TSP Meta-Learning using Deep Neural Networks

Adam Gastineau
Advisor: Dr. David Rader

Rose-Hulman Institute of Technology
Computer Science and Mathematics Departments

Last Revision May 25, 2018

Contents

Abstract

The Traveling Salesperson Problem (TSP) is a classically difficult combinatorial optimization problem, often taking immense computational time to find an optimal solution. Heuristics and meta-heuristics (heuristics designed for heuristics) are therefore commonly used to find quality solutions (though not provably optimal) in a short period of time. Since different heuristics can produce different solutions and there cannot be a best heuristic for all TSP instances, meta-heuristics are often used to choose the optimal heuristic to run. These meta-heuristics are traditionally engineered by hand and make many assumptions that are not necessarily appropriate for the problem in question. Machine learning largely eliminates this bias, but often introduces its own through the feature engineering process. Deep learning's end-to-end training philosophy removes this dependence on humans, often to great benefit. To this end, this paper explores using deep neural networks to create a learning meta-heuristic to identify which heuristics are best to run on various TSP instances. The DeepWalk algorithm for encoding TSP instances is used to convert instances into a usable format for learning. The final network construction, consisting of several Long-Short Term Memory (LSTM) layers followed by dense layers performing ListNet ranking, was found to perform better than the state of the art classical machine learning system on the same dataset. However, the results also indicate deficiencies in several research areas that must be addressed in order to produce high performance meta-learning systems for the TSP using deep learning.

1. Introduction

1.1. The Traveling Salesman Problem

The Traveling Salesman Problem (abbreviated TSP) is a commonly studied application of combinatorics, graph theory, and algorithm design due to its many uses in real-world scenarios, ranging from package distribution to scheduling. It is traditionally described as an optimization problem whose objective is to minimize the total cost of visiting all nodes in a graph, where cost is defined as a weighted directed edge connecting two nodes. The completed path, visiting all nodes and returning to the starting location, is known as a tour. Due to its many applications from vehicle routing to PCB production [1], solving this problem quickly and efficiently is highly valuable. In addition, it is considered a "traditional" NP-Hard problem, and as such is commonly used as a benchmark for new optimization algorithms and in the research of new optimization methods.

Recent developments in Deep Learning have resulted in increased interest in solving TSPs through learned optimization methods [2] [3]. Though far from producing practical results, several studies have shown there is a potential to learn combinatorial graph interpretation with the appropriate network structure [4] [5], thus motivating further exploration of solutions. These systems apply learning mechanisms to directly produce TSP solutions, attempting to reach optimality. In practice, however, solving to optimality is often not reasonable or even necessary, with results within 5% of the optimal cost typically available in fractions of the expected computation time. To this end, many heuristics have been developed over the course of the past 70 years (particularly with drastically increasing complexity in the past 20 years). As dictated by the No Free Lunch Theorems for optimization [6], each heuristic presents a bias towards a subset of problems for which it performs optimally. Therefore there is no singular heuristic to reasonably use in all cases. However, for practical purposes, examining multiple heuristics is time intensive and potentially infeasible. Recent studies in so called "meta-heuristics", i.e., heuristics designed to select heuristics, attempt to solve this problem by developing a system to predict the most beneficial heuristic to run given certain constraints.

1.2. Meta-Learning

Meta-heuristics run in a short (often linear or quadratic) time to select the most applicable heuristic or exact optimization algorithm. While meta-heuristics are still heuristics, and hence produce non-optimal results, they allow applications to make reasonable assumptions about the instance in question in a short period of time. This permits the development of programs that have some sense of the total running time of the solving algorithm and optimality of the result. In addition, meta-heuristics allow for industries to approach problems not previously possible, due to the time savings of generating “better” solutions.

Modern research in meta-heuristics typically involves significant human involvement, particularly in the development of the heuristic itself. The past several years have produced several studies applying machine learning techniques as a meta-heuristic for combinatorial problems, but this research is still dated by its use of traditional machine learning techniques, particularly feature engineering. In Kanda et al. [7], up to 41 different hand-constructed metrics are passed to a neural network or another machine learning system which produces a ranking of various optimization heuristics. Thus “meta-learning” is defined in a two-fold manner: first as learning which heuristic to choose (or a “learning” meta-heuristic) and second as learning about the TSP through metadata, that is, through hand-constructed features. Since “meta-learning” has no formal definition at the time of writing, this fusion will be the definition used throughout this paper. While this algorithm selection system produces interesting results, the engineered features incur significant additional computation time and evaluate the final TSP heuristics used for classification/ranking on subsets of the particular TSP instance. This computation time should be avoided if possible, but more importantly, running smaller subsets of problems in order to predict which heuristic to use is likely to misrepresent the utility of the meta-heuristic system (in this case machine learning) and as such is an egregious error.

Potential limitations in the research by Kanda et al., in addition to the many new advancements to machine learning due to the prevalence of deep learning, suggest that further experimentation would be prudent. The logical next advancement of this work can be framed as follows: Can deep learning perform better than traditional methods in classification and/or ranking tasks on the TSP? Specifically, can deep learning predict, given a particular TSP instance, the ranking of heuristics by lowest cost, so that the first heuristic in the ranking produces the result closest to optimality?

This thesis is organized as follows: Section 2 contains an overview of the pertinent topics discussed in the paper, including descriptions of various heuristics used and machine learning methodology. Section 3 discusses the design of various components of the thesis such as network structure, dataset creation, and the design of experiments. The actual experiments conducted and their results are shown in Section 4. Finally, the discussion of the entire process is wrapped up in Section 5.

2. Background

2.1. Heuristics

Throughout the years, the TSP has been approached from many different directions, largely due to its utility in a variety of scenarios. Of particular interest to this study is previous work in the area of meta-heuristics. Heuristics are commonly written as simple approximation solvers of a particular subset of TSP instances, with some characteristic shared between the instances. While most heuristics will produce a reasonably accurate result in a short amount of time, their performance is dictated by the type of problem it is applied to. Since they are hand-designed approximation algorithms of NP-hard problems, certain assumptions are made to simplify the solving process. These assumptions cannot hold true for all problems according to the No Free Lunch Theorems by Wolpert and Macready [6], hence introducing a significant margin of error if the problem instance in question does not correspond to the heuristic's expectation. For this reason, the heuristics used in this research were chosen that minimize this selection bias towards TSP features. In particular, the selected heuristics are general optimizers, and are not limited to optimizing combinatorial problems such as the TSP. To mirror the algorithm selection by Kanda et al. [7], this study includes Tabu Search [8], Simulated Annealing [9], Greedy Randomized Adaptive Search Procedure (GRASP) [10], Genetic Algorithms [11], and Ant Colony Optimization [12]. Each of these techniques perform exploration of the problem instance's search space with no inherent knowledge of the actual problem being inspected, often with random components to prevent being caught in a local minimum. This results in algorithms that have no explicitly designed bias for a particular type of TSP instance. However, this does not exclude other inherent biases. The No Free Lunch Theorems state that for every two algorithms for which one performs better on a problem instance, the other must perform better on a separate instance. Therefore, even these generic heuristics will present some sort of bias towards certain problem instances, dictating better performance in some scenarios.

2.1.1. k-Opt and Lin-Kernighan

The k-opt algorithm is very common in TSP optimization, both due to its simplicity and relative efficiency. It commonly appears as either $k = 2$ or $k = 3$, due to the diminishing returns as k grows. The algorithm was first proposed as 2-opt in 1958, and works by simply removing two edges from a symmetric TSP tour, and reconnecting the disjoint paths in the other available way [13]. If this results in a shorter tour, then the change is kept, otherwise it is ignored. The algorithm halts when an iterative cycle completes with no swaps producing a shorter tour. A k-opt algorithm removes k edges from the tour, and reconnects the paths in the other possible ways. This method has the benefit of rather efficiently removing all overlapping edges in a tour, as crossed edges cannot exist in the shortest tour. Care must be taken in implementation, however, as problematic edge cases exist (such that 2-opt will make $\Theta(2^{n/2})$ moves before halting), which are not feasible for large problems [14]. In light of potentially high running times, many different optimizations and limits have been suggested to maintain efficiency. On average, however, 2-opt will run in $O(n^2)$, which is rather insignificant compared to the improvements it can bring [15].

The Lin-Kernighan algorithm is a variation on k-opt that uses a variable k to further optimize a given tour [16]. Though it is significantly more complicated than 2 or 3-opt, it is capable of reliably producing tours within 2% of the Held-Karp lower bound, which is uncommon for the other k-opts [15]. The Held-Karp lower bound is the resulting solution's cost from a TSP relaxation designed by Held and Karp [17]. Since it is a lower bound for the TSP, it is often used to compare TSP solvers and determine the optimality of proposed solutions. Given a particular iteration step with some λ swaps being considered, the algorithm then examines whether some $\lambda + 1$ swaps should be considered. This process continues until $\lambda + 1 > n$, for some stopping condition n, or if the solution is within some optimum bounds. Later optimizations have produced an algorithm that completes in $O(n^{2.2})$, with optimal solutions practically guaranteed in some constant number of trials, making Lin-Kernighan the forefront heuristic for many years [18]. It is important to note that both the k-opt and Lin-Kernighan algorithms are specific to the TSP, and cannot be applied to other problems. Therefore, these algorithms are not actively considered as a part of this research.

2.1.2. Tabu Search

Tabu search is a relatively simple iterative optimization method that moves from the current solution to another in the "neighborhood" of the current solution. It is very similar to common descent methods, but instead of always opting for the most optimal next solution (and possibly getting stuck in a local minimum), it uses a memory to strategically choose a new optimal solution contained in the current solution's neighborhood. This memory prevents (or adds a strong negative weight) to revisiting previously checked solutions, known as marking the solution as "tabu" [19]. Other modifications have been made to this heuristic, mainly by modifying the memory structure, such as by adding long term memory containing solutions from which to restart the search [20], dynamically changing search parameters when solutions all have similar characteristics [21], and recording the frequency of certain attributes in previously found solutions to avoid in the future [22]. Though it depends on which algorithm is used, Tabu Search takes time $\Omega(n^3)$ (commonly $\Theta(n^3)$) on the TSP, while typically performing only marginally better than 2-opt and 3-opt [14].

2.1.3. Simulated Annealing

Annealing is the process of heating a material and allowing it to cool in order to strengthen the material. Simulated annealing takes that concept, and instead applies it to strengthening the solution of an optimization problem. The algorithm first "heats" the problem, then allows it to "cool", until it finally freezes in some final state. During this process, the algorithm performs standard descent, moving towards solutions that minimize the objective function. The temperature allows the algorithm to avoid local minima by introducing random "energy spikes", where the algorithm suddenly transitions to a new section of the search space, with the probability of these spikes increasing with increased temperature [9]. Specifically, this is implemented by selecting at random a solution from a given neighborhood surrounding the current solution. If this solution is an improvement, it becomes the current solution, otherwise, it is evaluated by the Metropolis criterion, which defines acceptance as a probability relating to the cost increase and the current temperature. The higher the temperature, the more likely the solution will be accepted [22] [9]. When run asymptotically, the algorithm is guaranteed to find a global minimum, though this obviously cannot hold in actual practice [23]. In real-world use, the temperature degrades too quickly to allow all energy states to become stable, and the algorithm must run for an infinite time before a guaranteed global minimum is reached. In general, without any added modifications for speed, Simulated Annealing will run for time $\Theta(n^2)$, with a large constant, when examining a typical number (n^2) of TSP neighborhoods [14].

2.1.4. Greedy Randomized Adaptive Search Procedure

Like both Tabu Search and Simulated Annealing, Greedy Randomized Adaptive Search Procedures (GRASP) are iterative processes. Each iteration is split into two phases, the "construction" and local search phases [10]. The construction is itself iterative, where it builds a feasible solution, adding one component per iteration. The next node to add to the feasible solution is suggested using a greedy randomized search, where the component is selected randomly from the best nodes in a greedily ordered list of all other elements [24] [25]. The construction process is adaptive, since selecting a new element will cause the benefits of non-selected elements to be updated; thus selecting a new element influences the selection of future elements. This cost/benefit update holds even across separate construction sequences, so that earlier constructions affect the creation of new solutions. Finally, once the solution (or tour) has been built, a local search of the current solution's neighborhood is conducted, to ensure a local optimum is found [10]. In optimizing this method for application to the TSP for use in this thesis, instead of setting some parameter α for the "greediness" or randomness of the construction, the absolute best next elements are collected, with the final element to add being chosen at random from that list. In addition, this TSP solver applies two local searches instead of one, a restricted 2-opt method followed by a restricted 3-opt method, presenting a significant improvement over other, more standard GRASP TSP implementations [25].

2.1.5. Genetic Algorithms

Different from the previously discussed heuristics, genetic algorithms are designed to emulate the evolution of organisms over time, by passing traits of parents onto their children in order to optimize a combinatorial problem [26]. This search process can be described as "a variant on local search" [14]. The organisms, or "individuals", as they are named, each represent a possible solution in the search space, and are combined to form a “population”, which represents the neighborhood of the search space being examined. For a given iteration, or “generation”, of the algorithm, parents are selected from the population which produce children with some similar characteristics in a process known as “crossover”, which will “mutate” with some predefined low probability. These children are added to the population, while some original inhabitants of the population are removed to keep the population size consistent. The crossover stage is designed to increase the overall quality of the population (by choosing the best features from the parents), while the mutation stage keeps the algorithm from settling into a local minimum, similarly to that of simulated annealing [27]. Various methods are used to determine when the algorithm should stop. This process by itself tends to not produce high quality solutions to the TSP, so algorithms will often integrate some heuristic, such as k-opt or Lin-Kernighan, into the iteration process, so that each individual is optimized in some uniform way [28]. Also suggested by Freisleben et al. is a crossover method that preserves the “distance” (the overlap between tour edges) between parent and child to be the same as between the parents. Additionally, this modification includes a mutation method that mutates that distance to be 4, selected because “it is the smallest possible non-sequential change” (likely because a move producing a distance of 4 cannot be performed by 3-Opt or Lin-Kernighan) [14].

2.1.6. Ant Colony Optimization

Similarly to genetic algorithms, ant colony algorithms rely on methods seen in nature to optimize a given solution. Ants will often find the shortest path between their nest and food, using pheromones they deposit as they walk. These pheromones dissipate over time, so the shortest found path will have the strongest pheromone concentration, resulting in more ants moving that direction [29]. Applying this to optimization (and specifically the TSP), a system of “agents” are created that mimic the ants and each iteratively build a tour using both past experience and greedy selection. As the agents build solutions, they “mark” the edges with pheromones, creating a desirability measure for future agents to explore that path [12]. More specifically, ants are placed in random cities in the tour, and instructed to move to another city. The city chosen by each ant will be determined in a stochastic greedy fashion, by the cost of that edge along with the desirability as determined by the pheromones (i.e. the memory of the heuristic) [22]. As each edge is added to the tour, pheromone is deposited on the edge according to a local pheromone rule. Once all agents have finished their tours, a global pheromone decay rule is run, degrading existing desirability measures on all edges in the graph. The best agent then “deposits” new pheromones proportional to the length of their tour, so that the edges in a shorter tour have a higher desirability [30]. The final solution is exacted after some number of iterations, or some manner of improvement is reached. This system allows for local minima to be avoided by forgetting some information on edges (the decaying desirability), along with some randomization on the edge to choose. The local pheromone rule is a development over ant system optimization that introduces additional decay in order to expand the search pool and prevent multiple identical solutions [29].

2.2. Meta-Learning

As defined previously, meta-learning encompasses both a learning meta-heuristic (a heuristic that learns to choose other heuristics) and learning about the problem domain indirectly through metadata. According to Kanda et al. [7], "Meta-learning consists of generating a meta-model that maps characteristics of problems (i.e., meta-features) to the performance of algorithms," specifically the TSP heuristics under examination. The mapper is typically some form of classification/ranking learning algorithm, such as those created by the machine learning community. This learning algorithm then produces a correspondence between certain TSP features and the heuristics which tends to produce the best results. Traditional meta-learning studies have constructed idealized features for the given problem (such as the number of vertices in a graph, average vertex cost, clustering coefficients calculated in various manners, etc.), not unlike what is done for many other machine learning problems [7]. However, this methodology presents significant issues when the best results are found using so called "complex features" [31]. Complex features are those that are computationally difficult to calculate (such as the clustering coefficients mentioned above), and can drastically increase the running time of a meta-learning algorithm. After all, the algorithm is only practically useful if it presents a significant decrease in computation time while solving to near-optimal values when compared to running multiple heuristics. Each and every complex feature in Kanda's work increases that compute time significantly, with an unknown benefit to the algorithm itself [7] [31].

2.3. Machine Learning

Machine learning is a very general area of Data Science and Computer Science that includes many methods for performing regression, classification, clustering, and other variable relationship models without directly informing the system how to arrive at solutions. The focus of machine learning for many years now has been the artificial neural network, or ANN. ANNs mimic the human brain by creating artificial neurons that respond to stimuli, firing at some determined intensity, given some activation function and threshold (commonly piecewise linear, sigmoid, Gaussian, or more recently the rectified linear unit, ReLU: $f(x) = \max(0, x)$) [32] [33]. The learning algorithm provides a cost (or loss) function to minimize, a structure of neurons represented as matrices, and some metric for determining error or accuracy. Given input "features" and labeled output data, the system iteratively runs this algorithm, learning through dynamic modification of the parameters of activation, known as weights, for each neuron, along with a more general bias, in order to minimize cost. Over many problem instances and sample data-points, the neurons "recognize" patterns that lead to generating optimal solutions, by activating when given particular inputs [34]. This method shows significant promise due to its ability to "analyze non-linear relationships" [35].

There are two network types typically used in both machine and deep learning, known as feed-forward neural networks and recurrent neural networks (RNNs) (though many others exist). Feed-forward networks can either be a single "layer", that is, one set of neurons receive the input data, and directly output their activation to that data, or multi-layered, where there exist multiple sets of neurons in between the input and output. These additional layers are known as "hidden layers", as these layers perform optimizations that are relatively unknown and difficult to determine [36]. Each layer has some notion of connectivity to the previous and next layer, such that a fully connected layer connects each neuron in that layer to each neuron in the previous and subsequent layer, where a partially connected layer only makes some of those connections. Multilayer networks are most commonly trained using back propagation by steepest (or gradient) decent, which takes partial derivatives throughout the network's operations and propagates those results through the network to set the appropriate weights and biases [34]. As suggested by the name, feed-forward networks always pass the data "forward" through the network, so previous layers never receive output from subsequent neurons. By contrast, recurrent neural networks contain at least one feedback loop, where neurons output data to other neurons in either previous layers, or back to themselves [36]. This type of network is most commonly used for processing of sequential data, or for processing methods that require some form of memory, and have become more common with the resurgence of deep learning [33].

2.3.1. Data Processing and Training

Good data science practices require the proper handling of data. In training an ANN, care must be taken to prevent overfitting, which causes overly specific analysis of input data and a lack of generalization, so that the network responds to trends present only in the dataset used to train or its format [37]. This bias is typically prevented by splitting data into training, validation, and test datasets, with ratios commonly around 50%, 25%, and 25%, respectively. Training is performed directly on the training dataset, using fixed size batches of training data. These batches contain some number of problem instances for the network to evaluate in a reasonably short amount of time, so many batches (potentially containing the same or overlapping data) selected at random will provide enough information for the network to produce reasonable results [37]. Validation data is used to estimate the real world error of the network, and allow for the tweaking of parameters to decrease that error as necessary. The validation set must never be used to train the network, as doing so would insert additional overfitting bias [38]. Instead, the validation set is purely used in order to make observations about network performance and tweak hyperparameters as necessary. The test data is not to be touched at any point until the network training and all other research is complete, to ensure the test data is fairly representative of real world usage. The results produced by evaluating the network on the test dataset are the final results produced by the network to be used in analysis.

Network training often follows one of two main schools of thought, that of "supervised" or "unsupervised" learning. In supervised learning, the network is provided with expected output data accompanying the input, such that given some predefined input X, the expected output is predefined to be Y. Through the training process, the weights and biases of the network are tweaked through the optimization algorithm to arrive as close to the desired output as possible [32]. Reinforcement learning takes supervised learning and modifies the output "data" to be a desired cost function, to be minimized or maximized. In this paradigm, network configurations that produce better optimizations are "reinforced" or rewarded [39]. Unsupervised learning does not provide any labeled data, so the training occurs without any sense of an expected answer. Unsupervised learning is mostly used in instances where trends and anomalies need to be identified, but labeling the data (providing the correct answers) is prohibitive due to time, cost, or size of the dataset [32].

2.3.2. Deep Learning

Deep Learning expands on the neural networks of machine learning and makes one simple modification: adding multiple hidden layers. In adding additional layers, computational complexity can increase significantly, though complexity is not necessarily correlated with depth [40]. This change is desirable, nevertheless, as layers add additional optimization capabilities to the network. The complexity increase made multilayer networks infeasible during the late 20th century and early 2000's, and the concept was not significantly researched during that time period. However, in 2012, the use of a deep neural network applied to image processing through the use of GPUs (referred to as AlexNet) caused an exponential increase of interest in multilayer networks [41]. Also present in AlexNet was the concept of end-to-end training, where raw, unmodified data is presented to the input layer of the network and the output layer is compared against some expected output. In this way, a deep neural network (DNN) can learn "from scratch" with very little human intervention [42].

Traditional machine learning involves the process of "feature selection", where a researcher must select subject-appropriate features from the input data to be used and processed by the neural network [34]. These features are often of mathematical significance (such as symmetry in a TSP instance), and therefore are chosen as good ways to differentiate problems to the neural network. However, this process is the point of highest human interaction in a given machine learning problem (besides potential data cleaning or labeling), and therefore makes up a significant time commitment to an experiment or practical use case. In addition, when applied to meta-learning, spending time selecting features is counter-intuitive to the original goal, that is, towards building a generalized algorithm that requires little designer knowledge of the subject area.

Instead, deep learning brought about the idea of end-to-end training. End-to-end training describes a system in which a system's input data is passed unmodified (such as raw audio for speech recognition) into a neural network. While this drastically increases training time, it tends to result in better predictive ability. This end-to-end training concept is incredibly important in most DNN implementations, and specifically so for combinatorial problems, like the TSP. It has made feature engineering, which can be very time consuming, largely irrelevant for large problems, instead allowing the researcher to focus on tweaking the network itself, including layer count, size, and type, along with input formats and training parameters. To this end, most (if not all) of the previous work on solving the TSP using deep learning has used deep learning as the TSP optimizer itself, as compared to meta-learning, where the optimizer proposes another optimizer best suited to the problem at hand. Google has been at the forefront of TSP solving using deep learning, likely due both to their investment in solving the problem efficiently and their massive computing abilities [2] [3] [43]. Their latest research proposed a new method for tackling combinatorial problems with neural networks, combining the variable input capabilities of sequence to sequence networks with reinforcement learning and multiple search strategies [2]. Other similar studies have examined the use of recurrent neural networks and memory to represent the TSP as it is processed by the network [3] [43].

2.3.3. Label Ranking

Information retrieval engines require some system to order the results that will be presented to the user. In machine learning, this problem was traditionally approached as a classification problem: given 5 items to rank, there are $5! = 120$ unique rankings or categories, which can be selected using traditional classification. While this system produces useful results, the number of possible unique rankings makes it infeasible for most n. However, alternatives are hard to come by. The loss function (the "objective function" for neural network optimizers) must have a known derivative in order to be used for backpropagation, the primary method of setting weights and "learning" using neural networks. Since ranking or applying order to values requires a sort, producing a less contrived methodology was difficult due to the inherent lack of a derivative for sorting. Pairwise comparisons were attempted, but proved to be cumbersome [44].

This problem was addressed by a Microsoft Research team, coining the term ListNet for a loss function that compared entire lists of rankings, instead of individual pairs [45]. ListNet loss is based upon the softmax function and cross-entropy, where the softmax probability is:

$$P(y = j|\boldsymbol{x}) = \frac{e^{\boldsymbol{x}^T \boldsymbol{w}_j}}{\sum_{k=1}^{n} e^{\boldsymbol{x}^T \boldsymbol{w}_k}} \tag{2.1}$$

for n inputs, weights $\boldsymbol{w}$, input $\boldsymbol{x}$, and output class $y \in \{1, 2, \ldots, m\}$, for a total of m classes to rank. The softmax function compresses a K-dimensional vector of arbitrary real values into a K-dimensional vector of real values in $(0, 1)$ such that the values sum to 1. e is raised to the given powers to ensure the numbers are always positive, and e itself is used as it has an easily computable derivative. Given two discrete probability distributions p and q, cross-entropy measures the number of bits required to identify an event taken from a set of events, if q is the observed distribution, and p is the true distribution. In an efficient system, one would use the fewest number of bits to represent common events, and cross-entropy exploits that, so that cross-entropy increases the more predicted values differ from the "true" value. Therefore cross-entropy can be trivially used as a loss function in classification problems, as it will be minimized when the prediction and true classes are most similar. This is represented by:

$$H(p,q) = -\sum_x p(x)\log q(x) \tag{2.2}$$

Similarly, ListNet loss is:

$$P_{\boldsymbol{y}}(\boldsymbol{x}_j^{(i)}) = \frac{e^{f(\boldsymbol{x}_j^{(i)})}}{\sum_{k=1}^{n} e^{f(\boldsymbol{x}_k^{(i)})}} \tag{2.3}$$

$$\text{Loss} = -\sum_{j=1}^{n} P_{\boldsymbol{y}}(\boldsymbol{x}_j^{(i)}) \log P_{\boldsymbol{z}}(\boldsymbol{x}_j^{(i)}) \tag{2.4}$$

for a ranking function f, and rankings $\boldsymbol{y}$ and $\boldsymbol{z}$ [45]. Basing this method off of cross-entropy allows for several nice properties: a simple to compute derivative, many common and fast implementations, and low time complexity. More recent work has presented other methods for document ranking, particularly ListMLE [46], p-ListMLE [47], and new deep architectures such as DeepRank [48], with generally improving results.

Once a ranking network is trained using one of the aforementioned methods, some notion of accuracy or error needs to be extracted. The most common ranking accuracy metric is Normalized Discounted Cumulative Gain, or NDCG [44]. NDCG calculates the Discounted Cumulative Gain, which penalizes highly relevant classes appearing far from the beginning of a ranking (due to the log reducing the value proportional to its position), determined by:

$$\text{DCG} = \sum_{i=1}^{n} \frac{2^{\mathbf{rank}_i} - 1}{\log_2(i+1)} \tag{2.5}$$

where **rank** is the predicted rank and n is the number of classes in the ranking. NDCG then divides the predicted DCG by the expected (or label) DCG, calculated such that **rank** is the labeled or "true" ranking in the dataset. This normalization ensures values are between 0 and 1 and can serve as a valid accuracy metric. Spearman's Correlation Coefficient is instead used by Kanda's work, which is essentially the "equivalent of normalized mean square error", a common accuracy metric for traditional classification [49]:

$$\text{Corr} = 1 - \frac{6\sum_{i=1}^{n}(\boldsymbol{r}_i - \boldsymbol{l}_i)^2}{n^3 - n} \tag{2.6}$$

for labeled ranking $\boldsymbol{l}$ and network output/prediction $\boldsymbol{r}$. This produces values between -1 and 1, with -1 being inverted correlation, and 1 being full agreement [7]. Other interesting metrics are the accordance of the first values in each of the rankings, along with the first two values in each ranking, which apply more typical classification-style metrics to ranking:

$$\text{top1} = \begin{cases} 1, & \boldsymbol{l}_1 = \boldsymbol{r}_1 \\ 0, & \boldsymbol{l}_1 \neq \boldsymbol{r}_1 \end{cases} \tag{2.7}$$

$$\text{top2} = \begin{cases} 1, & (\boldsymbol{l}_1 = \boldsymbol{r}_1) \vee (\boldsymbol{l}_2 = \boldsymbol{r}_2) \vee (\boldsymbol{l}_2 = \boldsymbol{r}_1) \vee (\boldsymbol{l}_1 = \boldsymbol{r}_2) \\ 0, & \text{Otherwise} \end{cases} \tag{2.8}$$

We developed these metrics to supplement NDCG and Spearman's, which were found to have poor predictive quality (see Section 4.5: Discussion). Due to their relationship to better understood metrics in classification (such as a simple equivalence comparison between two classes), both top1 and top2 were found to be more useful in determining accuracy than the standard metrics (Section 4).

2.4. Graph Encoding

The advent of modern Deep Learning and big data graphs (e.g. social networks) caused a surge in activity attempting to apply neural networks to arbitrary graphs. This research was intended to produce a system that could accurately cluster or classify different sections of the graph, providing the valuable friend or product predictions that make social networks profitable. In actuality, learning from a graph presents several issues. For this discussion, the most important of these is how to get the graph data into the neural network, a method known as graph encoding. Graph encoding is particularly difficult with sparse graphs, which are the essence of social networks. Fortunately, TSP graphs tend to be less sparse. Another issue is handling the dimensionality of the graph. Since graphs are inherently abstract, choosing the correct dimensionality to properly represent as much of the graph as is necessary to sufficiently learn and generalize is no simple task, and full of various trade-offs [50]. Research has taken two disparate directions on how to encode graphs: to consider the graph similar to natural language processing (NLP) problems, or to consider the graph as an "image" over which to run a convolution kernel. Due to familiarity with the problem and the well-tested results of certain algorithms, the NLP evaluation is considered here.

DeepWalk is a recently developed method of encoding graphs for use in deep learning analysis tasks. Designed to handle graphs of arbitrary size, for each node, DeepWalk randomly chooses a new node to travel to, repeating a predefined number of times for each node [51]. During these "walks", various statistics are gathered and fed into a recurrent neural network while essentially considering the walk as a "sentence" in a machine-modeled graph language, outputting a sequence of vectors related to these walks [52]. Each vector is of some predefined size and there is one such vector generated for each node. This process is very similar to that of word2vec, which converts human language sentences into dense vectors for training for NLP tasks, and some word2vec code is present in the actual implementation. While DeepWalk presents a "black box" function on the input data, therefore making interpretation difficult, it tends to work very well in practice on graph problems, outperforming many other common encoding methods [51].

3. Approach

3.1. Kanda Reproduction

To make accurate comparisons against past work (mainly by Kanda et al.), the state of the art research needed to be reproduced and verified to the extent possible. Unfortunately, Kanda et al. never released any code for the generation of the 41 features or the neural network itself, so it needed to be developed from the ground up. To this end, careful analysis of the formulas produced in Kanda et al. [7] was performed to determine the feasibility of implementing the aforementioned features, along with attempting to establish the reasoning behind it's inclusion. In the end, it was found that several features were infeasible, unclear, or likely to hinder the results of the research. The omitted or modified features are listed below:

- Average Geodesic Distance (AGD) - Geodesic distance is typically defined as the shortest path connecting two nodes in a graph [53]. It is unclear whether this feature was intended to be the shortest path, as it should be by name, or the average path length, as it is by formula. As such, both features (average and shortest) were included in the reproduction.

- Alternative Clustering Coefficient (ACC) - In the formula presented, both $N_{\Delta}(i)$ and $N_3(i)$ are not well defined. A best guess was chosen, but it may not be the same as Kanda's intention.

- Network Cyclic Coefficient (NCC) - Detecting cycles involving each vertex is non-trivial and computationally expensive. While the implementation should be correct, performance is less than ideal and larger instances may omit this feature.

- Entropy of the Degree (EDD) - The probability of the vertex degree m_i is unclear, and as such this feature is omitted.

- Vertex Participation Coefficient (PCV) - Determining "communities" in graphs is arbitrary and a difficult problem, and as such the implementation is unclear. This feature is omitted.

- Subsampling Landmarkers Properties (SLP) - All SLPs were omitted from this reproduction as including them significantly biases the study. The desired outcome of Kanda's work was to evaluate what features proved most useful in learning how to predict the best heuristic to use on a TSP instance. By including results from the subset heuristics, it is not unlikely that the network simply learned to feedthrough those results as the final answer. It is possible that the network would not learn with the inclusion of these features, and therefore they were not included.

The experimental procedure also differed from Kanda's; Kanda et al. opted to select a few large TSP instances from the popular TSPLib library and split those few instances into several thousand subset examples. Again, deciding this was poor experimental procedure, all experiments for this work have been performed on a unique subset of the TSPLib library and several thousand randomly generated instances (see Section 3.2: Dataset Generation. While no guarantee can be made that these randomly generated instances are representative of “hard” or useful problems, it is expected that a large sample size will alleviate any potential issues. To ensure comparability of results, the same five heuristics that were selected by Kanda are used in both the Kanda reproduction and the new research. Those heuristics are Simulated Annealing, Tabu Search, GRASP, Genetic Algorithms, and Ant Colony Optimization. Unfortunately, these five heuristics were observed to stray significantly from the expected, relatively equal distribution of results, and as such, were likely not the best choice (see Section 4.1: Dataset).

3.2. Dataset Generation

Deep Learning requires a very large number of examples in order to properly optimize its objective. TSPLib, the main source of TSP instances for research purposes, is unfortunately rather small and lacking in varied data. In the end, the smallest 83 instances from TSPLib were selected, each with less than 400 nodes. In order to supplement these instances, random graph generation was used, bringing the total size of the dataset up to ~ 3000 instances. Similar to TSPLib, generated instances were either symmetric or asymmetric (having unequal weights between two nodes), and had some constant probability of two nodes being connected.

The generation procedure was as follows, given number of nodes n and connection probability p:

- If generating a symmetric instance, for all nodes in the graph, generate x, y, and z coordinates from the uniform distribution $[0, 1000)$.
- For every node i, set the cost matrix to be 0 and the adjacency matrix to be disconnected on the diagonal.
- For every node i, for every node j ($i < j$ for symmetric graphs), generate a random number $a \in [0, 1)$. If $a > p$, mark the vertices as disconnected from i to j and j to i, and set the corresponding costs to 0. If $a \leq p$, mark the vertices as connected and set the cost.
- If this graph is symmetric, calculate the 3D Euclidean Distance between the x, y, z coordinates of i and j for the cost. Otherwise, choose a cost from the uniform distribution $[0, 1000)$.

This process was repeated thousands of times for $n = 10 \ldots 200$. The final dataset was built from a total of 50 symmetric and 50 asymmetric instances of each size. This resulted in $19,100$ unique generated instances, in addition to TSPLib, to be considered for training.

Due to complications in generation, all generated instances in the final dataset are fully-connected, that is, every node is connected to every other node. It was quickly discovered that truly random connectedness can produce "unsolvable" instances; instances where there is no possible complete path through the graph. Similarly, random connectedness can produce graphs that have very few potential complete paths, often less than 100 total solutions connecting all of the 200 nodes. This makes it very difficult for heuristics that "randomly" navigate the search space to find a solution. In addition, these difficulties are very difficult to control for. Therefore all partially-connected generated graphs were removed from the dataset (or in the case of the final dataset, not generated). This potentially has a large negative impact on the generalizing performance of any network applied to this dataset, but seems to be unavoidable.

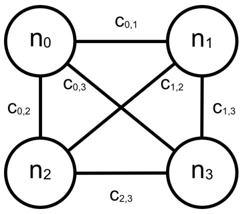
3.3. Graph Encoding

As discussed in Section 2.3.2: Deep Learning, modern Deep Learning practices dictate that end-to-end training be used to guarantee the best results. While this often results in significantly longer training times, the accuracy increase makes this a desirable trade-off. In addition, the main goal of this research was to minimize the influence researchers have on the results produced by the network, which requires the removal of the explicit features provided by Kanda et al., replacing them with some other data injection method, specifically an end-to-end solution [7]. It is then apparent that some system for translating TSP graph instances into a form understandable by neural network architectures is required in order to train and predict based off of these graphs.

The naive solution for graph encoding is simply to pass the cost matrix that represents a given TSP instance into the network. Neural networks consist of a series of matrix operations, and as such, all inputs must ultimately be converted into a matrix or vector. This makes the cost matrix form of a TSP graph a natural input mechanism and prevents any translation into a temporary, potentially lossy form.

$$\begin{bmatrix} 0 & c_{0,1} & \cdots & c_{0,n} \\ c_{1,0} & 0 & \cdots & c_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,0} & c_{n,1} & \cdots & 0 \end{bmatrix}$$

(a) A cost matrix for a TSP instance of n nodes



(b) A symmetric weighted graph

Figure 3.1.: The representation of a TSP instance as a cost matrix [1]

In theory, since this matrix encompasses the entirety of the TSP graph, if a learning system is unable to properly fit using this encoding, no encoding will be adequate.

[1]It is rare for a TSP instance to contain loops in the Graph Theory sense. All graphs considered in this research contain no loops.

Similarly, if a learning system is able to obtain a reasonable accuracy, it is possible that simplifications or lower dimensional solutions will perform likewise.

To test this, the 83 instances of the TSPLib dataset were each arranged into a square, zero-padded matrix of size 200×200. The instances themselves were kept square, so if an instance was of size 10, row 1 of the final matrix would contain ten values in the first ten columns, and zeros elsewhere. This was to preserve relationships between nodes, as flattening the cost matrix could remove those relationships. The costs were normalized globally and passed into a neural network consisting of two hidden layers of 256 and 128 nodes, respectively, and running classification with the 5 heuristic classes. This network was trained using the ADAM optimizer using cross entropy as its loss function and accuracy by the standard metric. Unexpectedly, this system experienced rapid overfitting, with the training set reaching an accuracy of 98% within 21 epochs, while the validation set presented an accuracy of only 38%. Unfortunately, it is not obvious why the overfitting occurred or how to correct it. The lack of any mention of such a training method in the literature suggests that this setup is known to be flawed. However, that reasoning is not known to us, and as such, does not appear in this thesis.

The failures of the cost matrix thus motivates the exploration of graph encoding methods. Section 2.4: Graph Encoding mentions the two-fold nature of graph encoding at the time of writing: handling the graph in the same manner as NLP tasks, or treating the graphs similar to images for convolutional learning. Since convolutional graph learning tasks are a relatively new innovation with little backing research, the prudent choice for an encoding method was one following the NLP methodology. The DeepWalk algorithm often performs the best out of similar methods on supervised and unsupervised learning tasks on large social network graphs, which suggests similar performance on the smaller and slightly different task of learning about entire graphs [54] [52]. The inherent randomness of the algorithm leaves much to be desired, however. Contrary to the typical applications of DeepWalk, the defining characteristics of TSP graphs is the weighting of edges, vs. simple connectivity. It is prudent to consider some method of directing the random search based upon the cost matrix, but the procedure with which to do so is ambiguous. A logical system should grant a high probability of travel to edges with either high or low weights, with an appropriately low probability to the inverse. This results in either a high probability to take walks of a low cost, or of a high cost. The former is expected to produce better results, but that remains to be seen with further testing.

$$s = \sum_{j \in N_i} w_{ij}$$

$$p_{ij} = \begin{cases} \frac{w_{ij}}{s} \cdot a_{ij} & i \neq j \\ 0 & i = j \end{cases} \tag{3.1}$$

OR

$$p_{ij} = \begin{cases} \frac{s - w_{ij}}{n \cdot s} \cdot a_{ij} & i \neq j \\ 0 & i = j \end{cases} \tag{3.2}$$

Probability of traveling along edge ij given the edge weight w_{ij} and adjacency binary value a_{ij}, given the set of neighbors of i, N_i, and $n = |N_i|$.

While both equations were tested, equation (3.2) was the one used for the final DeepWalk implementation and throughout the experimentation. (3.1) was abandoned as it gives a smaller probability to edges with lower costs, resulting in the algorithm building statistical models about the undesirable, higher weighted edges in the final tour.

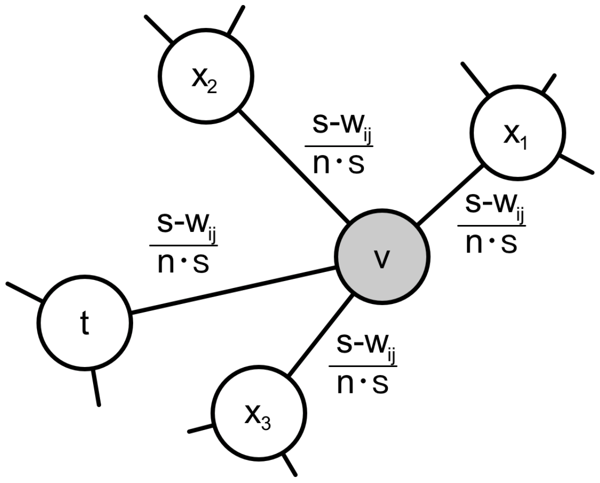

x_2
$\frac{s-w_{ij}}{n \cdot s}$
x_1
$\frac{s-w_{ij}}{n \cdot s}$
$\frac{s-w_{ij}}{n \cdot s}$
v
t
$\frac{s-w_{ij}}{n \cdot s}$
x_3

Figure 3.2.: Custom DeepWalk implementation (w_i is the cost of edge i and n is the number of neighbors of node v).

Since DeepWalk modifies the dimensionality of the graph in question, care must be taken to choose useful dimensions, as making the output too small or too large will hinder learning ability, due to data loss or to massively increased training times. This custom DeepWalk implementation was tested with the default number of walks from each node (10) and an output vector of length 64, 128, or 300 from each node. It was observed that the network performance appears to be highly coupled to the output size, with DeepWalk size 300 producing the best results.

3.4. Network Structure

All neural networks used in the development of this thesis were built using Google's Python machine learning library known as TensorFlow. TensorFlow was chosen due to our familiarity with the package and the relative availability of documentation, code samples, and other methods of support. In addition, the Anaconda Python distribution, which includes the libraries NumPy, Pandas, and Scikit-learn among others, was used for various supporting tasks, including data cleaning and neural network training.

DeepWalk is based upon and includes machine learning NLP techniques, and as such, the output from DeepWalk is traditionally handled in a similar manner to handling

NLP data. Specifically, DeepWalk outputs a sequence of vectors of length k, returning n vectors in total, where n is the number of nodes and k is defined to be the DeepWalk output length for each node (see Section 3.3: Graph Encoding). This output is not unlike traditional human language (e.g. English) sentence encoding methods for NLP tasks, which use recurrent neural networks almost exclusively due to their relationship learning properties. While the interdependence learning properties of recurrent networks are beneficial, recurrent neural networks are additionally useful for handling DeepWalk output's variable size. Given a graph of n nodes, DeepWalk will output a sequence of length $k \cdot n$. This presents challenges for feedforward neural networks, which require a constant, predefined input size. It is possible to zero-pad the input data to hold the input size constant, but that demonstrates significant bias towards the first $k \cdot m$ nodes, where all graphs are of size $n \geq m$. Randomization of the subvectors in the sequence would largely eliminate this issue, but the network is still likely to only consider the first m subvectors, as they are the ones that always appear.

A variant of recurrent neural networks, the Long Short-Term Memory cell (LSTM) [55], largely solves both of these problems. Notably, LSTMs are able to receive dynamic length sequences natively, disabling further processing once the sequence is complete (as compared to more traditional methods, which continue processing the zeros). Due to dynamic sequence handling, LSTMs are commonly used in NLP tasks, as sentences often vary in length. Notice that LSTMs share a relationship with NLP similar to DeepWalk, which borrows NLP techniques to apply them onto graphs. This grants a natural neural network method for handling the input from DeepWalk. Additionally, LSTMs are a form of recurrent neural networks that are able to learn interdependence on data, namely, between separate sections of a sequence (referred to as k-length subvectors above). While the exact sequence relations to be found via the graph encoding method are not obvious, it is advantageous to be able to learn at least some of the inherent features of the graph. As graphs, particularly weighted graphs, are essentially interconnected relationships, adding additional capability for learning connectivity should improve results. A variation of the LSTM, the Convolutional LSTM (ConvLSTM), was also examined, though it was not present for the majority of the research.

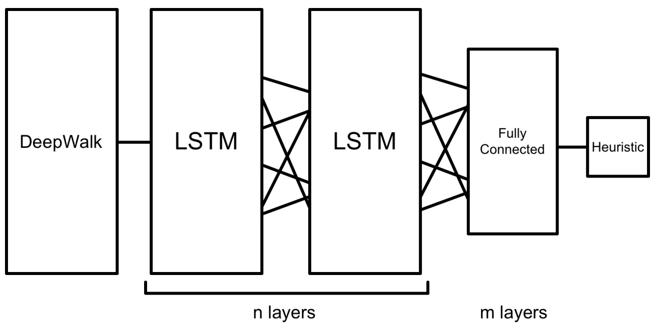


Figure 3.3.: Network implementation, tested with n LSTM layers of i cells each followed by m fully connected layers of varying sizes.

To aid in training, particularly in a situation where training data is not prevalent, dropout is implemented between each of the LSTM layers. Dropout is the recently developed practice of randomly "dropping", or removing, portions of weights from the network in order to prevent overfitting [56]. It is particularly useful in situations involving LSTM layers, as the memory aspect of LSTMs make them hard to train (the network can "remember" something that isn't useful). Due to implementation and performance difficulties in TensorFlow, dropout was only implemented in-between each pair of LSTM layers, instead of also being applied directly to each LSTM cell.

On the output side of the network is the ListNet ranking system [45], characterized by several fully-connected layers. Relatively little prior research exists to indicate the best hyperparameters for ranking, and even less exists for ranking applications similar to this one. As such, the experiments were all conducted with a relatively high number of fully-connected layers (seven in total) to prevent a lack of network learning capacity, i.e. underfitting. Since ranking is similar to classification (classification is a generalization of ranking), the standard network structure (several fully-connected layers) for classification is the logical system to use for ranking as well. Accuracy was measured using NDCG, Spearman's coefficient, top1, and top2 ranking metrics (see Section 2.3.3: Label Ranking). As described in Section 4: Experiments and Results, these metrics were not designed for this sort of ranking problem, and as such present accuracy and training issues.

4. Experiments and Results

All CPU-based calculations (data cleaning, Kanda feature generation, dataset generation, etc.) were performed on a machine with four Intel Xeon E5-2698 CPUs. All training was performed on a system with eight NVIDIA GeForce GTX 1080s, though all tests were conducted on only one or two GPUs.

4.1. Dataset

All experiments were run on the same dataset, generated as described in Section 3.2: Dataset Generation. All five heuristics, written in Python, were run on every TSP instance in the set (83 instances from TSPLib, $19,100$ generated), and the running times, tour costs, and final tour were recorded. For each instance, the heuristic costs were sorted in ascending order to form a ranking. Since 99% of the instances were generated at random, it was extremely unlikely that heuristic performance would follow a uniform distribution (e.g. 20% of the time, heuristic A performed best, 20% for heuristic B, etc.). As expected, performance ended up being far from uniform with certain heuristics performing significantly better than others on average (particularly GRASP and Ant Colony Optimization). This effect was compounded when exploring the resulting rankings. Unexpectedly, not only were the rankings poorly distributed, but several ranking permutations occurred quite frequently. Figure 4.1 shows the distribution of the first 20 rankings.

Ranking	# of instances
[4 1 3 2 0]	6771
[2 0 4 1 3]	6257
[0 2 4 1 3]	1178
[4 1 3 0 2]	1082
[4 3 1 2 0]	1070
[2 4 0 1 3]	495
[2 0 4 3 1]	371
[2 0 1 4 3]	361
[2 4 0 3 1]	215
[4 1 2 3 0]	171
[4 3 2 1 0]	135
[2 4 1 3 0]	130
[0 2 1 4 3]	113
[4 2 3 1 0]	86
[2 4 1 0 3]	81
[2 4 3 1 0]	76
[4 2 1 3 0]	52
[4 2 1 0 3]	52
[2 4 3 0 1]	50
[4 2 0 1 3]	43

(a) Original 19, 100 instances

Ranking	# of instances
[4 1 3 2 0]	200
[2 0 4 1 3]	200
[0 2 4 1 3]	200
[4 1 3 0 2]	200
[4 3 1 2 0]	200
[2 4 0 1 3]	200
[2 0 4 3 1]	200
[2 0 1 4 3]	200
[2 4 0 3 1]	200
[4 1 2 3 0]	171
[4 3 2 1 0]	135
[2 4 1 3 0]	130
[0 2 1 4 3]	113
[4 2 3 1 0]	86
[2 4 1 0 3]	81
[2 4 3 1 0]	76
[4 2 1 3 0]	52
[4 2 1 0 3]	52
[2 4 3 0 1]	50
[4 2 0 1 3]	43

(b) Final dataset

Figure 4.1.: First 20 ranking frequency counts [1]

It should be noted that both heuristic 2 and 4 (GRASP and Ant Colony Optimization, respectively) each appear as the optimal heuristic in 9 of the top 20 most frequent rankings. Similarly, one of the two are optimal in over 86% of the dataset. This is a sign of a poorly distributed dataset that will result in significant bias towards those categories. Due to the randomness inherent in generation and implementation differences in the heuristics, nothing can be done to solve this issue short of generating several orders of magnitude more instances. In an attempt to minimize bias, the dataset was trimmed, taking only a maximum of 200 instances of each ranking permutation. This reduced the dataset from 19,183 instances to 2,818 and flattened the distribution somewhat, with GRASP and Ant Colony Optimization being best in roughly 77% of the instances. While this distribution is not optimal, producing a more evenly distributed dataset would be prohibitively difficult and take more time than is alloted to this research. The high frequency of certain heuristics in the dataset should be considered when viewing all

[1]Ranking values are zero-indexed and reference Tabu Search, Simulated Annealing, GRASP, Genetic Algorithms, and Ant Colony Optimization in order.

experimental results.

4.2. Kanda Reproduction

As discussed in Section 3: Approach, the latest work by Kanda et al. was reproduced for comparison purposes. Of particular interest is any difference in results caused by the changes made in this reproduction (mainly removing certain features, including the problematic subset heuristics). In addition, the Kanda et al. reproduction sets the standard of performance for all other experiments; if one of the deep learning tests is able to surpass the results shown in the reproduction, the hypothesis of this research has been satisfied. To that end, the Kanda reproduction was run on the same final dataset as the other experiments to ensure comparability of results. As Kanda et al. focused on the time savings of this method, a framework was developed to separately time all of the operations required by each feature. Ideally, each feature (including the five heuristics) would be run multiple times, and the timing and performance data averaged, to use as the final input in training. All data presented here is the result of running every heuristic and feature a total of five times on every TSP instance.

Both classification and ranking networks were given the 39 features copied from Kanda et al. 2016 [7] as input, with labels produced by either the lowest cost heuristic (for use in classification) or the ranking of heuristics in ascending cost order (for use in ranking). Both networks use standard architectures of seven hidden layers of 512, 512, 256, 256, 128, 128, and 64 nodes, respectively, trained using the ADAM optimizer. For the classification network, labels were one-hot-encoded, Mean Squared Error was used to display accuracy, and softmax + cross-entropy was used as the loss function. For the ranking network, the ListNet loss function was used instead, along with NDCG, Spearman's Coefficient, top1, and top2 accuracy metrics. The classification results are not shown here, as they presented no interesting results, with accuracy hovering around 40%.

Training	Loss	Spearman	NDCG	Top1	Top2
Full	0.312	0.385	0.820	0.114	0.687
Symmetric	0.310	0.504	0.871	0.762	0.988

Validation	Loss	Spearman	NDCG	Top1	Top2
Full	0.312	0.381	0.818	0.133	0.664
Symmetric	0.311	0.493	0.870	0.726	0.991

Figure 4.2.: Kanda Reproduction Results

Figure 4.2 shows the results of running the Kanda ranking reproduction network on the same dataset as in later experiments, with one test on the symmetric only subset of the dataset (referenced as "Symmetric"), and one on the full dataset (referenced as "Full"). Both tests had a batch size of 2, and ran for 40,000 steps, The division of training vs validation sections of the dataset was held constant for the other experiments, so differing results between tests show differing prediction ability on the same data. Batch size represents the number of instances from the dataset that are trained at once, meaning the weights of the network are only updated after all of the instances in the batch have been processed. The number of steps is the number of iterations the network trained for, where each iteration is processing one batch. Of particular note is the relatively high top1 and top2 accuracies for Symmetric. After closer examination, it was found that the network was only predicting one ranking for the entirety of the validation dataset, [2 1 3 4 0]. This is in part due to the method of creating the validation set, choosing 25% of the dataset at random. This resulted in a less uniform validation set, compared to the entire dataset, particularly when limited to only symmetric instances. However, this same problem appears in Full, where the network only predicts the ranking [0 2 1 3 4] for the entire validation dataset. No manner of parameter tweaking could prevent this fit, which suggests that the dataset is inadequate, and indicates a potential inability of this network to generalize given the selected features. The dataset issues identified here should indicate difficulties in future experiments on the Deep LSTM Network. However, those difficulties were less severe than anticipated.

4.3. Deep LSTM Network

The remaining experiments take place with the deep LSTM network described in Section 3.4: Network Structure. As in the Kanda reproduction, seven hidden layers were used following the LSTM layers, with the same sizes as in the reproduction. All were run as pure ListNet ranking problems, though ListMLE was tested several times. ListMLE ultimately is not included in this discussion as it performed extremely poorly in tests, with long training times and very low accuracy. Accuracy was measured simultaneously with the NDCG, Spearman's Coefficient, top1, and top2 accuracy metrics. The LSTM layers were implemented using TensorFlow's "LSTMBlockFusedCell", which offers better training performance than the default LSTMCell implementation, particularly when training on GPUs. DeepWalk values were passed directly as produced from the algorithm: no scaling was applied.

	LSTM Layers	DeepWalk Size	Symmetric	Batch Size	Steps
Full 1	1	64	False	5	38000
Full 2	2	64	False	5	24765
Full 3	2	128	False	5	30000
Full 4	3	128	False	5	30000
Symmetric 1	2	128	True	5	60660
Symmetric 2	4	128	True	5	60660
Symmetric 3	4	300	True	5	36396
Symmetric 4	5	128	True	5	97057

Figure 4.3.: Deep LSTM Experiment Parameters

The tests were conducted making small parameter tweaks for each new test. The parameters considered for tweaking were: the number of LSTM layers, DeepWalk's single node vector output size (which defaults to 64), and instance symmetry. The parameters of the LSTM layers were tweaked according to the results found by Reimers and Gurevych [57]. In particular, the number of LSTM layers varied based on the test, with as few as 1 and as many as 5 layers considered, all with 150 cells per layer. DeepWalk was run on the entire dataset outputting a sequence of vectors each of length 64, 128, or 300. Tests run on the symmetric only subset of the dataset are denoted "Symmetric i" and tests run on the full dataset are denoted "Full j".

Training	Loss	Spearman	NDCG	Top1	Top2
Full 1	0.204	0.337	0.808	0.185	0.475
Full 2	0.219	0.316	0.801	0.185	0.441
Full 3	0.203	0.340	0.809	0.187	0.478
Full 4	0.217	0.327	0.806	0.180	0.464
Symmetric 1	0.200	0.437	0.841	0.305	0.705
Symmetric 2	0.203	0.432	0.840	0.305	0.701
Symmetric 3	0.207	0.448	0.847	0.305	0.706
Symmetric 4	0.308	0.490	0.871	0.811	0.999

Validation	Loss	Spearman	NDCG	Top1	Top2
Full 1	0.412	0.437	0.842	0.153	0.647
Full 2	0.378	0.387	0.830	0.139	0.496
Full 3	0.405	0.429	0.842	0.135	0.645
Full 4	0.382	0.433	0.843	0.136	0.617
Symmetric 1	0.429	0.557	0.884	0.155	0.696
Symmetric 2	0.442	0.549	0.882	0.167	0.708
Symmetric 3	0.476	0.612	0.897	0.216	0.675
Symmetric 4	0.320	0.535	0.872	0.578	0.958

Figure 4.4.: Deep LSTM Experiment Results

Note that the steps values listed here were not the number of steps required for convergence, but instead the number of steps run overall. When running the tests, it was unclear whether further steps would allow for a better fit, so the optimizer was given plenty of time to run. Examining the loss and accuracy from lesser step counts almost always resulted in similar values, however, so it appears training for an extended period was neither beneficial nor harmful.

It is difficult to draw conclusions from the data resulting from these experiments, as they all produced rather similar numbers. It should be noted that switching to the symmetric only subset of the dataset drastically decreased the time to convergence, particularly for tests with a large number of LSTM layers. This likely occurs due to the smaller symmetric only dataset and the increased computational complexity resulting from more LSTM layers. In particular, Symmetric 4, which was performed using 5 LSTM layers converged to a network which only ever predicted one vector, [2 1 3 4 0].

It is possible that given more time, this network would have produced better results, but time pressure prevented further exploration in this area. In addition, all of the symmetric only networks show increased predictive accuracy (mainly visible in top1 and top2), which is likely a function of a smaller dataset and less variation to predict.

Examining all tests suggests that there may be a strong correlation between DeepWalk size and network performance, particularly indicated by the performance of Symmetric 3, a network with a DeepWalk output size of 300. As described in Section 3.3: Graph Encoding, DeepWalk will output a vector of this size for every node in the graph, so that with n nodes, DeepWalk output size 300 will output a sequence of $300n$. As noted in Section 3.2: Dataset Generation, the dataset is made up primarily of randomly generated instances of size no more than 200 nodes, and of TSPLib instances of size up to 400 nodes. Therefore the majority of the dataset are instances of 200 nodes or less, so when run with DeepWalk output size 300, the amount of information present in the encoding is greater than that of a simple cost matrix (which has at most $200 \cdot 200$ values). This additional capacity could be used to further encode the structure of a TSP instance, which would result in improved predictive performance, as is seen in Symmetric 3. This can also be seen to a lesser extent in the other tests, where moving from 64 to 128 DeepWalk output size in tests Full 2 and 3 result in a slight improvement to all metrics, particularly top1 and top2.

Also of note is the difference between training and validation loss. Traditionally, when there is a large discrepancy between training and validation loss (and in particular, training loss is lower), the network is experiencing overfitting. Indeed, given that training loss is nearly half of validation loss in all experiments, one could argue that is the case here as well. However, the accuracy metrics are very similar between training and validation, and Spearman's coefficient and NDCG are always higher for validation. Though these metrics are admittedly flawed (see Section 4.5: Discussion), these numbers perhaps suggest that the network is not overfitting and that it might even be underfitting. The top1 and top2 metrics designed for this research are also rather similar in all cases except for top1 in the symmetric experiments and top2 often being higher in the validation stage.

4.4. Deep ConvLSTM Network

In addition to standard LSTMs, late in the research a modified LSTM implementation known as Convolutional LSTM (or ConvLSTM) was brought to our attention. This adds a convolutional kernel to the LSTM cells and allows for increased relationship learning abilities, similar to that seen in image processing [58]. Several tests were run replacing the LSTM layers with an external ConvLSTM implementation (the version of TensorFlow in use did not include ConvLSTM). In these tests, the DeepWalk sequence was passed into a ConvLSTM layer of size 5, a 2×2 max pooling layer, a normal 5×5 convolutional layer with 64 filters, a 2×2 max pooling layer, and finally 7 dense hidden layers. Due to the memory requirements of this network, it was run on two NVIDIA GeForce GTX 1080 GPUs.

DeepWalk Size	Symmetric	Batch Size	Steps
300	True	2	8000

Training Loss	NDCG	Top1	Top2	Valid Loss	NDCG	Top1	Top2
0.208	0.863	0.304	0.705	0.413	0.874	0.188	0.717

Figure 4.5.: ConvLSTM Experiment Results

Compared to the standard LSTM networks, the ConvLSTM network converged much more quickly at around 8000 steps with a small batch size of 2. Even though the ConvLSTM structure is more computationally intensive, this resulted in a significant decrease of real-world training times; down to around 10 minutes instead of 5-6 hours. Although the results are inconclusive, it appears that using the ConvLSTM resulted in a marginally worse fit than than the normal LSTM layers, with a top1 validation accuracy of 18.8% in the ConvLSTM test vs a max of 21.6% in Symmetric 3, holding all other parameters constant. Further testing is required to establish whether or not ConvLSTMs are a better choice than normal LSTMs for this problem.

4.5. Discussion

As indicated by the disparate accuracy values, it was quickly discovered that ranking techniques are not completely transferable to ranking heuristics, due to differences in intent. Most ranking systems were designed for document ranking (e.g. by a search engine), which is a relaxation of the ranking problem. The study of document ranking is not focused on developing a strict, deterministic ordering with distinct classes, but instead on providing useful results to a user, who only cares if result A is on the first page, not whether result A appears before result B. While a user would accept a deterministic ordering of results (assuming it was correct), providing that ordering is significantly more difficult than simply providing a rough ranking. This is reflected in the ranking learning mechanisms such as ListNet, and presents problems given our need for a distinct total order.

This is further exasperated by the available ranking accuracy metrics, which display significant deficiencies. For example, consider the prediction vector $[1, 4, 3, 0, 2]$ and label $[3, 2, 4, 1, 0]$, which were chosen to be as visually different as possible. When evaluated by the two most common accuracy metrics, NDCG and Spearman's cofficient, we get:

$$\text{NDCG} = 0.91$$

$$\text{Spearman} = 0.65$$

which are high correlations for two vectors that show no obvious patterns. This suggests that the standard ranking metrics are not suitable for problems such as this one, which require a total order (or a distinct set of unique classes, each appearing exactly once in the ranking). It is possible that these accuracy values are a result of all of the classes $(0-4)$ being present, as will be true for all vectors under consideration in this problem. The lack of an understandable system for measuring accuracy motivated the creation of the top1 and top2 accuracy metrics for this research (both of which return a 0% accuracy for these two vectors), which provide a more human-relatable method for accuracy validation.

Even with these evaluation issues, the dataset is the most likely cause of undesirable fitting. Kanda et al. was able to produce a highly accurate fit [7] [49] using the list of engineered features (even when some tests only used a subset of those features) while the Kanda reproduction provided here could only predict a single vector for any input. Since some tests for the DeepWalk LSTM-based network also only produced a single vector, and the dataset was the only component shared between the latter two tests, the dataset was undoubtedly an issue. As discussed in Section 4.1: Dataset, the dataset was heavily biased towards two heuristics, GRASP and Ant Colony Optimization, with one of the two being optimal roughly 77% of the time. This problem is exacerbated by a small dataset of only 2,818 instances, which is particularly problematic for a deep learning problem. Kanda et al. solved this issue by splitting four large problems from TSPLib into many small, overlapping instances [7], but this was considered to be poor procedure (see Section 3.1: Kanda Reproduction). This could explain the Kanda team's success, in that their instances were not varied enough or all contained overly-similar properties, but there is not sufficient evidence to make this claim. In the very least, our dataset is insufficient for the problem in question, and this issue cannot be remedied easily.

Whatever the cause of the poor fitting performance, the numbers do suggest that using DeepWalk and LSTMs offer an improvement over the work by Kanda et. al, particularly since almost every Deep LSTM experiment has a higher validation top1 accuracy than the full Kanda reproduction test [7]. As top1 is equivalent to simple classification accuracy (comparing the best heuristics in the label and predicted rankings), it is inherently a good measure of performance, particularly when top1 is below roughly 70%. Low top1 performance indicates that the network is often not selecting the most efficient heuristic, which is arguably the most important part of the prediction. By design, the dataset used by the Kanda reproduction experiments was the same used in the Deep LSTM experiments, and the network construction was exactly the same, with the addition of n LSTM layers at the beginning of the Deep LSTM network. The only other difference was batch size, with a batch size of 2 used for the Kanda reproduction vs. 5 used for the Deep LSTM network, though such a small change should not have a large effect on the training. Since the Kanda reproduction network only ever produced a single vector for any input, it has almost no predictive capabilities, regardless of the results given by the accuracy metrics. Therefore, at least on the dataset used, replacing the 39 engineered features of the Kanda reproduction with DeepWalk and several LSTM layers is not only viable, but produces significantly better results. Further research should expand upon this finding, isolating the best techniques to predict heuristics with as little prior information as possible.

5. Conclusion

We intended to build a system that would predict the best optimization heuristic to use given a particular TSP instance using deep learning. This system trains end-to-end after the now common deep learning practice, which requires that the input data be passed "unmodified" into the learning neural network. Based on the requirements set forth by Kanda et al. [7], the output of this network is an ordered ranking of the heuristics from best to worst. The dataset was provided by combining TSPLib and randomly generated fully-connected TSP instances, both symmetric and asymmetric, into a set of over 19,000 instances, evaluated by each of the five heuristics: Tabu Search, Simulated Annealing, GRASP, Genetic Algorithms, and Ant Colony Optimization. The rankings of the five heuristics were calculated and instances selected so that no more than 200 examples of each unique ranking appear in the final dataset.

This fusion of many varied ideas resulted in the exploration of a broad area of research surrounding the TSP and neural networks. Throughout the research process, many deficiencies were found in the currently available research, particularly in the areas of graph encoding and label ranking. Some method for the input of TSP instances into the neural network is necessary in order to train end-to-end, which motivates the survey of graph encoding methods. Graph encoding is traditionally designed to handle very large social network graphs, and as such translates high dimensional problems into a lower dimension more easily processed by neural networks. However, the TSP is not a large social network graph, and the machine learning tasks that typically use graph encoding are very different than determining which heuristic to use. Similarly, label ranking research centers around the concept of document ranking, which does not have the same strict requirements as this ranking problem. We discovered that the standard loss functions and accuracy metrics for learning how to rank make assumptions about the data in question, which actively hinder ranking performance when a total order (one with unique, always-present classes) is required.

With these issues in mind, the DeepWalk algorithm for graph encoding, the ListNet loss function for ranking, and recurrent LSTM layers in the neural network were chosen for this study due to observed performance results or relative ease in use. Experimental results show that this particular combination of these diverse concepts does not perform particularly well, though it is difficult to determine how well the system is functioning due to deficiencies in the ranking metrics. The top1 and top2 accuracy metrics were introduced in an attempt to provide some common and relatable methods of determining accuracy. Using these new metrics and a reproduction of the latest Kanda et al. methodology for comparison, a marginal performance improvement is observed on the same dataset used for our other experiments. This result may be invalidated by the recognized insufficiency of our dataset (particularly as the Kanda reproduction presented almost no predictive ability), but future experimentation is required for conclusive results.

Future work for this study requires a more thorough examination of graph input methods, including but not limited to graph encoding methods, testing of various label ranking methods, and establishing a varied but large TSP dataset, suitable for deep learning. Very recently, including during the development of this thesis, several alternative methods of learning on graphs have been developed, such as GraphGL [59], DeepGraph [60], and Graph2Seq [5], all of which approach deep learning on graphs in different ways. Similarly, there are many graph encoding methods and algorithms similar to DeepWalk that warrant further study in this area. Label ranking presents significant issues due to being designed for partial orders, so it is important to develop new ranking methods using total orders. Additionally, the existing loss functions such as ListMLE [46] and DeepRank [48] should be explored for any benefits they may offer. Finally, it is imperative that a suitable, large dataset for future TSP experiments is developed. TSPLib is insufficient for modern research which often involves learning methods, therefore requiring a large number of example instances. This dataset would allow for more comprehensive and conclusive results for all TSP research.

Bibliography

[1] R. Matai, S. Singh, and M. Lal, "Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches," in *Traveling Salesman Problem, Theory and Applications*, 2010.

[2] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural Combinatorial Optimization with Reinforcement Learning," pp. 1–15, 2016.

[3] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks," pp. 1–9, 2015.

[4] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning Combinatorial Optimization Algorithms over Graphs," pp. 1–28, 2017.

[5] Anonymous, "Graph2Seq: Scalable Learning Dynamics for Graphs," pp. 1–14, 2018.

[6] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[7] J. Kanda, A. de Carvalho, E. Hruschka, C. Soares, and P. Brazdil, "Meta-learning to select the best meta-heuristic for the Traveling Salesman Problem: A comparison of meta-features," *Neurocomputing*, vol. 205, pp. 393–406, sep 2016.

[8] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing*, vol. 8, no. 2, pp. 239–287, 2009.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[10] T. A. Feo and M. G. C. Resende, "Greedy Randomized Adaptive Search Procedures," *Journal of Global Optimization*, vol. 6, no. 2, pp. 109–133, 1995.

[11] L.-N. Xing, Y.-W. Chen, K.-W. Yang, F. Hou, X.-S. Shen, and H.-P. Cai, "A hybrid approach combining an improved genetic algorithm and optimization strategies for the asymmetric traveling salesman problem," *Engineering Applications of Artificial Intelligence*, vol. 21, no. 8, pp. 1370–1380, 2008.

[12] M. Dorigo and L. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, 1997.

[13] G. A. Croes, "A Method for Solving Traveling-Salesman Problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958.

[14] D. S. Johnson and L. a. McGeoch, "The Traveling Salesman Problem: A Case Study in Local Optimization," *Local Search in Combinatorial Optimization*, pp. 215–310, 1997.

[15] C. Nilsson, "Heuristics for the traveling salesman problem," *Linkoping University*, pp. 3–8, 2003.

[16] S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling-Salesman Problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, 1973.

[17] M. Held and R. M. Karp, "A Dynamic Programming Approach to Sequencing Problems," 1962.

[18] K. Helsgaun, "Effective implementation of the Lin-Kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[19] F. Glover, E. Taillard, and D. de Werra, "A user's guide to tabu search," *Annals of Operations Research*, vol. 41, no. 1, pp. 1–28, 1993.

[20] Y. Rochat and É. D. Taillard, "Probabilistic diversification and intensification in local search for vehicle routing," *Journal of Heuristics*, vol. 1, no. 1, pp. 147–167, 1995.

[21] R. Battiti and G. Tecchiolli, "The Reactive Tabu Search," *ORSA Journal of Computing*, vol. 6, no. 2, pp. 126–140, 1994.

[22] M. Gendreau and J. Y. Potvin, "Metaheuristics in combinatorial optimization," *Annals of Operations Research*, vol. 140, no. 1, pp. 189–213, 2005.

[23] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications.* 1987.

[24] Y. Marinakis, "Heuristic and Metaheuristic Algorithms for the Traveling Salesman Problem," in *Encyclopedia of Optimization*, pp. 1498–1506, 2008.

[25] Y. Marinakis, A. Migdalas, and P. M. Pardalos, "Expanding neighborhood GRASP for the traveling salesman problem," *Computational Optimization and Applications*, vol. 32, no. 3, pp. 231–257, 2005.

[26] J. H. Holland, "Genetic algorithms and the optimal allocation of trials," *SIAM Journal on Computing*, vol. 2, no. 2, pp. 88–105, 1973.

[27] P. Larr Naga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators," *Artificial Intelligence Review*, vol. 13, no. Holland 1975, pp. 129–170, 1999.

[28] B. Freisleben and P. Merz, "A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems," in *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 616–621, 1996.

[29] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[30] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 189–213, 2003.

[31] J. Kanda, A. Carvalho, E. Hruschka, and C. Soares, "Selection of algorithms to solve traveling salesman problems using meta-learning," *International Journal of Hybrid Intelligent Systems*, vol. 8, no. Sbrn 2010, pp. 1–13, 2011.

[32] A. K. Jain and J. Mao, "Artificial Neural Network: A Tutorial," *Communications*, vol. 29, pp. 31–44, 1996.

[33] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[34] D. Svozil, V. Kvasnička, and J. Pospíchal, "Introduction to multi-layer feed-forward neural networks," in *Chemometrics and Intelligent Laboratory Systems*, vol. 39, pp. 43–62, 1997.

[35] M. Gevrey, I. Dimopoulos, and S. Lek, "Review and comparison of methods to study the contribution of variables in artificial neural network models," *Ecological Modelling*, vol. 160, no. 3, pp. 249–264, 2003.

[36] S. Haykin, *Neural Networks - A Comprehensive Foundation.* 1999.

[37] K.-L. Du and M. N. S. Swamy, *Neural Networks and Statistical Learning*, vol. 9781447155. 2014.

[38] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, vol. 1. 2009.

[39] A. G. Barto, R. S. Sutton, and P. S. Brouwer, "Associative search network: A reinforcement learning associative memory," *Biological Cybernetics*, vol. 40, no. 3, pp. 201–211, 1979.

[40] J. Schmidhuber, "Deep Learning in Neural Networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2014.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[42] O. E. David, N. S. Netanyahu, and L. Wolf, "DeepChess: End-to-end deep neural network for automatic learning in chess," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9887 LNCS, pp. 88–96, 2016.

[43] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *Advances in Neural Information Processing Systems (NIPS)*, pp. 3104–3112, sep 2014.

[44] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," *Proceedings of the 22nd international conference on Machine learning - ICML '05*, pp. 89–96, 2005.

[45] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to Rank : From Pairwise Approach to Listwise Approach," *Proceedings of the 24th international conference on Machine learning*, pp. 129–136, 2007.

[46] F. Xia, T.-Y. Liu, J. Wang, W. Zhang, and H. Li, "Listwise Approach to Learning to Rank - Theory and Algorithm," *Proceedings of the 25th international conference on Machine learning - ICML '08*, pp. 1192–1199, 2008.

[47] Y. Lan, "Position-Aware ListMLE : A Sequential Learning Process for Ranking," *Uai*, 2014.

[48] L. Pang, Y. Lan, J. Guo, J. Xu, J. Xu, and X. Cheng, "DeepRank: A New Deep Architecture for Relevance Ranking in Information Retrieval," 2017.

[49] J. Kanda, A. de Carvalho, E. Hruschka, and C. Soares, "Using Meta-learning to Recommend Meta-heuristics for the Traveling Salesman Problem," in *2011 10th International Conference on Machine Learning and Applications and Workshops*, vol. 1, pp. 346–351, IEEE, dec 2011.

[50] P. Goyal and E. Ferrara, "Graph Embedding Techniques, Applications, and Performance: A Survey," 2017.

[51] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online Learning of Social Representations," 2014.

[52] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation Learning on Graphs: Methods and Applications," pp. 1–23, 2017.

[53] L. Da, F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas, "Characterization of complex networks: A survey of measurements," *Advances in Physics*, vol. 56, no. 1, pp. 167–242, 2007.

[54] A. Grover and J. Leskovec, "node2vec: Scalable Feature Learning for Networks," 2016.

[55] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[56] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[57] N. Reimers and I. Gurevych, "Optimal Hyperparameters for Deep LSTM-Networks for Sequence Labeling Tasks," 2017.

[58] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong, and W.-c. Woo, "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting," pp. 1–12, 2015.

[59] R. A. Rossi, R. Zhou, and N. K. Ahmed, "Deep Feature Learning for Graphs," pp. 1–11, 2017.

[60] C. Li, X. Guo, and Q. Mei, "DeepGraph: Graph Structure Predicts Network Growth," 2016.