Graph Convolutional Neural Networks for the Travelling Salesman Problem

Chaitanya Krishna Joshi

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering (Computer Science)

Supervised by Assoc. Prof. Xavier Bresson

April 2019

School of Computer Science and Engineering
ABSTRACT

Combinatorial optimization problems, also called NP-hard problems, are practical constraint satisfaction problems that are impossible to solve optimally at large scales. In practice, handcrafted heuristic algorithms are able to solve problems with up to a million variables and constraints. These algorithms are the backbone of modern industries such as transportation, supply chain, and scheduling. However, coming up with good heuristics requires significant specialized knowledge. A recent line of work investigates using deep neural networks to directly learn these heuristics from problem instances instead.

In this paper, we introduce a novel deep learning approach for approximately solving the most famous NP-hard problem in recent history, the Travelling Salesman Problem. We focus on the 2D Euclidean TSP and use Graph Convolutional Neural Networks and beam search to predict a valid TSP tour given an input graph with up to 100 nodes. Our approach outperforms all recently proposed deep learning techniques in terms of both solution quality and speed when evaluated on problem instances of fixed graph sizes. However, experiments on the generalization capabilities of our models show a drastic drop in performance when evaluated on graph sizes different from those that models were trained on.

Our results highlight an important flaw in the current paradigm of learning-based approaches for TSP and combinatorial optimization: comparing among approaches based on performance for discrete problem sizes completely ignores generalization. We advocate for the machine learning community to switch focus towards building size-agnostic models with strong generalization capabilities in order to scale up to realistic problem sizes.
ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Xavier Bresson. Dr. Bresson provided the perfect balance of freedom to pursue my own ideas and guidance at critical stages throughout the course of this project. His passion for research and attention to detail are inspiring and infectious. I am infinitely thankful to Dr. Bresson for his mentorship and generosity with his time. (And his GPUs!)

This thesis is the culmination of the skills I picked up during 4 enriching years at the School of Computer Science and Engineering, NTU, Singapore. I am grateful to all my professors, classmates and staff at NTU for this opportunity. I am also grateful for all the friends I made throughout my journey at NTU. Thank you, Vidur, Genevieve, Suyash, Bobby, Nikhil, Shruthi, Harivansh, and countless others, for the endless inspiration and for keeping me sane till the end.

My sister Kuhu, my parents Veena and Vivek, and my entire family were my constant pillar of support and motivation. Thank you for believing in me ten times more than I did.

Finally, this thesis is dedicated to my late grandfathers, M.C. Joshi and G.N. Joshi, who taught me how to live life with passion and perseverance.
CONTENTS

1 Introduction ................................................. 7
   1.1 Background ............................................. 7
   1.2 Contributions .......................................... 7
   1.3 Implications ............................................. 8

2 Literature Review .......................................... 9
   2.1 Deep Learning on Euclidean Data ......................... 9
   2.2 Non-Euclidean and Graph-structured Data .............. 9
   2.3 Deep Learning on Non-Euclidean Data .................. 10
       2.3.1 Transductive Graph Neural Networks ............. 10
       2.3.2 Inductive Graph Neural Networks ................. 10
   2.4 Combinatorial Optimization ............................ 11
   2.5 Machine Learning for Combinatorial Optimization ...... 11
   2.6 The Travelling Salesman Problem ...................... 12
       2.6.1 Non-learning Approaches ......................... 12
       2.6.2 Learning-based Approaches ....................... 13

3 Dataset .................................................. 15
   3.1 Problem Formulation ................................... 15
   3.2 Dataset Preparation ................................... 16

4 Model ...................................................... 17
   4.1 Generic formulation of ConvNets ....................... 17
   4.2 Model Architecture .................................... 17
   4.3 Loss Function ......................................... 19
   4.4 Tour Decoder ............................................. 20
   4.5 Hyperparameter Configurations ......................... 21

5 Experiments ................................................ 22
   5.1 Training Procedure ..................................... 22
   5.2 Evaluation Procedure .................................. 22
   5.3 On Comparison with State-of-the-art .................. 23
   5.4 Evaluating Generalization via Transfer Learning ...... 23
6 Results

6.1 Fixed Sized TSP Instances ......................................................... 24
6.2 Comparison to State-of-the-Art ..................................................... 25
6.3 Visualizations ............................................................................ 27
6.4 Generalization to Variable Sized TSPs .......................................... 29
6.5 Comments on Recent Work ......................................................... 30

7 Conclusions ................................................................................ 32

7.1 Findings .................................................................................... 32
7.2 Future Work ............................................................................... 32

References ..................................................................................... 33

A Training Logs ............................................................................. 38

B Additional Visualizations .............................................................. 39
LIST OF FIGURES

1  Overview of our approach ................................................................. 8
2  A generic ConvNet architecture ......................................................... 9
3  A sample TSP instance of size 30 nodes ............................................. 15
4  The proposed Graph Convolution layer ............................................. 18
5  Validation Optimality Gap vs Training Epochs for various runs .......... 24
6  TSP10 model prediction visualization for a sample test set instance ... 27
7  TSP20 model prediction visualization for a sample test set instance ... 27
8  TSP30 model prediction visualization for a sample test set instance ... 28
9  TSP50 model prediction visualization for a sample test set instance ... 28
10 TSP100 model prediction visualization for a sample test set instance .. 28
11 Optimality gap of different methods as a function of problem size ... 30
12 Training logs for various runs .......................................................... 38
13 TSP10 model prediction visualization for sample test set instances .... 39
14 TSP20 model prediction visualization for sample test set instances .... 40
15 TSP30 model prediction visualization for sample test set instances .... 41
16 TSP50 model prediction visualization for sample test set instances .... 42
17 TSP100 model prediction visualization for sample test set instances ... 43
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Recent learning-based approaches to TSP</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Summary statistics for TSP datasets</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Model hyperparameter configurations for various TSP instances</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Timing and performance of our technique for various TSP instance sizes</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>Performance of our technique compared to non-learned baselines and state-of-the-art methods for various TSP instance sizes</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Generalization performance of our technique to variable instance sizes in the greedy setting</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Generalization performance of our technique to variable instance sizes using beam search</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Generalization performance of our technique to variable instance sizes using beam search and shortest tour heuristic</td>
<td>29</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 BACKGROUND

Deep Learning is changing the world. The growing computational power of modern GPU-based computers and the availability of large datasets for training deep neural networks have led to qualitative breakthroughs for large-scale machine learning problems, from speech recognition (Hinton et al., 2012) and machine translation (Sutskever et al., 2014) to image analysis and computer vision (LeCun et al., 2010; Krizhevsky et al., 2012).

On the other hand, Combinatorial optimization problems, also called NP-hard problems, are the family of integer constrained optimization problems which are impossible to solve with optimality at large scales. In practice it is possible to compute approximate solutions for instances up to a million decision variables and constraints using carefully handcrafted heuristics (Applegate et al., 2006). Robust approximation algorithms to popular NP-hard problems have various practical applications and are the backbone of modern industries such as transportation, supply chain, energy, finance, and scheduling.

The most famous NP-hard problem in recent history, the Travelling Salesman Problem (TSP), asks the following question: “Given a list 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 graph, one needs to search the space of permutations to find an optimal sequence of nodes, called a tour, with minimal total edge weights (tour length).

In general, many NP-hard problems are highly structured and can be formulated as sequential decision making tasks on graphs. Hence, machine learning can be used to train policies for these problems instead of handcrafting good heuristics, which may be expensive or require significant specialized knowledge (Bengio et al., 2018). In particular, recent advances in graph neural network techniques (Bronstein et al., 2017) are a good fit for the task because they naturally operate on the graph structure of these problems.

1.2 CONTRIBUTIONS

In this paper, we introduce a novel deep learning approach to approximately solving the 2D Euclidean TSP using Graph Convolutional Neural Networks (Bresson & Laurent, 2017) and beam search. Figure 1 presents an overview of our approach.

Our model takes a graph as an input and extracts compositional feature vectors from its nodes and edges by stacking several graph convolution layers. The output of the neural network is an edge adjacency matrix denoting the probabilities of edges occurring on the TSP tour. This edge prediction heat-map is converted to a valid tour using a post-hoc beam search strategy. The model parameters are trained in a supervised manner using pairs of problem instances and optimal solutions.
Figure 1: Overview of our approach. Taking a 2D graph as input, our Graph ConvNet model outputs an edge adjacency matrix denoting the probabilities of edges occurring on the TSP tour. This is converted to a valid tour using beam search.

We perform several experiments to compare the empirical performance of our approach to non-learned baselines and state-of-the-art deep learning techniques for TSP instances of fixed graph sizes with 20, 50 and 100 nodes.

Our approach outperforms all recently proposed learning-based techniques in terms of both solution quality and evaluation time. We are further able to improve our performance at the cost of time by using a hybrid approach combining our learned model with local search heuristics. We attribute our gains in performance to more powerful representation learning capabilities of very deep Graph ConvNet architectures and the highly parallelized nature of the beam search algorithm.

Additionally, we evaluate the generalization capabilities of our models trained on smaller problem instances to larger instances, and vice-versa. We find that our models learn to specialize to fixed problem instances and exhibit poor generalization to new sizes.

1.3 Implications

Our strong empirical results with a supervised learning approach are extremely surprising: TSP can be naturally formulated as a reinforcement learning task, and recent work has focused heavily on training models using the RL formulation. Comparatively poor empirical results of supervised learning-based methods has supported the argument in favour of RL. However, our results should not be used to conclude that supervised learning is now the superior approach for combinatorial optimization problems. Instead, we feel that our results highlight an important flaw in the current paradigm of learning-based approaches for TSP and combinatorial optimization.

Experiments on the generalization capabilities of our model show a drastic drop in performance when evaluated on problem sizes different from those that models were trained on. Despite their state-of-the-art performance on fixed problem sizes, it is clear that our models are overfit to the training graphs and are unable to generalize to variable sizes. Hence, the current status quo of comparing among learning-based approaches is flawed due to its focus on performance for discrete problem sizes and ignores generalization to variable sizes.
2 LITERATURE REVIEW

2.1 DEEP LEARNING ON EUCLIDEAN DATA

Deep Learning is a subfield of Machine Learning that focuses on learning complicated concepts by building them from simpler ones in a hierarchical or multi-layer manner. Artificial neural networks are popular realizations of such deep multi-layer hierarchies. In the past few years, the growing computational power of modern GPU-based computers and the availability of large datasets have enabled the training of neural networks with many layers and degrees of freedom (LeCun et al., 2015; Goodfellow et al., 2016). This has led to qualitative breakthroughs on a wide variety of machine learning tasks, from speech recognition (Hinton et al., 2012) and machine translation (Sutskever et al., 2014) to image analysis and computer vision (LeCun et al., 2010; Krizhevsky et al., 2012).

Convolutional Neural Networks (LeCun et al., 1998) and Recurrent Neural Networks (Hochreiter & Schmidhuber, 1997) are the generic building blocks of modern deep learning architectures for computer vision (CV) and natural language processing (NLP) tasks.

![Figure 2: A generic ConvNet architecture which interpolates between convolution and pooling layers to extract compositional features from the input.](image)

2.2 NON-EUCLIDEAN AND GRAPH-STRUCTURED DATA

ConvNets and RNNs require the data domain to be regular, such as 2D or 3D Euclidean grids for CV and 1D lines for NLP. However, most real-world data beyond images and language has an underlying structure that is non-Euclidean. Such complex data commonly occurs in science and engineering, and can be modelled by heterogeneous graphs.

Chemical Graphs A chemical graph is a labeled graph whose vertices correspond to the atoms of the compound and edges correspond to chemical bonds (McNaught & McNaught, 1997).

Computer Graphics In computer graphics, 3D objects are modeled as Riemannian manifolds (surfaces) endowed with properties such as color texture (Bronstein et al., 2006).

Social Networks The characteristics of users can be modeled as signals on the vertices of the social graph (Lazer et al., 2009).
Genetics In genetics, gene expression data are modeled as signals defined on the regulatory network (Davidson et al., 2002).

Neuroscience In neuroscience, graph models are used to represent anatomical and functional structures of the brain. (Bielza & Larranaga, 2014)

Sensor Networks Sensor networks are graph models of distributed interconnected sensors, whose readings are modelled as time-dependent signals on the vertices (Akyildiz et al., 2002).

In many applications, geometric data are large and complex (in the case of social networks, on the scale of billions), and are natural targets for machine learning techniques. However, the non-Euclidean nature of such data implies that there are no familiar properties such as common system of coordinates, vector space structure, or shift invariance.

2.3 Deep Learning on Non-Euclidean Data

Geometric deep learning is an umbrella term for emerging techniques attempting to generalize deep neural networks to non-Euclidean domains such as graphs and manifolds. Bronstein et al. (2017) classify these techniques based on the neural network architectures which operate on fixed domains (transductive) and those which can handle multiple domains (inductive).

2.3.1 Transductive Graph Neural Networks

An example of a fixed graph domain is the prediction of the position of users in a location-based social network representing the geographic coordinates of users as a time-dependent signal on the vertices of the social graph (Cho et al., 2011). In this problem, the domain (social graph) is assumed to be fixed and the model does not need to generalize to unseen graphs.

Convolutional neural networks for fixed domain graphs have been developed based on spectral graph theory (Chung, 1997). Bruna et al. (2013) proposed to formulate graph convolution operations in the spectral domain using the graph Laplacian as an analogy for the Euclidean Fourier transform (Hammond et al., 2011). Henaff et al. (2015) extended the spectral graph convolution framework to use smooth spectral filters for spatial localization. Defferrard et al. (2016) used Chebyshev polynomials to achieve linear complexity for sparse graphs, Levie et al. (2017) applied Cayley polynomials to focus on narrow-band frequencies, and Monti et al. (2017) dealt with multiple (fixed) graphs. Kipf & Welling (2016) simplified the spectral ConvNet architecture using 1-hop filters to solve the semi-supervised clustering task.

2.3.2 Inductive Graph Neural Networks

Computer graphics and vision applications are examples of variable graph domains where models must generalize to completely unseen graphs. For example, to find correspondence between shapes modeled as manifolds, one has to work with multiple such domains (shape manifolds).

A generic formulation of neural networks on variable graphs was proposed by Gori et al. (2005), Scarselli et al. (2009) based on RNNs. This work was extended by Li et al. (2015) using a GRU architecture and a hidden state that captures the average information in local neighborhoods of the graph. Sukhbaatar et al. (2016) introduced a vanilla Graph ConvNet and used this new architecture to solve learning communication tasks. Marcheggiani & Titov (2017) introduced an edge gating
mechanism in Graph ConvNets for semantic role labeling. Bresson & Laurent (2017) proposed the most
generic and powerful formulation of Graph ConvNets by combining the architecture of Sukhbaatar et al.
(2016) with edge gating (Marcheggiani & Titov 2017) and residual connections (He et al. 2016).

Other formulations of graph neural networks for the inductive setting are based on random walks
(Grover & Leskovec 2016), sampling and aggregation from local neighborhoods (Hamilton et al., 2017),
and learning non-linear approximations of the power of graph Laplacian operators (Bruna &
Li 2017).

2.4 Combinatorial Optimization

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. Nowadays it is widely used in the
industry, including but not limited to transportation, supply chain, energy, finance, and scheduling.

OR Problems are formulated as integer constrained optimization, i.e. with integral or binary variables
called decision variables). While not all such problems are hard to solve (e.g. shortest path
problems), we concentrate on combinatorial (NP-hard) problems. This is a worst case complexity
for a general purpose problem as exhaustive search algorithms are not tractable. In practice it is
possible to solve instances with up to a million decision variables and constraints.

The theory and algorithm design communities have typically used graphs to formulate NP-hard
Combinatorial optimization problems or treated them as decision versions of graph optimization
problems. The Travelling Salesman Problem (TSP) and the Minimum Spanning Tree Problem
(MST) are two of the most well studied graph combinatorial optimization problems.

Classically, approaches to tackling an optimization problem can be divided into three categories:
exact algorithms, approximation algorithms, and heuristics. Exact algorithms are based on enumeration
or branch-and-bound with an integer programming formulation. They are guaranteed to find
optimal solutions but are generally prohibitive for large instances. On the other hand, polynomial-
time approximation algorithms are tractable for large instances, but may suffer from weak optimality
guarantees or empirical performance. Finally, Heuristics are fast, effective algorithms that lack any
theoretical guarantees. They usually require substantial problem-specific research on the part of
algorithm designers.

2.5 Machine Learning for Combinatorial Optimization

Designing good heuristics for combinatorial optimization problems often requires significant special-
ized knowledge and trial-and-error. Since most problems are highly structured, heuristics are
typically expressed in the form of rules, which can be interpreted as policies to make decisions.
Such policies can be parameterized using deep neural networks and be trained to obtain new and
stronger algorithms for many different problems.

The application of neural networks to combinatorial optimization has a distinguished history, where
the majority of research focuses on TSP (Smith 1999). In addition to TSP, recent work by the
deep learning community has used inductive graph neural networks to learn solutions to other well-
studied problems such as finding Planer Convex Hulls, the Minimum Vertex Cover Problem, and the
Maximum Cut Problem (Vinyals et al. 2015, Dai et al. 2017, Venkatakrishnan et al. 2018). The
OR community has criticized this line of work for showing results solely on small or impractical problem instances.

However, the more profound motivation of using deep learning for combinatorial optimization is not to outperform non-learned, specialized approaches on well-studied problems. Neural networks can be used as a general tool for tackling previously un-encountered NP-hard problems, especially those that are non-trivial to design heuristics for (Bello et al., 2016). See Bengio et al. (2018) for a comprehensive survey of machine learning for combinatorial optimization.

2.6 THE TRAVELLING SALESMAN PROBLEM

The Traveling Salesman Problem (TSP), first formulated in 1930, is one of the most intensively studied combinatorial optimization problems. It asks the following question: “Given a list 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 graph, one needs to search the space of permutations to find an optimal sequence of nodes with minimal total edge weights (tour length). The TSP and its variants have myriad applications in planning, manufacturing, and genetics, among others. It is also used as a benchmark for many optimization methods. See (Applegate et al., 2006) for a comprehensive overview of TSP.

Finding the optimal TSP solution is NP-hard, even in the two-dimensional Euclidean case (Papadimitriou, 1977), where the nodes are 2D points and edge weights are Euclidean distances between pairs of points. In practice, TSP solvers rely on handcrafted heuristics that guide their search procedures to find competitive tours efficiently for graphs with thousands of nodes.

Learning-based approaches to TSP have the potential to be a breakthrough for OR if they are able to firstly learn efficient heuristics on small scale problems and then generalize robustly to larger instances. However, current state-of-the-art deep learning techniques for TSP do not show desirable generalization properties and are unable to outperform handcrafted heuristic solvers in terms of both speed and problem scale (Kool et al., 2019).

2.6.1 NON-LEARNING APPROACHES

The Traveling Salesman Problem is a well studied combinatorial optimization problem. Many exact or approximate algorithms have been proposed for both Euclidean and non-Euclidean graphs. Christofides (1976) proposed a heuristic algorithm that involves computing a minimum-spanning tree and a minimum-weight perfect matching. The algorithm has polynomial running time and returns solutions that are guaranteed to be within a factor of $1.5 \times$ to the optimal solution.

The best known exact dynamic programming algorithm for TSP has a complexity of $\Theta(2^n n^2)$, making it infeasible to scale up to instances with over 40 points. Nevertheless, state-of-the-art TSP solvers are able to solve TSP instances with thousands of nodes using carefully handcrafted heuristics that describe how to navigate the space of feasible solutions efficiently. Concorde (Applegate et al., 2006), widely accepted as one of the best exact TSP solvers, makes use of cutting plane algorithms (Dantzig et al., 1954; Padberg & Rinaldi, 1991; Applegate et al., 2003), iteratively solving linear programming relaxations of the TSP, in conjunction with a branch-and-bound approach that prunes parts of the search space that provably will not contain an optimal solution. Similarly, the Lin-
Kernighan-Helsgaun heuristic (Helsgaun 2000) is a state-of-the-art approximate search heuristic for the symmetric TSP and has been shown to solve instances with thousands of nodes to optimality.

More generic solvers, such as Google’s vehicle routing problem solver (Perron 2011) that tackles a superset of the TSP, typically rely on a combination of local search algorithms and metaheuristics. Local search algorithms apply a specified set of local move operators on candidate solutions, based on hand-engineered heuristics such as 2-opt (Johnson 1990), to navigate from solution to solution in the search space. A metaheuristic is then applied to propose uphill moves and escape local optima. A popular choice of metaheuristic for the TSP and its variants is guided local search (Voudouris & Tsang 1999), which moves out of a local minimum by penalizing particular solution features that it considers should not occur in a good solution.

2.6.2 Learning-based Approaches

Early works on learning-based approaches for TSP proposed the use of Hopfield networks (Hopfield & Tank 1985; Wilson & Pawley 1988) and deformable template models, such as Elastic Nets (Durbin & Willshaw 1987) and Self Organizing Map (Fort 1988; Angeniol et al. 1988), to solve TSP. When being carefully benchmarked, they did not yield satisfying results compared to algorithmic methods (Sarwar & Bhatti 2012; La Maire & Mladenov 2012).

Following recent advances in sequence-to-sequence learning (Sutskever et al. 2014), Vinyals et al. (2015) introduced the sequence-to-sequence Pointer Network (PtrNet) model that uses attention to output a permutation of an input sequence. This model is trained offline to solve the (Euclidean) TSP, supervised via example solutions generated by Concorde. Upon test time, they use a beam search procedure to build valid tours in a fashion similar to neural machine translation (Wu et al. 2016). They are able to closely approximate TSP solutions produced by Concorde for graphs of sizes up to 50 nodes.

Bello et al. (2016) tackle TSP by introducing an Actor-Critic reinforcement learning algorithm to train the PtrNet without supervised solutions. They consider each instance as a training sample and use the cost (tour length) of a sampled solution for an unbiased Monte-Carlo estimate of the policy gradient. They improve upon the results of Vinyals et al. (2015) and address some limitations of supervised learning, such as the need to compute optimal (or at least high quality) TSP solutions as targets, that in turn, may be ill-defined when those solutions are not computed exactly, or when multiple solutions exist.

Dai et al. (2017) introduce TSP solvers based on inductive graph neural networks, which are invariant to node order and better reflect the combinatorial structure of the problem compared to sequence-to-sequence models. They train a graph embedding model (Dai et al. 2016) to output the order in which nodes are inserted into a partial tour, using a helper function to insert at the best possible location. Their 1-step DQN (Mnih et al. 2013) training method trains the algorithm per step. Incremental rewards provided to the agent at every step effectively encourage greedy behavior. They match Bello et al. (2016) in terms of performance and additionally demonstrate their model’s ability to scale up to 1000 nodes.

Nowak et al. (2017) train a graph neural network (Scarselli et al. 2009) in a supervised manner to directly output a tour as an adjacency matrix, which is converted into a feasible solution by a beam search. Unlike the autoregressive approaches of Bello et al. (2016) and Dai et al. (2017), the model cannot condition its output on the partial tour, and performs poorly for very small problem instances.
Concurrent work by Deudon et al. (2018) in the OR community and Kool et al. (2019) in the deep learning community uses the recently proposed Graph Attention Network (Veličković et al., 2017) and attention-based decoder trained with reinforcement learning to autoregressively build TSP solutions. Deudon et al. (2018) show that a hybrid approach of using 2OPT local search (Croes, 1958) on top of tours produced by the model improves performance. Kool et al. (2019) use a different decoder and train the model using REINFORCE (Williams, 1992) with a greedy rollout baseline to achieve state-of-the-art results among learning-based approaches for TSP up to 100 nodes.

Table 1 summarizes recent learning-based approaches to TSP, starting from Vinyals et al. (2015). Similar techniques have also been applied to generalizations of TSP such as the Vehicle Routing Problem (Nazari et al., 2018; Kool et al., 2019) and the multiple TSP (Kaempfer & Wolf, 2018).

Table 1: Recent learning-based approaches to TSP characterized by type of model, training setting and solution search. In the Training Setting column, SL denotes Supervised Learning and RL denotes Reinforcement Learning.

<table>
<thead>
<tr>
<th>Method</th>
<th>Neural Network</th>
<th>Model Type</th>
<th>Training Setting</th>
<th>Solution Search Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vinyals et al. 2015</td>
<td>Pointer Network</td>
<td>Autoregressive</td>
<td>SL</td>
<td>Beam search</td>
</tr>
<tr>
<td>Bello et al. 2016</td>
<td>Pointer Network</td>
<td>Autoregressive</td>
<td>RL</td>
<td>Sample from policy</td>
</tr>
<tr>
<td>Dai et al. 2017</td>
<td>Structure2vec</td>
<td>Autoregressive</td>
<td>RL</td>
<td>Greedy algorithm</td>
</tr>
<tr>
<td>Nowak et al. 2017</td>
<td>Graph Neural Network</td>
<td>Non-autoregressive</td>
<td>SL</td>
<td>Beam search</td>
</tr>
<tr>
<td>Deudon et al. 2018</td>
<td>Graph Attention Network</td>
<td>Autoregressive</td>
<td>RL</td>
<td>Sample from policy</td>
</tr>
<tr>
<td>Kool et al. 2019</td>
<td>Graph Attention Network</td>
<td>Autoregressive</td>
<td>RL</td>
<td>Sample from policy</td>
</tr>
<tr>
<td>Ours</td>
<td>Graph ConvNet</td>
<td>Non-autoregressive</td>
<td>RL</td>
<td>Beam search</td>
</tr>
</tbody>
</table>
3 DATASET

The Travelling Salesman Problem asks the following question: “Given a list 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 graph, one needs to search the space of permutations to find an optimal sequence of nodes, called a tour, with minimal total edge weights (tour length).

Following Vinyals et al. (2015), we generate datasets containing graphs and their corresponding TSP tours to train machine learning models in a supervised setting. In order to directly compare to the current state-of-the-art techniques, separate training, validation and test datasets are generated for graphs of sizes 10, 20, 30, 50 and 100 nodes using the Concorde solver.

3.1 PROBLEM FORMULATION

We focus on the 2D Euclidean TSP in this paper. Given an input graph, represented as a sequence 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 are concerned with finding a permutation of the points \( \hat{\pi} \), termed a tour, that visits each node once and has the minimum total length. We define the length of a tour defined by a permutation \( \hat{\pi} \) as

\[
L(\hat{\pi} \mid s) = x_{\hat{\pi}(n)} - x_{\hat{\pi}(1)}^2 + \sum_{i=1}^{n-1} x_{\hat{\pi}(i)} - x_{\hat{\pi}(i+1)}^2
\]

where \( \cdot_2 \) denotes \( \ell_2 \) norm.

Given the optimal tour permutation \( \pi \), we can minimize a loss function based on \( \hat{\pi} \) and \( \pi \) to train models in an end-to-end manner.

Figure 3: A sample TSP instance of size 30 nodes. The coordinates of each node (in blue) are randomly initialized in \([0, 1]^2\). The optimal tour \( \pi \), starting from the green node, is denoted by the red edges.
3.2 Dataset Preparation

Introduced by Vinyals et al. (2015), the current paradigm for machine learning approaches to TSP is based on training and evaluating model performance on problem instances of fixed sizes. Thus, we create separate training, validation and test datasets for graphs of sizes 10, 20, 30, 50 and 100 nodes. The training sets consists of 1 Million pairs of problem instances and solutions, and the validation and test sets consist of 10,000 pairs each.

For each TSP instance, the $n$ node locations are sampled uniformly at random in the unit square. The optimal tour $\pi$ is found using the Concorde solver (Applegate et al., 2006). We chose Concorde over other optimal solvers such as Gurobi (Inc, 2015) or LKH3 (Helsgaun, 2000) because of its highly specialized nature for 2D Euclidean TSP and its comparatively faster speed.

Summary statistics for the five TSP datasets are presented in Table 2. The approximate solver timings for Concorde are computed on a 32-core CPU server under average load. Clearly, Concorde requires longer durations to find exact solutions as instance size increases. Generating datasets for problems larger than 100 nodes would be impractical on a similar machine. (Generating 1 Million instances of TSP100 took roughly four days.)

Table 2: Summary statistics for TSP datasets. Solver Time is the average time taken by Concorde to solve each instance on a 32-core CPU server under average load. Each Tour Len. column denotes the average TSP tour length over the corresponding set, and the Std. dev. columns denote the standard deviation of the same.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solver Time (approx.)</th>
<th>Training set</th>
<th>Validation set</th>
<th>Test set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tour Len.</td>
<td>Std. dev.</td>
<td>Tour Len.</td>
</tr>
<tr>
<td>TSP10</td>
<td>1 ms</td>
<td>2.869</td>
<td>0.337</td>
<td>2.870</td>
</tr>
<tr>
<td>TSP20</td>
<td>1 ms</td>
<td>3.829</td>
<td>0.304</td>
<td>3.830</td>
</tr>
<tr>
<td>TSP30</td>
<td>1 ms</td>
<td>4.555</td>
<td>0.281</td>
<td>4.540</td>
</tr>
<tr>
<td>TSP50</td>
<td>10 ms</td>
<td>5.693</td>
<td>0.252</td>
<td>5.693</td>
</tr>
<tr>
<td>TSP100</td>
<td>250 ms</td>
<td>7.766</td>
<td>0.230</td>
<td>7.765</td>
</tr>
</tbody>
</table>
4 MODEL

[Bresson & Laurent (2017)] proposed the most generic and powerful formulation of Graph ConvNets for learning representations on variable graphs by combining the vanilla Graph ConvNet architecture of [Sukhbaatar et al. (2016)] with edge gating [Marcheggiani & Titov (2017)] and residual connections [He et al. (2016)].

Given a graph as an input, we train a Residual Gated Graph ConvNet model to directly output an adjacency matrix corresponding to a TSP tour. The ConvNet computes $h$-dimensional embedding representations for each node and edge in the graph. These embeddings are linked to the groundtruth TSP tour through a softmax output layer so that the model parameters can be trained end-to-end by minimizing the cross-entropy loss via gradient descent. During test time, the adjacency matrix obtained from the model is converted to a valid tour via beam search.

4.1 GENERIC FORMULATION OF CONVNETS

Consider a classical ConvNet for computer vision. Let $h_{ij}^\ell$ denote the feature vector at layer $\ell$ associated with pixel $(i, j)$. In a regular ConvNet, $h_{ij}^{\ell+1}$ is obtained by applying a non linear transformation to the feature vectors $h_{i'j'}^{\ell}$ for all pixels $(i', j')$ in a neighborhood of pixel $(i, j)$. For example, with $3 \times 3$ filters, we would have:

$$h_{ij}^{\ell+1} = f_{\text{CNN}}^{\ell}(\{h_{i'j'}^{\ell} : |i - i'| \leq 1 \text{ and } |j - j'| \leq 1\})$$ (2)

In the above, the notation $\{h_{i'j'}^{\ell} : |i - i'| \leq 1 \text{ and } |j - j'| \leq 1\}$ denote the concatenation of all feature vectors $h_{i'j'}^{\ell}$ belonging to the $3 \times 3$ neighborhood of vertex $(i, j)$.

For Graph ConvNets, the notion of neighborhood is given by the graph structure instead of the euclidean distance. Thus, the most generic version of a feature vector $h_i$ at vertex $i$ for a graph ConvNet is:

$$h_i^{\ell+1} = f_{\text{G-CNN}}^{\ell}(h_i^{\ell}, \{h_j^{\ell} : j \rightarrow i\})$$ (3)

where $\{h_j^{\ell} : j \rightarrow i\}$ denotes the set of feature vectors of the neighboring vertices. In other words, to define a Graph ConvNet, one needs a mapping $f_{\text{G-CNN}}^{\ell}$ taking as input a vector $h_i^{\ell}$ (the feature vector of the center vertex) as well as an unordered set of vectors $\{h_j^{\ell}\}$ (the feature vectors of all neighboring vertices). We also refer to the mapping $f_{\text{G-CNN}}^{\ell}$ as the neighborhood transfer function.

We additionally define a feature vector $e_{ij}$ on the edge between vertex $i$ and vertex $j$, which is computed as:

$$e_{ij}^{\ell+1} = g_{\text{G-CNN}}^{\ell}(e_{ij}^{\ell}, h_i^{\ell}, h_j^{\ell})$$ (4)

4.2 MODEL ARCHITECTURE

**Input Layer** As inputs, we are given the two dimensional coordinates $x_i \in [0, 1]^2$ for $n$ nodes in the graph. The coordinates are converted to $h$-dimensional node inputs for $n$ nodes and $h$-dimensional edge inputs for $n^2$ edges, which undergo $l_{\text{conv}}$ layers of graph convolution operations to compute $h$-dimensional embeddings for each node and edge in the graph.

The node inputs $\alpha_i$ are computed via a simply linear transformation:

$$\alpha_i = \theta_1 x_i$$ (5)

where $\theta_1 \in \mathbb{R}^{h \times 2}$ is a learnable model parameter.
To compute edge inputs $\beta_{ij}$ for an edge between nodes $i$ and $j$, we first compute the edge distance matrix $W_{\text{dist}}$, where $w_{\text{dist}}^{ij}$ corresponds to the euclidean distance between nodes $i$ and $j$, and the edge adjacency matrix $W_{\text{adj}}$, where $w_{\text{adj}}^{ij}$ denotes the presence/absence of edges between nodes $i$ and $j$ in a $k$-nearest neighbor graph. The edge input $\beta_{ij}$ is computed by concatenating linear transformations of $w_{\text{dist}}^{ij}$ and $w_{\text{adj}}^{ij}$:

$$\beta_{ij} = [ \theta_2 w_{\text{dist}}^{ij}, \theta_3 w_{\text{adj}}^{ij} ]$$

where $\theta_2 \in \mathbb{R}^{\frac{h}{2} \times 1}$, $\theta_3 \in \mathbb{R}^{\frac{h}{2} \times 2}$, and $[\cdot, \cdot]$ is the concatenation operator.

**Graph Convolution Layers** Our Residual Gated Graph ConvNet architecture is characterized by the following neighborhood transfer function for node feature vectors $h_i$:

$$h_{i}^{\ell+1} = f^\ell( h_i^\ell, \{ h_j^\ell : j \rightarrow i \} ) = \text{ReLU} \left( \theta_4^\ell h_i^\ell + \sum_{j \rightarrow i} \eta_{ij}^\ell \odot \theta_5^\ell h_j^\ell \right)$$

where $h_i^{\ell=0} = \alpha_i$, $\theta_4^\ell \in \mathbb{R}^{h \times h}$, $\theta_5^\ell \in \mathbb{R}^{h \times h}$, $\eta_{ij}^\ell$ are the edge gates, $\epsilon$ is a small positive constant to avoid division by zeros, and ReLU is the rectified linear unit ($\text{ReLU}(z) = \max(0, z)$) applied element-wise to its input.

The edge gates $\eta_{ij}$ are computed as:

$$\eta_{ij}^{\ell+1} = \sigma( \theta_6^\ell e_{ij}^\ell + \theta_7^\ell ( h_i^\ell + h_j^\ell ) )$$

where $\theta_6^\ell \in \mathbb{R}^{h \times h}$, $\theta_7^\ell \in \mathbb{R}^{h \times h}$, $e_{ij}^\ell$ are the edge feature vectors, and $\sigma$ is the sigmoid function applied element-wise to its input.

The edge feature vector $e_{ij}$ is computed using the same parameters $\theta_6$ and $\theta_7$ as the edge gates $\eta_{ij}$:

$$e_{ij}^{\ell+1} = g^\ell( e_{ij}^\ell, h_i^\ell, h_j^\ell ) = \text{ReLU} \left( \theta_6^\ell e_{ij}^\ell + \theta_7^\ell ( h_i^\ell + h_j^\ell ) \right)$$

where $e_{ij}^{\ell=0} = \beta_{ij}$.

**Figure 4:** The proposed Graph Convolution layer for computing $h$-dimensional representations $h_i$ for each node $i$ and $e_{ij}$ for the edge between each node $i$ and $j$ in the graph. Multiple layers of graph convolution are applied to progressively extract compositional features of the input graph.
Equation [7] can be termed as an edge-weighted version of the mean node aggregation scheme in generic Graph ConvNets (Xu et al., 2018). We believe that the edge gating mechanism is important for the model as it will be able to learn what types of edges are important for solving the problem.

Empirical experiments by Bresson & Laurent (2017) show that Graph ConvNets are able to learn more powerful representations of graphs by stacking convolution layers. To enable the training of very deep architectures, we perform batch normalization (Ioffe & Szegedy, 2015) to the node and edge feature vectors before the ReLU function in Equations 7 and 9.

We also add residual connections between successive convolutional layers by adding the identity operator to the node and edge feature vectors:

\[ h_{i}^{\ell+1} = f^{\ell} \left( h_{j}^{\ell}, \{ h_{j}^{\ell}, j \rightarrow i \} \right) + h_{i}^{\ell} \]  
\[ e_{ij}^{\ell+1} = g^{\ell} \left( e_{ij}^{\ell}, h_{i}^{\ell}, h_{j}^{\ell} \right) + e_{ij}^{\ell} \]  

This allows us to further stack convolution layers by providing an additional path in the computational graph where backpropagation is not affected by the vanishing gradient problem.

**Multi-layer Perceptron Classifier**  The final \( h \)-dimensional embedding \( e_{ij}^{\ell} \) for each edge in the input graph is used to compute the probability of that edge being connected in the TSP tour of the graph. This can be seen as computing a probabilistic heat-map \( W^{\text{TSP}} \) over the adjacency matrix of tour connections. Each \( w_{ij}^{\text{TSP}} \in [0, 1]^2 \) is given by a multi-layer perceptron with softmax output as follows:

\[ w_{ij}^{\text{TSP}} = \text{softmax} \left( \theta_{9} \text{ReLU} \left( \theta_{8} e_{ij}^{\ell} \right) \right) \]  

where \( \theta_{8} \in \mathbb{R}^{h \times h} \) and \( \theta_{9} \in \mathbb{R}^{2 \times h} \).

For simplicity, Equation [12] shows a 2-layer perceptron. In practice, we may have an arbitrary number of layers denoted by \( l_{\text{mlp}} \).

**4.3 Loss Function**

Given the groundtruth TSP tour permutation \( \pi \), we convert the tour into an adjacency matrix \( Y^{\text{TSP}} \). Each \( y_{ij} \) denotes the presence or absence of an edge between nodes \( i \) and \( j \) in the TSP tour. The prediction for each edge \( \hat{y}_{ij} \) is simply the second dimension (the probability of the positive class, i.e. an edge) of each \( w_{ij}^{\text{TSP}} \) in the probabilistic heat-map \( W^{\text{TSP}} \).

The trainable model parameters \( \Theta = \{ \theta_{i} \}_{i=1}^{9} \) are learned end-to-end via gradient descent. We minimize the binary cross-entropy loss averaged over each of the \( n^2 \) edges of \( m \) problem instances in mini-batch \( B \):

\[ L \left( \Theta, B \right) = \frac{1}{m} \sum_{i=1}^{m} \left( \sum_{j \rightarrow i} - \left( y_{ij} \log(\hat{y}_{ij}) + (1 - y_{ij}) \log(1 - \hat{y}_{ij}) \right) \right) \]  

As the size of the problem instances increases, the classification task becomes highly imbalanced towards the negative class. This corresponds to having a very high ratio of non-edges to edges in \( Y^{\text{TSP}} \). In practice, we use the weighted binary cross entropy loss and automatically compute balanced class weights to scale up to larger TSP instances.
4.4 Tour Decoder

The output of our model is a probabilistic heat-map $W_{TSP}$ over the adjacency matrix of tour connections. Each $w_{ij}^{TSP} \in [0, 1]^2$ denotes the strength of the edge prediction between nodes $i$ and $j$. Based on the chain rule of probability, the probability of a partial TSP tour $\pi'$ can be formulated as:

$$p(\pi') = \prod_{j' \rightarrow i'} w_{ij}^{TSP}$$

(14)

where each node $j'$ follows node $i'$ in the partial tour $\pi'$.

However, directly converting the probabilistic heat-map $W_{TSP}$ to an adjacency matrix representation of the predicted TSP tour $\hat{\pi}$ via an argmax function will generally give invalid tours or extra edges in $\hat{\pi}$. Thus, we employ three possible search strategies at evaluation time to convert $W_{TSP}$ to a valid permutation of nodes denoting $\hat{\pi}$.

**Greedy search** In general, greedy algorithms choose the local optimal solution with the hope that this will lead to the global optimal solution. Starting from the first node, we greedily select the next node from its neighbors based on the highest probability of the presence of an edge. The search terminates when all nodes have been visited. We mask out nodes that have previously been visited in order to construct valid solutions.

It is interesting to draw a parallel between our probabilistic greedy search and a popular distance-based greedy strategy for TSP. The distance-based heuristic is described as: “At each step of the journey, visit the nearest unvisited city.” Our greedy search can similarly be described as: “At each step of the journey, take the most probable path to an unvisited city.”

**Beam search** A beam search is a limited-width breadth-first search. In the context of sequence-to-sequence models in natural language generation, it is often used to obtain a set of high-probability sequences from the model (Wu et al., 2016; Wiseman & Rush, 2016).

Starting from an empty sequence (e.g. $t = 0$), a beam search expands at every step $t = 0, 1, 2, ...$ at most $b$ partial sequences (those with highest probability) to compute the probabilities of sequences with length $t + 1$. It terminates with a beam of $b$ complete sequences. (Note that $b$ is referred to as the beam width.)

Similarly, starting from the first node, we explore the heat-map by expanding the $b$ most probable edge connections among the node’s neighbors. We iteratively expand the top-$b$ partial tours at each stage till we have visited all nodes in the graph. We follow the same masking strategy as greedy search to construct valid tours. The final prediction is the tour with the highest probability among the $b$ tours at the end of beam search.

Other supervised learning-based techniques for TSP (Vinyals et al., 2015; Nowak et al., 2017) follow the same search strategy to generate valid solutions.

**Beam search and shortest tour heuristic** Instead of selecting the tour with the highest probability at the end of beam search, we select the shortest tour among the set of $b$ final tours as the final solution.

This heuristic-based beam search is directly comparable to reinforcement learning-based techniques for TSP (Bello et al., 2016; Deudon et al., 2018; Kool et al., 2019) which sample a set of solutions from the learned policy and select the shortest tour among the set as the final solution.
4.5 Hyperparameter Configurations

Table 3 presents the model hyperparameters used for the five problem instances described in Section 3. As problem size increases, we increment the hyperparameters arbitrarily to add more capacity to the model. We use a fixed beam width $b = 1,280$ in order to directly compare our results to the current state-of-the-art (Kool et al., 2019) which samples 1,280 solutions from a learned policy.

Table 3: Model hyperparameter configurations for various TSP instances. The $l_{\text{conv}}$ column denotes number of graph convolution layers, $l_{\text{mlp}}$ denotes number of layers in the multi-layer perceptron, $h$ denotes the hidden dimension for the model, $k$ denotes the number of neighbors considered for each node in the $k$-nearest neighbor adjacency matrix $W^{\text{adj}}$, $b$ denotes beam search width, and Parameters lists the total number of trainable parameters for each model.

<table>
<thead>
<tr>
<th>Problem</th>
<th>$l_{\text{conv}}$</th>
<th>$l_{\text{mlp}}$</th>
<th>$h$</th>
<th>$k$</th>
<th>$b$</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP10</td>
<td>5</td>
<td>2</td>
<td>100</td>
<td>5</td>
<td>1,280</td>
<td>214,702</td>
</tr>
<tr>
<td>TSP20</td>
<td>10</td>
<td>2</td>
<td>200</td>
<td>10</td>
<td>1,280</td>
<td>1,657,402</td>
</tr>
<tr>
<td>TSP30</td>
<td>15</td>
<td>2</td>
<td>200</td>
<td>10</td>
<td>1,280</td>
<td>2,465,402</td>
</tr>
<tr>
<td>TSP50</td>
<td>20</td>
<td>3</td>
<td>200</td>
<td>10</td>
<td>1,280</td>
<td>3,313,602</td>
</tr>
<tr>
<td>TSP100</td>
<td>30</td>
<td>3</td>
<td>300</td>
<td>20</td>
<td>1,280</td>
<td>11,054,402</td>
</tr>
</tbody>
</table>
5 Experiments

Introduced by Vinyals et al. (2015), the current paradigm for machine learning approaches to TSP is based on training and evaluating model performance on problem instances of fixed sizes. We train five models using training sets of 1 Million problem instances with 10, 20, 30, 50 and 100 nodes each, and evaluate them on held-out test sets of 10,000 instances of the same size. We report metrics for empirical performance compared to optimal solutions (obtained using Concorde) and time taken to solve the test set.

Additionally, we perform experiments to evaluate the transfer learning capabilities of our architecture. We test each model trained on instances of a fixed size on instances of other sizes in order to measure the model’s extent of generalization to arbitrarily large graphs.

5.1 Training Procedure

We follow a standard training procedure to train models for each of the five problem instances. Given a graph as an input, we train our Graph ConvNet model to directly output an adjacency matrix corresponding to a TSP tour by minimizing the cross-entropy loss via gradient descent. See Section 4 for details of the model architecture and loss function.

Training Loop For each training epoch, we randomly select a subset of 10,000 problem instances out of 1 Million from the training set. The subset is divided into 500 mini-batches of 20 instances each. We use the Adam optimizer (Kingma & Ba, 2014) with an initial learning rate of $1 \times 10^{-3}$ to minimize the cross-entropy loss over each mini-batch (Equation 13).

Learning Rate Decay We evaluate our model on a held-out validation set of 10,000 instances at regular intervals of five training epochs. If the validation loss has not decreased by at least 1% of the previous validation loss, we divide the optimizer’s learning rate by a decay factor of 1.01. Using smaller learning rates as training progresses allows our models to learn faster and converge to better local minima. Larger problem instances require more training epochs and lower learning rates to reach convergence.

5.2 Evaluation Procedure

During evaluation on the validation and test sets, the adjacency matrix obtained from the model is converted to a valid tour via search strategies described in Section 4.4. As we do not need to do backpropagation during evaluation, we use arbitrarily large batch sizes that fit the entire GPU memory. Following Kool et al. (2019), we report the following metrics to evaluate performance of our models compared to optimal solutions (obtained using Concorde).

Predicted tour length The average predicted TSP tour length $\hat{c}$ over 10,000 test instances, computed as $\frac{1}{m} \sum_{i=1}^{m} \hat{c}_i$.

Optimality gap The average percentage ratio of the predicted tour length $\hat{c}$ relative to the optimal solution $c$ over 10,000 test instances, computed as $\frac{1}{m} \sum_{i=1}^{m} \left( \frac{\hat{c}_i}{c_i} - 1 \right)$.

Evaluation time The total wall clock time it takes to solve 10,000 test instances, either on a single GPU (Nvidia 1080Ti) or 32 instances in parallel on a 32 virtual CPU system ($2 \times$ Xeon E5-2630).
5.3 ON COMPARISON WITH STATE-OF-THE-ART

As noted in Section 4.4, all deep learning approaches use search or sampling techniques to predict solutions. For example, Bello et al. (2016) and Kool et al. (2019) sample 1,280 solutions from the learned policy and select the shortest tour among the set as the final solution. Thus, it is possible to trade off run time for performance by searching for longer or sampling many solutions, in which case even a random policy performs well.

Run times can vary by two orders of magnitude as a result of implementation (Python vs C++) or hardware (GPU vs CPU), and are hard to compare. Kool et al. (2019) take a practical view and report the time it takes to solve the test set of 10,000 instances. This is conservative: deep learning models are parallelizable while their non-learning baselines are single thread CPU implementations which cannot parallelize when running individually. Kool et al. (2019) do not report running times for results reported by others as they are not directly comparable, but in general they find that their model and implementation performs better and faster than Bello et al. (2016).

For direct comparison with Kool et al. (2019), we report the test set performance and run time of our models on a single Nvidia 1080Ti GPU using the heuristic-based beam search decoder with beam width of 1,280. (See Section 4.4 for details.)

5.4 EVALUATING GENERALIZATION VIA TRANSFER LEARNING

Strong generalization across TSP instances of various sizes is a highly desirable property for deep learning approaches. Generalizing to problem sizes larger than what the model was trained for would allow us to scale up to very large TSP instances while training efficiently on smaller instances. In theory, since our model’s parameters Θ are independent of the size of an instance n, we can use a model trained on smaller graphs to solve arbitrarily large instances.

To evaluate the generalization of our models, we use each model for size n to compute performance metrics on the test sets of instances of all other sizes.
6 RESULTS

6.1 FIXED SIZED TSP Instances

Table 4 presents timing and performance metrics of our Graph ConvNet models for TSP instances of sizes 10, 20, 30, 50 and 100. Figure 5 plots the validation optimality gap (computed after beam search) vs. the training epochs for our models. See Appendix A for additional training logs.

For smaller instances such as TSP10 and TSP20, our model is able to get very close to optimal solutions using beam search within as few as 10 training epochs. The model requires more training epochs to learn close to optimal solutions for larger problem instances such as TSP50 and TSP100. Using beam search instead of greedy search improves our performance drastically for larger instances, but leads to an increase in evaluation time. Adding the shortest tour heuristic to our beam search further boosts our performance at the cost of longer evaluation time.

Table 4: Timing and performance of our technique for various TSP instance sizes. The three possible decoding strategies for the same model are denoted by \textbf{G}: Greedy, \textbf{BS}: Beam search, and \textbf{BS*}: Beam search and shortest tour heuristic. (See Section 4.4 for details.)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>G  BS BS*</td>
<td>G  BS BS*</td>
<td>G  BS BS*</td>
<td>G  BS BS*</td>
<td>G  BS BS*</td>
<td></td>
</tr>
<tr>
<td>TSP10</td>
<td>200</td>
<td>10s</td>
<td>2.867 2.879 2.879</td>
<td>2.867 0.66% 0.10%</td>
<td>2.886 6.26% 0.42%</td>
<td>2.886 6.26% 0.42%</td>
<td>2.879 6.26% 0.42%</td>
<td>2s 5s 10m</td>
</tr>
<tr>
<td>TSP20</td>
<td>500</td>
<td>20s</td>
<td>3.831 3.842 3.831</td>
<td>3.831 1.46% 0.28%</td>
<td>3.887 7.94% 0.28%</td>
<td>3.887 7.94% 0.28%</td>
<td>3.842 7.94% 0.28%</td>
<td>6s 20s 12m</td>
</tr>
<tr>
<td>TSP30</td>
<td>500</td>
<td>45s</td>
<td>4.539 4.551 4.539</td>
<td>4.539 2.37% 0.26%</td>
<td>4.647 10.41% 0.26%</td>
<td>4.647 10.41% 0.26%</td>
<td>4.551 10.41% 0.26%</td>
<td>15s 40s 14m</td>
</tr>
<tr>
<td>TSP50</td>
<td>1,000</td>
<td>2m</td>
<td>5.692 5.716 5.694</td>
<td>5.694 4.37% 0.42%</td>
<td>5.941 12.85% 0.42%</td>
<td>5.941 12.85% 0.42%</td>
<td>5.716 12.85% 0.42%</td>
<td>55s 2m 18m</td>
</tr>
<tr>
<td>TSP100</td>
<td>1,700</td>
<td>9m</td>
<td>7.764 7.924 7.874</td>
<td>7.874 8.23% 2.06%</td>
<td>8.403 15.95% 1.41%</td>
<td>8.403 15.95% 1.41%</td>
<td>7.924 15.95% 1.41%</td>
<td>6m 10m 40m</td>
</tr>
</tbody>
</table>

Figure 5: Validation Optimality Gap vs Training Epochs for various runs.
Table 5: Performance of our technique compared to non-learned baselines and state-of-the-art methods for various TSP instance sizes. Deep learning approaches are named according to the type of neural network used. The optimality gap is computed w.r.t Concorde. In the Type column, H: Heuristic, SL: Supervised Learning, RL: Reinforcement Learning, S: Sampling, G: Greedy, BS: Beam search, BS*: Beam search and shortest tour heuristic, and 2OPT: 2OPT local search.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Concorde</td>
<td>Solver</td>
<td>3.84 0.00%</td>
<td>(1s)</td>
<td>5.70 0.00%</td>
<td>(2m)</td>
<td>7.76 0.00%</td>
<td>(3m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LKH3</td>
<td>Solver</td>
<td>3.84 0.00%</td>
<td>(18s)</td>
<td>5.70 0.00%</td>
<td>(5m)</td>
<td>7.76 0.00%</td>
<td>(21m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gurobi</td>
<td>Solver</td>
<td>3.84 0.00%</td>
<td>(7s)</td>
<td>5.70 0.00%</td>
<td>(2m)</td>
<td>7.76 0.00%</td>
<td>(17m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nearest Insertion</td>
<td>H, G</td>
<td>4.33 12.93%</td>
<td>(1s)</td>
<td>6.78 19.03%</td>
<td>(2s)</td>
<td>9.46 21.82%</td>
<td>(6s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Insertion</td>
<td>H, G</td>
<td>4.00 4.96%</td>
<td>(3s)</td>
<td>6.13 7.65%</td>
<td>(1s)</td>
<td>8.52 9.69%</td>
<td>(3s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farthest Insertion</td>
<td>H, G</td>
<td>3.93 2.36%</td>
<td>(1s)</td>
<td>6.01 5.53%</td>
<td>(2s)</td>
<td>8.35 7.59%</td>
<td>(7s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nearest Neighbor</td>
<td>H, G</td>
<td>4.50 17.23%</td>
<td>(8s)</td>
<td>7.00 22.94%</td>
<td>(6s)</td>
<td>9.68 24.73%</td>
<td>(6s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrNet</td>
<td>SL, G</td>
<td>3.88 1.15%</td>
<td></td>
<td>7.66 34.48%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrNet</td>
<td>RL, G</td>
<td>3.89 1.42%</td>
<td></td>
<td>5.95 4.46%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2V</td>
<td>RL, G</td>
<td>3.89 1.42%</td>
<td></td>
<td>5.99 5.16%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT</td>
<td>RL, G</td>
<td>3.86 0.66%</td>
<td>(2m)</td>
<td>5.92 3.98%</td>
<td>(5m)</td>
<td>8.42 8.41%</td>
<td>(8m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT</td>
<td>RL, G</td>
<td>3.85 0.34%</td>
<td>(8s)</td>
<td>5.80 1.76%</td>
<td>(2s)</td>
<td>8.12 4.53%</td>
<td>(6s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCN (Ours)</td>
<td>SL, G</td>
<td>3.89 1.46%</td>
<td>(6s)</td>
<td>5.94 4.37%</td>
<td>(55s)</td>
<td>8.40 8.23%</td>
<td>(6m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR Tools</td>
<td>H, S</td>
<td>3.85 0.37%</td>
<td></td>
<td>5.80 1.83%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chef + 2OPT</td>
<td>H, 2OPT</td>
<td>3.85 0.37%</td>
<td></td>
<td>5.79 1.65%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNN</td>
<td>SL, BS</td>
<td>3.93 2.46%</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrNet</td>
<td>RL, S</td>
<td>-</td>
<td></td>
<td>5.75 0.95%</td>
<td></td>
<td>8.00 3.03%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT</td>
<td>RL, S</td>
<td>3.84 0.11%</td>
<td>(6m)</td>
<td>5.77 1.28%</td>
<td>(17m)</td>
<td>8.75 12.70%</td>
<td>(56m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT</td>
<td>RL, 2OPT</td>
<td>3.85 0.42%</td>
<td>(4m)</td>
<td>5.85 2.77%</td>
<td>(26m)</td>
<td>8.17 5.21%</td>
<td>(3h)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT</td>
<td>RL, S, 2OPT</td>
<td>3.84 0.06%</td>
<td>(6m)</td>
<td>5.75 1.00%</td>
<td>(32m)</td>
<td>8.12 4.64%</td>
<td>(5h)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT</td>
<td>RL, S</td>
<td>3.84 0.08%</td>
<td>(5m)</td>
<td>5.73 0.52%</td>
<td>(24m)</td>
<td>7.94 2.26%</td>
<td>(1h)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCN (Ours)</td>
<td>SL, BS</td>
<td>3.85 0.28%</td>
<td>(20s)</td>
<td>5.72 0.42%</td>
<td>(2m)</td>
<td>7.92 2.06%</td>
<td>(10m)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCN (Ours)</td>
<td>SL, BS*</td>
<td>3.84 0.00%</td>
<td>(12m)</td>
<td>5.70 0.03%</td>
<td>(18m)</td>
<td>7.87 1.41%</td>
<td>(40m)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For faster training, TSP100 models were trained using four Nvidia 1080Ti GPUs. However, it is not essential to use a multi-GPU setup to train TSP100 models: the same results can be attained with a single GPU at the cost of longer training time. Evaluation of TSP100 models on the test set was done with two GPUs in order to directly compare with Kool et al. (2019). All other timing results are reported for single GPU models.

6.2 Comparison to State-of-the-Art

Table 5 presents the performance of our technique compared to non-learned baselines and state-of-the-art deep learning techniques for various TSP instance sizes. The table is divided into three sections: exact solvers, greedy methods, and sampling/search-based methods. Methods are further categorized according to the training technique: supervised learning, reinforcement learning, and non-learned heuristics.

All results except ours (in bold) are taken from Table 1 in Kool et al. (2019). More details about solvers (Concorde, LKH3 and Gurobi) and non-learned heuristic baselines (Nearest Insertion, Random Insertion, Farthest Insertion, Nearest Neighbor) can be found in Appendix B of Kool et al. (2019).
Greedy setting  In the greedy setting, learning-based approaches clearly outperform all non-learned heuristics. All deep learning approaches are able to solve TSP20 within 1.5% of optimality, which suggests that the instance size is too simple for comparing models. For the larger instance sizes TSP50 and TSP100, graph neural network approaches (Deudon et al., 2018; Kool et al., 2019) generally outperform sequence-to-sequence models (Vinyals et al., 2015; Bello et al., 2016).

Our Graph ConvNet model is not able to match the performance or evaluation time of the Graph Attention Network model of Kool et al. (2019) in the greedy setting. However, we feel it is unfair to compare our non-autoregressive model with autoregressive approaches in this setting, especially in terms of evaluation time. Autoregressive models directly output the TSP tour permutation node-by-node, conditioning each prediction on the partial tour. In contrast, our model’s prediction of an edge between any pair of nodes is independent of other edge predictions. Moreover, the two-step process of getting the output from our model and performing greedy search to convert it into a valid tour adds additional time overhead.

Search/Sampling setting  In general, all learning-based approaches are able to improve performance by searching or sampling for solutions. Our Graph ConvNet model with beam search outperforms the best performing model of Kool et al. (2019) in terms of both closeness to optimality and evaluation time in this setting.

We attribute our gains in performance to more powerful representation learning for the input graphs through our use of very deep architectures with up to 30 graph convolution layers. In contrast, Kool et al. (2019) use only 3 graph attention layers. Despite larger models being more computationally expensive, our beam search implementation is highly parallelized, leading to significantly faster evaluation compared to sampling from a reinforcement learning policy.

Combining learned and traditional heuristics  As seen from the results of Deudon et al. (2018), autoregressive models may not produce a local optimum and performance can improve by using a hybrid approach of a learned algorithm with a local search heuristic such as 2-OPT. The same property holds for our non-autoregressive model. Adding the shortest tour heuristic to beam search boosts our performance at the cost of evaluation time. We may further trade-off computation for better performance by incorporating more heuristics such as 2-OPT into our beam search decoder.

Drawbacks  Our model outperforms all other learning-based approaches in terms of both solution quality and speed. However, it is important to highlight a major drawback of our supervised learning setup: generating datasets and training models for instances beyond TSP100 is extremely costly in terms of computation and speed. The rapid increase in combinatorial complexity of TSP as problem size increases, termed as combinatorial explosion, makes it intractable to scale our approach to realistic problem sizes beyond hundreds of nodes.

Hence, reinforcement learning, which does not require the creation of labelled datasets, has been the more popular choice of training framework for learning-based approaches to TSP.
6.3 Visualizations

Figures 6, 7, 8, 9 and 10 display prediction visualizations for samples from test sets of various problem instances. In each figure, the first panel shows the input $k$-nearest neighbor graph and the groundtruth TSP tour. The second panel represents the probabilistic heat-map output of the Graph ConvNet model. The final panel shows the predicted TSP tour after a beam search procedure on the heat-map.

For small instances such as TSP10, the model is able to confidently identify most of the tour edges in the heat-map, resulting in greedy search being able to find close to optimal tours. As instance size increases, the prediction heat-map reflects the combinatorial explosion in TSP. Beam search is essential for finding the optimal tour for more complex instances.

See Appendix B for additional visualizations and discussions.

Figure 6: TSP10 model prediction visualization for a sample test set instance.

Figure 7: TSP20 model prediction visualization for a sample test set instance.
Figure 8: TSP30 model prediction visualization for a sample test set instance.

Figure 9: TSP50 model prediction visualization for a sample test set instance.

Figure 10: TSP100 model prediction visualization for a sample test set instance.
6.4 Generalization to Variable Sized TSPs

Generalizing to problem sizes larger than what the model was trained for would allow us to scale up to very large TSP instances while training efficiently on smaller instances. Since the parameters \( \Theta \) of our Graph ConvNet are independent of the size of an instance \( n \), we can evaluate the generalization of our models by computing performance metrics on test set instances of variable sizes.

Tables 6, 7 and 8 present the optimality gap of our models when evaluated on variable instance sizes using greedy search, beam search and beam search with the shortest tour heuristic, respectively. We observe drastic drops in performance for our models in all settings, indicating very poor generalization capabilities.

It is clear that the Graph ConvNet models have overfit to their respective training instance sizes. The representations learned by the models have memorized patterns for the specific graph size, and are not able to transfer these patterns to new graph sizes.

Table 6: Generalization performance of our technique to variable instance sizes in the greedy setting.

<table>
<thead>
<tr>
<th>Problem</th>
<th>TSP10 Model</th>
<th>TSP20 Model</th>
<th>TSP30 Model</th>
<th>TSP50 Model</th>
<th>TSP100 Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP10</td>
<td>0.66%</td>
<td>7.00%</td>
<td>38.65%</td>
<td>26.53%</td>
<td>28.45%</td>
</tr>
<tr>
<td>TSP20</td>
<td>18.60%</td>
<td>1.46%</td>
<td>10.36%</td>
<td>23.87%</td>
<td>38.05%</td>
</tr>
<tr>
<td>TSP30</td>
<td>39.32%</td>
<td>17.60%</td>
<td>2.37%</td>
<td>18.56%</td>
<td>41.40%</td>
</tr>
<tr>
<td>TSP50</td>
<td>65.03%</td>
<td>42.14%</td>
<td>34.23%</td>
<td>4.37%</td>
<td>38.29%</td>
</tr>
<tr>
<td>TSP100</td>
<td>119.00%</td>
<td>70.53%</td>
<td>54.14%</td>
<td>52.87%</td>
<td>8.23%</td>
</tr>
</tbody>
</table>

Table 7: Generalization performance of our technique to variable instance sizes using beam search.

<table>
<thead>
<tr>
<th>Problem</th>
<th>TSP10 Model</th>
<th>TSP20 Model</th>
<th>TSP30 Model</th>
<th>TSP50 Model</th>
<th>TSP100 Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP10</td>
<td>0.42%</td>
<td>4.04%</td>
<td>26.02%</td>
<td>20.84%</td>
<td>23.13%</td>
</tr>
<tr>
<td>TSP20</td>
<td>8.62%</td>
<td>0.28%</td>
<td>2.22%</td>
<td>14.98%</td>
<td>31.66%</td>
</tr>
<tr>
<td>TSP30</td>
<td>26.37%</td>
<td>4.36%</td>
<td>0.26%</td>
<td>6.24%</td>
<td>28.42%</td>
</tr>
<tr>
<td>TSP50</td>
<td>57.33%</td>
<td>33.87%</td>
<td>25.22%</td>
<td>0.42%</td>
<td>31.46%</td>
</tr>
<tr>
<td>TSP100</td>
<td>114.50%</td>
<td>70.30%</td>
<td>53.28%</td>
<td>44.43%</td>
<td>2.06%</td>
</tr>
</tbody>
</table>

Table 8: Generalization performance of our technique to variable instance sizes using beam search and shortest tour heuristic.

<table>
<thead>
<tr>
<th>Problem</th>
<th>TSP10 Model</th>
<th>TSP20 Model</th>
<th>TSP30 Model</th>
<th>TSP50 Model</th>
<th>TSP100 Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP10</td>
<td>0.00%</td>
<td>0.00%</td>
<td>5.04%</td>
<td>3.44%</td>
<td>2.05%</td>
</tr>
<tr>
<td>TSP20</td>
<td>1.91%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>6.05%</td>
<td>14.96%</td>
</tr>
<tr>
<td>TSP30</td>
<td>15.12%</td>
<td>1.67%</td>
<td>0.00%</td>
<td>2.72%</td>
<td>19.92%</td>
</tr>
<tr>
<td>TSP50</td>
<td>49.82%</td>
<td>27.26%</td>
<td>19.06%</td>
<td>0.03%</td>
<td>24.64%</td>
</tr>
<tr>
<td>TSP100</td>
<td>110.23%</td>
<td>66.07%</td>
<td>48.06%</td>
<td>39.76%</td>
<td>1.41%</td>
</tr>
</tbody>
</table>
Figure 11: Optimality gap of different methods as a function of problem size. Non-learnt baselines are drawn using dashed lines while learning-based approaches are drawn with a solid line. Approaches that perform search or sampling are plotted without connecting lines for clarity. Adapted from Figure 5 in Kool et al. (2019).

Kool et al. (2019) notice a similar albeit less drastic trend when they plotted the optimality gap of recently proposed learning-based approaches evaluated on variable instance sizes (Figure 11). They find the performance of all approaches degrades as the difference between training and evaluation instance size increases. This is in contrast to handcrafted solvers such as Concorde, which perform equally strongly on small and large TSP instances. This result indicates that strong performance on problem instances of the same size as those used for training is extremely misleading for comparing among approaches.

6.5 Comments on Recent Work

Supervised Learning vs. Reinforcement Learning As noted in Bengio et al. (2018), the performance of supervised learning-based models for combinatorial optimization problems is dependent on the availability of a large set of optimal or high-quality solutions. Thus, two key issues arise when formulating these problems as supervised learning tasks: (1) we are restricted to learning well-studied problems for which optimal solvers or high-quality heuristic algorithms are available; and (2) we can only train on small-scale problem instances as it is intractable to build datasets for large instances of NP-hard problems.

Although reinforcement learning is known to be more computationally expensive than supervised learning, it does not require the generation of pairs of problem instances and solutions. As long as a problem can be formulated via a reward signal for making sequential decisions, a policy can be trained via reinforcement learning. Hence, most recent work on learning-based approaches for TSP have used reinforcement learning (Deudon et al. 2018; Kool et al. 2019) and dismissed supervised learning. Additionally, the poor empirical results of supervised learning-based methods (Vinyals et al. 2015; Nowak et al. 2017) has supported the argument in favour of reinforcement learning.
Criticism of the current paradigm  We agree with the above-mentioned argument. At the same time, it is surprising that our Graph ConvNet model trained with supervised learning outperforms all reinforcement learning-based methods in terms of both solution quality and speed. However, our results should not be used to conclude that supervised learning is the superior approach for training models to solve combinatorial optimization problems.

Instead, we feel that our results highlight an important flaw in the current paradigm of learning-based approaches for TSP. Despite strong performance on problem instances of various fixed sizes, our models show extremely poor generalization capabilities. [Kool et al.] (2019) observe a similar albeit less drastic trend for generalization in other recently proposed deep learning approaches.

Thus, we feel that future work should abandon the notion of training and evaluating models for discrete problem sizes as this is extremely misleading for comparing among approaches. The broader goal for the machine learning community should be to build size-agnostic models with strong generalization capabilities. This would help us realize the initial promise of learning-based approach for combinatorial optimization: training on small problem instances and being able to solve instances of larger, more realistic sizes.
7 CONCLUSIONS

7.1 FINDINGS

In this paper, we introduce a novel deep learning approach to approximately solving the most famous NP-hard problem in recent history, the Travelling Salesman Problem. We focus on the 2D Euclidean TSP and use Graph Convolutional Neural Networks and beam search to predict a valid TSP tour given an input graph with up to 100 nodes.

Our approach outperforms all recently proposed learning-based techniques in terms of both solution quality and speed for fixed problem sizes. We are further able to improve our performance at the cost of evaluation time by using a hybrid approach combining our learned model with local search heuristics.

However, experiments on the generalization capabilities of our model show a drastic drop in performance when evaluated on problem sizes different from those that models were trained on. Despite their state-of-the-art performance on fixed problem sizes, it is clear that our models are overfit to the training graph sizes and are unable to generalize to variable sizes.

The initial promise of learning-based approach for combinatorial optimization was that models could be trained on small problem instances and generalize to larger instances. Our empirical results demonstrate that comparing among learning-based approaches based on performance for discrete problem sizes completely ignores generalization. Future work in this field should focus on building size-agnostic models with strong generalization capabilities in order to scale up to realistic problem sizes.

7.2 FUTURE WORK

Future work will investigate why our approach cannot generalize to variable problem sizes by critically analyzing the model architecture and training setup.

Model Capacity As noted in Zhang et al. (2016), models with extremely large amount of tunable parameters quickly memorize training data, leading to poor generalization. For our approach, model hyperparameters are currently arbitrarily chosen to match the increasing complexity of problem instances. It is important to select hyperparameters in a more principled way in the future. It would also be interesting to find the minimum model size which is able to learn to solve each TSP instance size.

Stress testing Supervised Learning Currently, we generate datasets consisting of 1 Million pairs of problem instances and solutions which are used to train our models in a supervised fashion. It is important to stress test our supervised learning setup in terms of data/time efficiency: (1) how few TSP instances do we need to learn good solvers; and (2) how fast can we learn good solvers.
REFERENCES


A TRAINING LOGS

Figure 12 displays the learning rate and loss values vs. training epochs for various runs. Decaying the learning rate to very small values allows the training loss function to smoothly decrease, even after 1,500 epochs. Loss curves for TSP50 and TSP100 show that models start overfitting to the training set after around 500 epochs. However, as seen in Figure 5, validation optimality gap does not get worse even as models overfit to training data. This indicates that generalization to unseen problem instances of the same size as those seen in training is not a good indicator of model performance.

![Learning Rate vs Training Epochs](image1)

(a) Learning Rate vs Training Epochs

![Loss vs Training Epochs for TSP10](image2)

(b) Loss vs Training Epochs for TSP10

![Loss vs Training Epochs for TSP20](image3)

(c) Loss vs Training Epochs for TSP20

![Loss vs Training Epochs for TSP30](image4)

(d) Loss vs Training Epochs for TSP30

![Loss vs Training Epochs for TSP50](image5)

(e) Loss vs Training Epochs for TSP50

![Loss vs Training Epochs for TSP100](image6)

(f) Loss vs Training Epochs for TSP100

Figure 12: Training logs for various runs.
B ADDITIONAL VISUALIZATIONS

Figures 13, 14, 15, 16 and 17 display additional prediction visualizations for samples from test sets of various problem instances. We have included examples where the models are able to predict shorter tours than the groundtruth solution obtained using Concorde. Although we are cherry-picking the best examples, this is a very puzzling result: it was initially taken for granted that training models using supervised learning would mean that they can never perform better than the solver used to find the groundtruth solutions.

(a) The model strongly predicts the edges which form a cycle enclosing all nodes in the graph. The exact contours are then found using beam search.

(b) The model predicts a shorter tour than Concorde by choosing a different contour around the four points at the top of the graph.

Figure 13: TSP10 model prediction visualization for sample test set instances.
(a) This instance can be considered easy because the optimal tour is simply the cycle enclosing all the nodes. The model strongly predicts most of the groundtruth edges. It is less confident about the longer edges, but more-or-less solves this instance without relying on beam search.

(b) This instance is comparatively harder because it envelopes around the nodes inside the enclosing cycle of the graph. Our model predicts a shorter tour than Concorde by choosing a different contour for the nodes at the bottom-left corner.

(c) Another instance where the model strongly predicts edges on the optimal tour without relying on beam search.

Figure 14: TSP20 model prediction visualization for sample test set instances.
(a) A simple instance where the model predicts edges on the optimal tour without relying on beam search.

(b) The model predicts a shorter tour than Concorde by choosing a different contour around the nodes at the top-right corner.

(c) This instance is similar to (b), but with more complexity in the prediction heat map due to several nodes being in close proximity in the top-left corner. The model relies on beam search to find the optimal tour.

Figure 15: TSP30 model prediction visualization for sample test set instances.
(a) The model predicts a tour that makes different envelopes around the nodes compared to the groundtruth. However, it is very close to optimal in terms of length.

(b) Most of the complexity of this instance comes from the group of nodes at the bottom-right corner. The model predicts a shorter tour than Concorde by choosing a different contour around the nodes at this corner.

(c) This instance is complex due to several nodes being inside the enclosing cycle of the graph. The model relies on beam search to find the optimal tour.

Figure 16: TSP50 model prediction visualization for sample test set instances.
(a) The complexity of this instance is due to the envelope from the bottom-center into the center of the graph in the optimal tour. The model relies on beam search to navigate the complexity and predicts a tour close to optimal.

(b) This instance is harder than (a) because there are several envelopes towards the center of the graph in the optimal tour. The model is able to identify these envelopes but beam search is unable to find the optimal contours around the center due to very high complexity.

(c) This comparatively simpler instance is solved better than Concorde by choosing different contours around the nodes at the top-left corner.

Figure 17: TSP100 model prediction visualization for sample test set instances.