51] A. K. Lampinen and J. L. McClelland. Transforming task representations to allow deep learning models to perform novel tasks. *arXiv preprint arXiv:2005.04318*, 2020.

[52] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[53] J. Lee, I. Lee, and J. Kang. Self-attention graph pooling. *arXiv preprint arXiv:1904.08082*, 2019.

[54] J. K. Lenstra and A. R. Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975.

[55] R. Levie, M. M. Bronstein, and G. Kutyniok. Transferability of spectral graph convolutional neural networks. *arXiv preprint arXiv:1907.12972*, 2019.

[56] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[57] Z. Li, Q. Chen, and V. Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 539–548, 2018.

[58] T. Linzen. How can we accelerate progress towards human-like linguistic generalization? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Seattle, Washington, July 2020. Association for Computational Linguistics.

[59] J. D. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989, 1963.

[60] A. Lodi and G. Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

[61] 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: Methodologies and Applications*, 2020.

[62] J. Malik. Technical perspective: What led computer vision to deep learning? *Commun. ACM*, 60(6):82–83, May 2017.

[63] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. ACM, 2019.

[64] D. Marcheggiani and I. Titov. Encoding sentences with graph convolutional networks for semantic role labeling. *arXiv preprint arXiv:1703.04826*, 2017.

[65] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[66] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, et al. Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*, 2020.

[67] 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 *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2430–2439. JMLR.org, 2017.

[68] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9861–9871, 2018.

[69] A. Nigam, P. Friederich, M. Krenn, and A. Aspuru-Guzik. Augmenting genetic algorithms with deep neural networks for exploring the chemical space. *arXiv preprint arXiv:1909.11655*, 2019.

[70] A. Nowak, D. Folqué, and J. B. Estrach. Divide and conquer networks. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

[71] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint arXiv:1706.07450*, 2017.

[72] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[73] 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 arXiv:1905.02494*, 2019.

[74] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[75] 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. *arXiv preprint arXiv:1910.10683*, 2019.

[76] M. Raghu and E. Schmidt. A survey of deep learning for scientific discovery. *arXiv preprint arXiv:2003.11755*, 2020.

[77] M. Samaan. *The Effect of Income Inequality on Mobile Phone Penetration*. PhD thesis, Boston College. College of Arts and Sciences, 2003.

[78] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia. Learning to simulate complex physics with graph networks. *arXiv preprint arXiv:2002.09405*, 2020.

[79] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[80] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[81] 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*, pages 1–5, 2020.

[82] K. A. Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

[83] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[84] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[85] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[86] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020.

[87] S. B. Venkatakrishnan, M. Alizadeh, and P. Viswanath. Graph2seq: Scalable learning dynamics for graphs. *arXiv preprint arXiv:1802.04948*, 2018.

[88] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[89] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[90] P.-W. Wang, P. L. Donti, B. Wilder, and Z. Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *arXiv preprint arXiv:1905.12149*, 2019.

[91] Y. Wang, R. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio, et al. Tacotron: Towards end-to-end speech synthesis. *arXiv preprint arXiv:1703.10135*, 2017.

[92] S. Welleck, I. Kulikov, S. Roller, E. Dinan, K. Cho, and J. Weston. Neural text generation with unlikelihood training. In *International Conference on Learning Representations*, 2020.

[93] B. Wilder, B. Dilkina, and M. Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.

[94] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

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

[96] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim. Learning improvement heuristics for solving the travelling salesman problem. *arXiv preprint arXiv:1912.05784*, 2019.

[97] Z. Xing and S. Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem, 2020.

[98] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[99] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*, pages 4800–4810, 2018.

[100] E. Yolcu and B. Poczos. Learning local search heuristics for boolean satisfiability. In *Advances in Neural Information Processing Systems*, pages 7990–8001, 2019.

[101] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in neural information processing systems*, pages 6410–6421, 2018.

[102] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[103] 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 arXiv:1910.01578*, 2019.

A Additional Context for Figure 1: Computational challenges of learning large scale TSP

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 4.1. 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 [50] 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. [10].

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. The Appendix of Kool et al. [50] provides a detailed description of insertion heuristic approaches.

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, training data and sample efficient learning algorithms. 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 [20, 59, 40].

TSP and the associated class of 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, Minimum Vertex Cover [47, 57, 80][4], the same architectures do not show strong generalization for TSP. For example, the *furthest insertion* heuristic outperforms or is competitive with state-of-the-art approaches for TSP above tens of nodes, see Figure D.1.(e) and (f) from Khalil et al. [47] or Figure 5 from Kool et al. [50], despite using more computation and data than our controlled experimental study.

[4] 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 neural networks in enabling such generalization [57, 6].

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 4.1. All experiments were implemented in PyTorch [74] 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.

We experimented with improving the latency of GNN-based models by using graph machine learning libraries such as DGL [89]. DGL requires graphs to be prepared as sparse library-specific data objects, which significantly boosts the inference speed of GNNs. However, using DGL had a negative impact on the speed of the rest of our pipeline (batched data preparation, decoders, beam search). This issue is especially amplified for reinforcement learning, where we constantly generate new random datasets at each epoch. For now, we present timings and results with pure PyTorch code. We confirm that results are consistent with using DGL, but decided against it in order to run a large volume of experiments for more comprehensive analysis.

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 Datasets

We generate 2D Euclidean TSP instances of varying sizes and complexities as graphs of n node locations sampled uniformly in the unit square $S = \{x_i\}_{i=1}^{n}$ and $x_i \in [0, 1]^2$. For supervised learning, we generate a training set of 1,280,000 samples each for TSP20, TSP50, TSP100, and TSP20-50. The groundtruth tours are obtained using the Concorde solver [4]. For reinforcement learning, 128,000 samples are randomly generated for each epoch (without optimal solutions). We compare models on a held-out test set of 25,600 TSP samples and their corresponding optimal tours, consisting of 1,280 instances each of TSP10, TSP20, ..., TSP200. We release all dataset files as well as the associated scripts to produce TSP datasets of arbitrarily large sizes along with our open-source codebase.

D Omitted Results

NAR Decoders and Aggregation Functions In Section 4, 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 in the neural combinatorial optimization pipeline, altogether. Interestingly, Figure 13 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, see Figure 7.)

Encode-Process-Decode Architectures To motivate the use of recurrent and adaptive architectures, consider how students in high-school learn new mathematical concepts: An assessment of how well a student understands a concept involves testing on problems of increasing difficulty that generally require extrapolating beyond what was encountered during practice sessions. Intuitively,

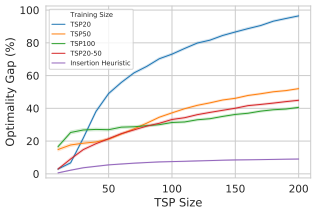


Figure 12: Learning from various TSP sizes (NAR decoder).

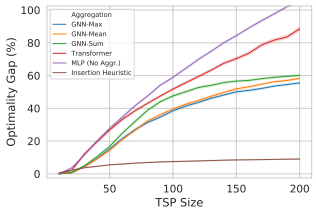


Figure 13: GNN aggregation functions (NAR decoder).

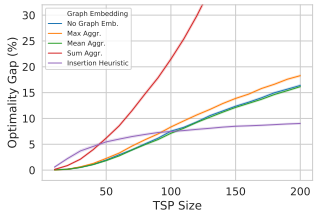


Figure 14: Impact of graph embedding functions.

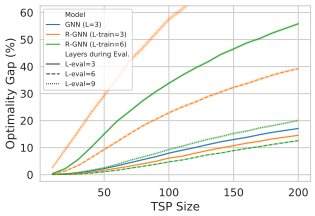


Figure 15: Encode-Process-Decode Architectures.

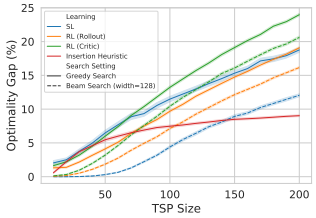


Figure 16: Comparing solution search settings.

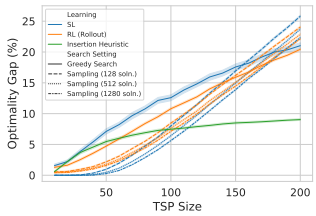


Figure 17: Impact of sampling more solutions.

humans can generalize to more complex problems than what they 'train' on through deeper chains of processing and reasoning.

We explored this inductive bias by instantiating Recurrent GNN encoders, which can be trained for a certain amount of message passing steps/layers (L-train), but can perform more rounds of message passing (L-eval) during inference [56, 87]. To do so, we leverage the *Encode-Process-Decode* architectural paradigm [9], which has shown promising generalization for physics simulation engines [78], graph algorithms [86, 19] and SAT problems [80]. We redefine the GNN encoder from Section 3.2 to use a shared Gated Recurrent Unit-based block across message passing steps:

$$h_i^{\ell+1} = \text{GRU}\left(\text{AGGR}_{j\in\mathcal{N}_i}\left(\sigma(e_{ij}^{\ell}) \odot V h_j^{\ell}\right),\ h_i^{\ell}\right), \tag{8}$$

$$e_{ij}^{\ell+1} = \text{GRU}\left(Bh_i^{\ell} + Ch_j^{\ell},\ e_{ij}^{\ell}\right), \tag{9}$$

where GRU denotes the GRU cell [17] with LayerNorm [7], $V, B, C \in \mathbb{R}^{d\times d}$ are other learnable parameters, AGGR represents the neighborhood aggregation function (we use 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. We generate predictions using the AR decoder after an arbitrary number of message passing steps L.

In Figure 15, we compare performance and generalization of Recurrent GNN encoders trained with $L_{\text{train}} = \{3, 6\}$ message passing steps to standard GNN encoders with $L = 3$ layers (with identical parameter budgets of approximately 350,000). During evaluation, we allow the R-GNN encoders to perform a variable number of steps $L_{\text{eval}} = \{3, 6, 9\}$. R-GNNs differ from standard GNNs by not using residual connections and controlling information flow across layers through shared GRU units and show better generalization than standard GNNs when $L_{\text{train}} = L_{\text{eval}}$. Unfortunately, we were unable to improve the performance of R-GNNs by performing more message passing steps, suggesting further development of *encode-process-decode* architectures for routing problems.

Critic baseline Figure 16 illustrates that, for identical models, the critic baseline [10, 23] is unable to match the performance of the rollout baseline [50] under both greedy and beam search settings. We did not explore separately 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 18 and 19, we present extended results for Section 4.7, 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

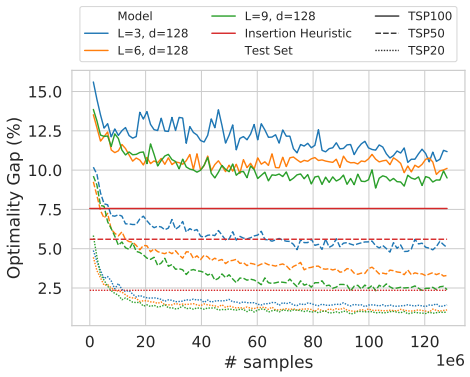


Figure 18: Scaling computation and model parameters for AR decoder.

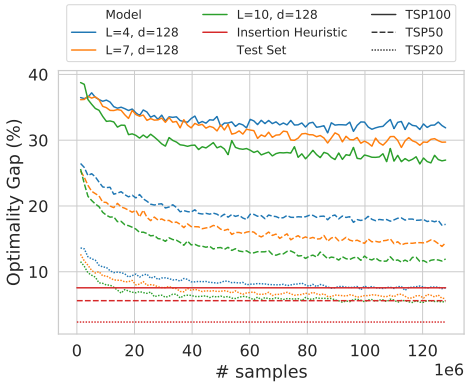


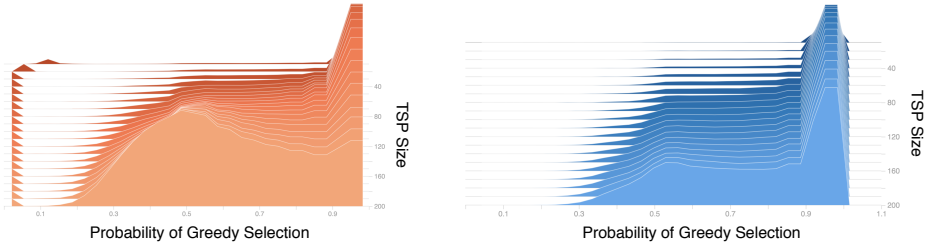
Figure 19: Scaling computation and model parameters for NAR decoder.

larger models is especially evident for NAR architectures. As previously noted, recent NAR-based models [71, 45] 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 [102], but the learnt policy is unable to generalize beyond training graph sizes as NAR decoding does not provide a useful inductive bias for TSP.

E Learning Paradigms and Amenity to Search

Figure 10 (Section 4.6) and Figure 17 demonstrate that SL models are more amenable to beam search and sampling, but are outperformed by RL-rollout models under greedy search. In Figure 20, 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 20b) 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 20a) 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.



(a) Supervised Learning

(b) Reinforcement Learning

Figure 20: Histograms of greedy selection probabilities (x-axis) across TSP sizes (y-axis).

F Visualizing Node and Graph Embedding Spaces

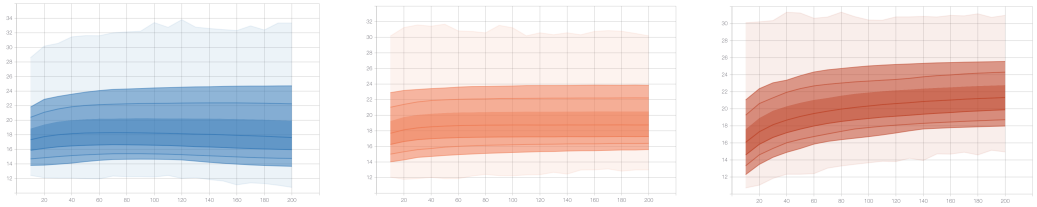
Our results in Section 4.4 suggest that inference beyond training sizes requires the development of GNN architectures and normalization layers that are both expressive as well as invariant to distribution shifts. We explore how node and graph embeddings for TSP graphs evolve across training distribution (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. (In Figure 14, we confirm that mean is the most effective strategy for aggregating node embeddings to form the graph embedding, while summation leads to unstable generalization beyond training size. All our experiments use mean to compute graph embeddings.)

We utilize distribution plots to study the variation in embedding statistics[5] of three identical models: (1) ***GNN-Max***, which represents our best model configuration from Section 4: 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 [37] 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.

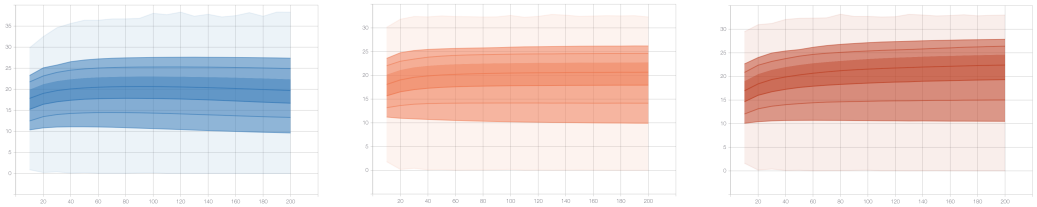
Node Embedding Space In Figures 21 and 22, we see that *GNN-Max* leads to the most stable node embedding norms and pair-wise distances (which are calculated at an intra-graph level) across TSP sizes. On the other hand, *GNN-Sum* and *GNN-Max + learnt BN* lead to fluctuating and monotonically increasing embedding norms as size increases, *e.g.* compare Figure 21b and Figure 21c. 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

[5] 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, and are easily implemented via TensorBoard [2].



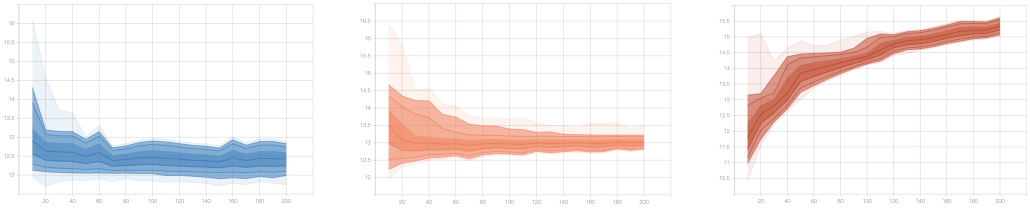
(a) *GNN-Sum* (b) *GNN-Max* (c) *GNN-Max + learnt BN*

Figure 21: Distribution plots of node embedding ℓ_2 norms (y-axis) across TSP sizes (x-axis).



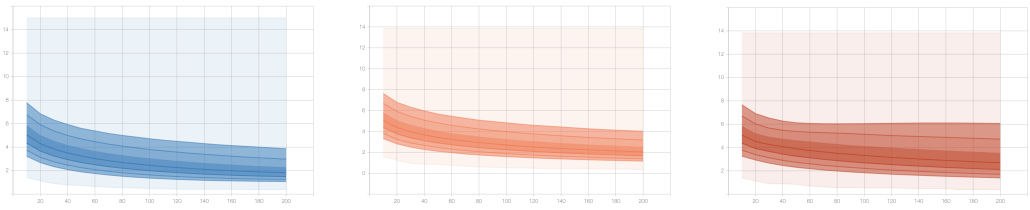
(a) *GNN-Sum* (b) *GNN-Max* (c) *GNN-Max + learnt BN*

Figure 22: Distribution plots of node embedding distances (y-axis) across TSP sizes (x-axis).



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

Figure 23: 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 24: Distribution plots of graph embedding distances (y-axis) across TSP sizes (x-axis).

large graphs. This enables better generalization, as the decoder has lower chances of encountering embeddings which are statistically different than those seen during training.

Graph Embedding Space Figures 23 and 24 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 23b and Figure 23c.

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 25 and 26. 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 attention [53] or hierarchical methods [99] could be an interesting approach towards representing global graph structure beyond training sizes.

G Visualizing Model Predictions

As a final note, we present a visualization tool for generating model predictions and heatmaps of TSP instances, see Figures 27, 28 and 29. 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 (obtained using the Gurobi solver [42]) may help debug and improve learning-driven approaches in the future.

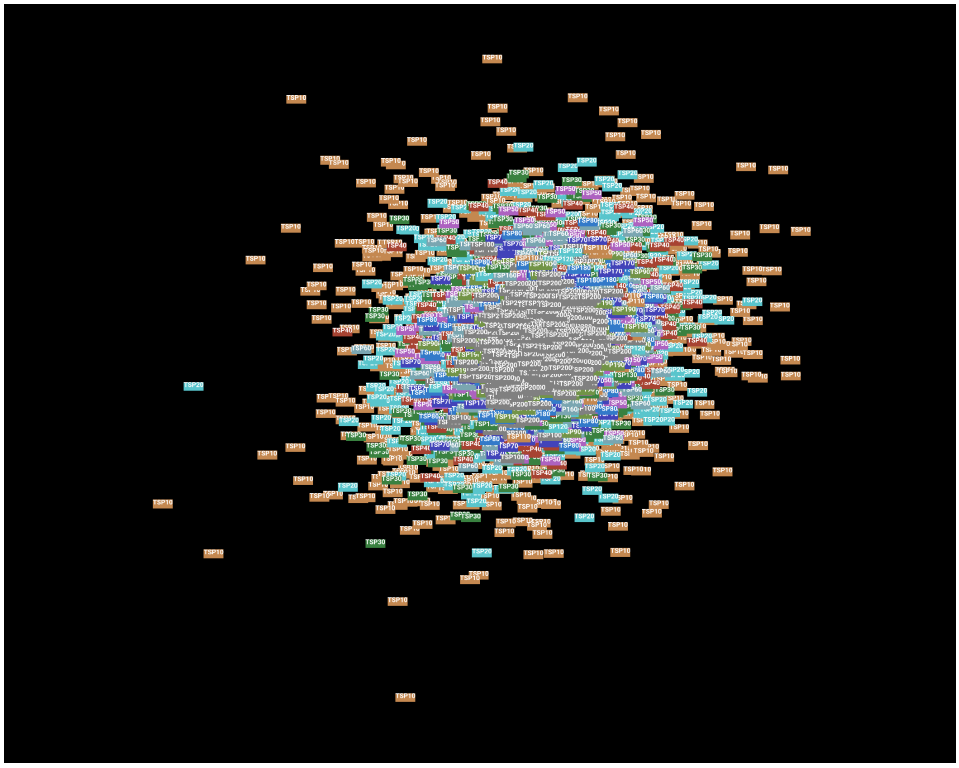


Figure 25: 2D PCA of graph embedding space for *GNN-Max*.

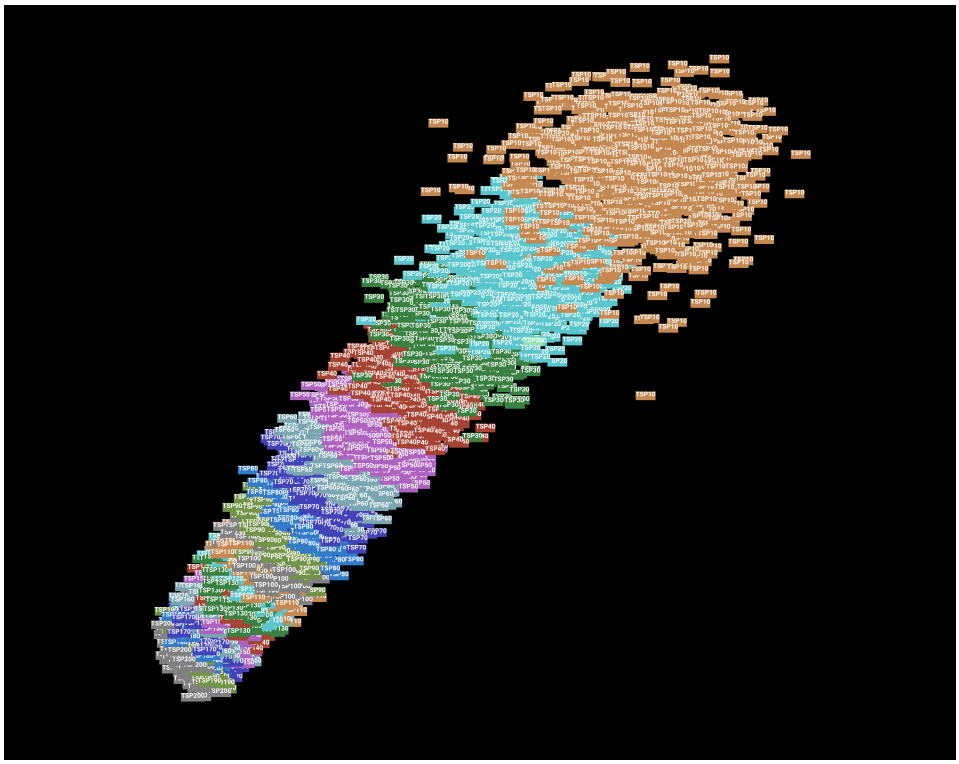


Figure 26: 2D PCA of graph embedding space for *GNN-Max + learnt BN*.

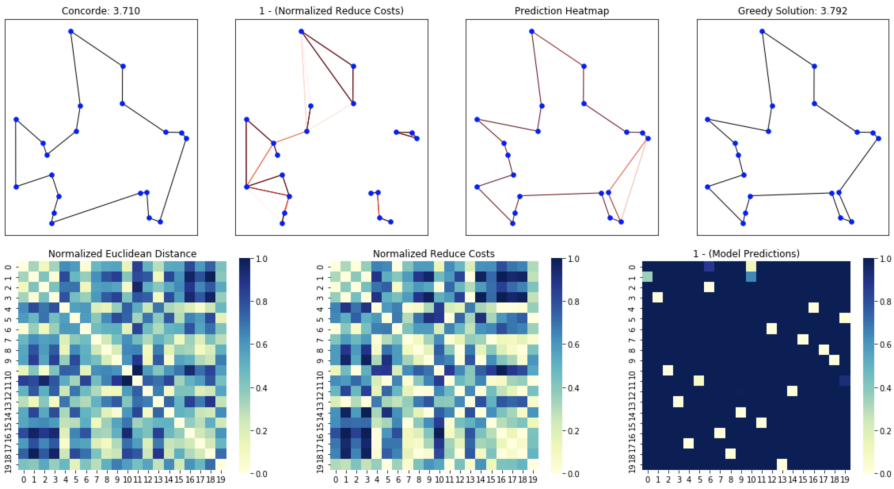


Figure 27: Prediction visualization for TSP20.

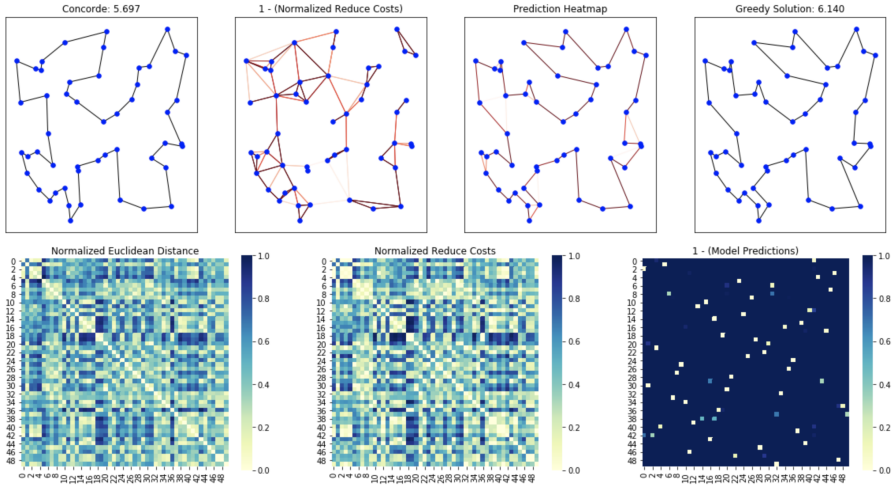


Figure 28: Prediction visualization for TSP50

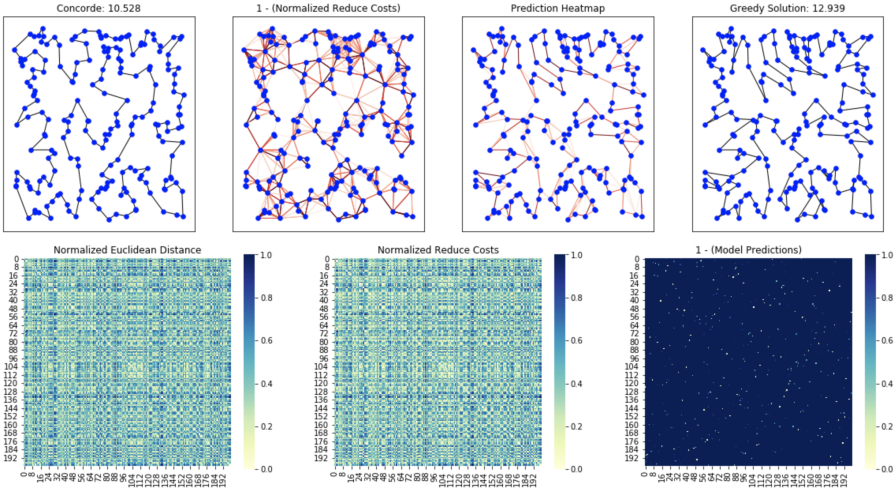


Figure 29: Prediction visualization for TSP200