UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

GRAPH ATTENTION AND PERSISTENCE FOR TRAVELING SALESMAN PROBLEM

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

BY

JOSE E. AGUILAR ESCAMILLA
Norman, Oklahoma
2023

GRAPH ATTENTION AND PERSISTENCE
FOR COMBINATORIAL PROBLEMS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Acknowledgements

I would like to thank the wonderful team of advisors and mentors that helped me discover my passion for research at the University of Oklahoma. I am thankful for the patience and interest Dr. Dean Hougen has shown in my development as a scientist and his help through the many obstacles this thesis encountered. Similarly, I am very thankful to Dr. Dimitrios Diochnos for his time and advice as I began pursuing my first research endeavors and for his availability as I learned, made mistakes, and succeeded in our joint research endeavors. I am also thankful with Dr. Sophia Bollin-Dills, who provided an environment of growth and motivation to pursue research and graduate education through the McNair Scholars Program at OU.

More personally, I am thankful for the help and support my wife, Breashay Aguilar, has given me as I completed this milestone in my career. Similarly, I am grateful to my parents, Georgina Escamilla Hernandez and Efrain Aguilar Martinez, and my siblings for their wisdom and guidance as I encountered obstacles. I am also indebted to the Burke family, who became my second close family during my time in the United States, offering the support my family wished to provide but could not due to distance.

I also would like to thank the OSCER team for providing support and resources to carry out this research project and the research scientists from the

Oklahoma Aerospace and Defense Innovation Institute working at the OC-ALC offices on the south campus for their wisdom and advice with this project.

Abstract

Combinatorial optimization problems have long been a computationally-challenging family of problems with high importance within science. Although algorithms to solve such problems exist, these classical/exact algorithms tend to become computationally intractable as the problem size increases. Due to the relevancy of such problems in real life, research has explored other algorithms that forego the optimality guarantee of classical algorithms. In doing so, heuristic algorithms have achieved fast inference times with performance that is not too far from the optimal performance of classical algorithms. A particular heuristic algorithm from artificial intelligence, the attention model (AM), has achieved state-of-the-art performance in prevalent combinatorial optimization problems such as the traveling salesman, vehicle routing, and orienteering problems, where the gap between classical and heuristic algorithms was diminished. This success is attributed to the ability of the AM to tend to the structure of the input through the use of graph attention (GAT) mechanisms, a type of graph neural network. While the mechanism by which these models extract structural information, it is unknown what kind of information is extracted. This thesis presents a novel variant of the attention model (AM), the *persistence attention model* (AM-P), which explicitly uses a type of structural information, *persistent homology.* The model is tested on the traveling salesman problem with different problem node counts to compare the performance of the attention model with and without persistent homology information. It is hypothesized that persistent homology information will help the attention model better exploit the structure of the model, achieving better performance on combinatorial problems. This hypothesis is demonstrated *false* as there is no statistically-significant differences between the performance of the AM and AM-P. This negative result raises the possibility that the attention model may already extract structural information similar to persistent homology through its GAT mechanisms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Combinatorial optimization problems encompass a family of problems with historical importance for many fields of science. Some examples of popular problems, such as traveling salesman, vehicle routing, and orienteering problems have been heavily studied for their implications in theory as well as their relevancy for real-world problems. Many algorithms have been developed due to research into these problems. Yet, due to the difficulty and computational hardness of these problems, classical algorithms often fail to solve large problems within an acceptable time.

Given the importance of combinatorial optimization problems for solving real-life problems, research has explored new directions in algorithm design to circumvent classical algorithms' computational intractability. Many of these algorithms, called *heuristic* algorithms, forego the theoretical guarantee that a classical, exact algorithm will always produce an optimal solution. In doing so, these algorithms produce solutions faster than classical, exact algorithms at the cost of producing imperfect solutions.

With the popularization of artificial intelligence, research began exploring new ways to employ machine learning to solve combinatorial optimization problems. A recent success was achieved by the dynamic attention model (AM-D) [Peng et al., 2020], based on the attention model Kool et al. [2019], which achieved state-of-the-art performance on many combinatorial optimization problems. Its solutions were shown to further close the gap between heuristic and exact algorithms, maintaining fast inference times in problems where exact algorithms become highly intractable.

The key to the success of the attention model and its dynamic variant is due to the ability of its Graph Attention mechanisms, a subtype of the graph attention model, to extract graph information effectively. Previous heuristic algorithms could not often extract and utilize structural information from the shape of a given problem. This was hypothesized to inhibit good performance by heuristic algorithms. With the introduction of graph neural networks, this hypothesis was proven true as graph neural networks helped improve performance on many problems, including combinatorial optimization problems [Vinyals et al., 2015].

The rise to prominence of GNN and GAT models highlights the renewed interest and importance of the structural information of data. While these techniques experimentally showed an excellent ability to capture the high-level structure of the data, there is a poor understanding of what kind of structural information is extracted. Understanding how these models decipher structural information can shed light to understand better the problems these algorithms solve (like TSP) and increase awareness of the importance of the intrinsic structure of data points in machine learning.

Within mathematics, *algebraic topology* provides well-studied and theoretically-grounded tools for analyzing the structure of mathematical objects. Recently, the development of computational methods to perform this analysis has allowed

Topological analysis to extend past mathematical objects into real-life data. This new field of data analysis called *topological data analysis (TDA)* has allowed scientists better understand the nature of data past the individual datum.

Many researchers, seeing the potential of TDA, have investigated whether topological information could aid machine learning in solving problems where the data structure is thought to be critical [Hensel et al., 2021]. By using *persistent homology*, the flagship tool from TDA, scientists have achieved state-of-the-art in several graph-based problems by using topological signatures to inform a machine learning model of the data structure.

Another successful application of persistent homology is introducing topological priors into machine learning through parameter regularization [Brüel-Gabrielsson et al., 2020]. The work by Brüel-Gabrielsson et al. [2020] showed how topological regularization could improve the stability and performance of neural network models on specific problems.

Research has also identified stability as a potential property that could be introduced to machine learning models using TDA. Persistent homology signatures have the theoretical stability property amid the noise [Carlsson and Vejdemo-Johansson, 2021]. This property can be desirable in neural networks, which are susceptible to adversarial attacks capable of derailing learning Szegedy et al. [2014].

Given these potential benefits of persistent homology, this thesis aims at studying whether introducing persistent homology could improve the performance of the Attention Model, the state-of-the-art combinatorial optimization heuristic algorithm. Because it is unknown what kind of structural information the attention model uses, it is uncertain whether persistent homology signatures will improve or transfer desirable properties to the attention model.

This thesis presents a novel variant of the Attention Model: the persistent attention model (AM-P). This model utilizes persistent homology signatures computed from its input to inform the model of the structural characteristics of the problem. This model is then tested on the Traveling Salesman Problem (on 10, 20, and 50 node problems) and compared with the results by the original Attention Model.

The experiments showed little to no change in performance by the AM-P concerning the AM model. Statistical analysis showed that the models' performance did not differ significantly among the different traveling salesman problems. These results indicate that persistent homology signatures are less useful for the AM, suggesting that GAT models could share the same property.

While no improvement in heuristic algorithm performance was achieved, the results hint at a possible special relationship between GAT models and persistent signatures. Further research is encouraged to study the nature of this relationship, asking whether information similar to persistent homology signatures is already being computed within the AM/AM-D and whether GAT models share this ability.

Chapter 2

Background

This thesis's main contribution encompasses three general research fields: combinatorial optimization, deep learning, and topological data analysis. Tools and ideas are extracted from deep learning and topological data analysis to create a novel architecture, the persistence attention model (AM-P), capable of solving combinatorial optimization problems (here, focusing on the traveling salesman problem.) It is through comparing the learning behavior and performance of the persistence attention model and the attention model, the main inspiration of the AM-P architecture, that results are obtained to assess the effect of topological data analysis on learning.

To gain an appreciation and understanding of the contributions hereby presented, a survey of combinatorial optimization problems, deep learning, and topological data analysis is presented, focusing on the most relevant topics to this research. Sections 2.1 and 2.2 cover the field of combinatorial optimization, focusing on the traveling salesman problem and the two main algorithm families that solve these problems: classical/exact and heuristic. Next, in Sections 2.3, 2.4, 2.5, a survey on deep learning is presented, covering the field of neural networks and their importance for solving different problems and the Transformer, a prevalent and recent deep-learning architecture that catalyzes the AM and AM-P models. Building on top of these Sections, section 2.6 presents the attention model (AM), covering in detail the architecture as well as its reinforcement learning training algorithm. Finally, section 2.8 provides an intuitive survey on topological data analysis (TDA), focusing on tools that will help extract structural information from data.

2.1 Combinatorial Optimization Problems

Combinatorial optimization problems are of great interest within the mathematical and computer science community, given their significance for many real-world problems and their relationship with the P vs. NP problem [Russell and Norvig, 2020]. At their core, combinatorial optimization problems entail generalized decision problems, where a solver attempts to make the best decisions over a discrete set of options [Hoos and Stützle, 2005]. Here, a "best" solution is discriminated by the help of a quality function, which the solver attempts to maximize.

There exists a myriad of combinatorial optimization problems that are of great interest to the scientific community. These problems include the traveling salesman, vehicle routing, and orienteering problems [Golden et al., 1987a]. These problems are instrumental, given their use to model many real-world problems. For this reason, combinatorial optimization algorithms play an essential role in solving challenging real-world problems efficiently.

An example use-case of combinatorial optimization is modeling the problem of finding an optimal route for an airplane through an existing set of *waypoints*. In real-world aircraft navigation, commercial aircraft navigate using a network of geographical locations referred to as *waypoints* or fixes. These waypoints help pilots draft a trajectory without requiring access to visuals of the airplane surroundings. Thus, a trajectory for a commercial airliner is a list of different waypoints that the pilot will visit. Provided that a plane has a finite amount of fuel, the problem of drafting a trajectory encompasses selecting a set of waypoints to optimize resources. This problem can be modeled as an orienteering problem [Golden et al., 1987b], where the waypoints are nodes, and the goal is to find a trajectory that uses the least amount of fuel. If we remove the many complexities of air navigation, the aforementioned problem can be solved simply using an algorithm to solve an orienteering problem.

Despite the excellent understanding of combinatorial optimization problems, many problems lack efficient algorithms [Russell and Norvig, 2020]. Specifically, as the problem size increases, algorithms tend to become computationally intractable due to the fast increase of the set of solutions to the problem, which an algorithm must traverse. This problem is encapsulated within the P vs NP problem. It is unknown whether a polynomial-time deterministic algorithm exists to solve NP problems, a class to which many combinatorial optimization problems belong.

2.1.1 Definition of Traveling Salesman

This thesis will make use of the Traveling Salesman Problem (TSP) to compare two heuristic algorithms. This problem is chosen given the historical importance of the problem as well as its straightforward formulation.

Among all the TSP variants, I use a classical variant of the problem (same used in [Kool et al., 2019]) where the objective is to find the shortest route among graph nodes *without* re-visiting a node. Specifically, a TSP comprises a graph G containing a finite number of nodes $N \in \mathbb{R}^2$. The graph of nodes is assumed to be fully connected; thus, an edge always exists that connects any two nodes in the graph. With this setting, a problem solution would entail an ordered set of nodes where no node is repeated. Because of the constraint on the repetition of nodes, all solutions will always have a cardinality equal to the total number of nodes used in the problem. A solution is said to *solve* the problem if its length of travel is the smallest among all possible solutions. Here, the length of travel is defined as the sum of the Euclidean distances between each node following the order of the solution: $\sum_{i=0}^{|S|-2} ||S_{i+1} - S_i||$.

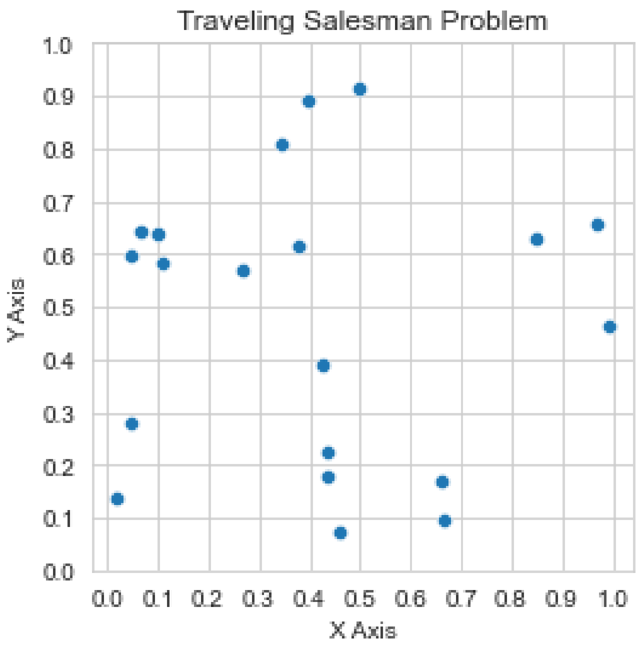


Figure 2.1: Example TSP problem. A solution to this problem would connect all points in the graph, forming a network with minimal distance to visit all nodes.

2.2 Heuristic Approach through Deep Learning

As previously mentioned, classical (or exact) algorithms for combinatorial optimization problems often struggle with time complexity. These algorithms possess a guarantee that an optimal solution will always be found. Unfortunately, by holding this guarantee, classical algorithms become computationally intractable as the problem size increases.

Given the importance of combinatorial optimization problems in real life, where problems can become intractable, recent research has focused on alternatives to classical algorithms. This field of study looks at *heuristic algorithms*, which forego the guarantee that the algorithms will always find an optimal solution, instead allowing some degree of error. Breaking this guarantee permits heuristical algorithms to obtain low inference times for finding solutions.

The current state-of-the-art heuristic algorithms use artificial intelligence, where neural network models have been most successful [Russell and Norvig, 2020]. These models extract information from the input problem to find a solution that is most likely to be optimal. Compared to classical algorithms, the time to find a solution depends on how long the information is fore-propagated in a neural network, which is often very fast in modern hardware.

Despite lacking the theoretical guarantees of classical, exact algorithms, neural network approaches to combinatorial optimization have achieved significant results, finding the best solution to problems with high probability. For example, the state-of-the-art algorithm for graph combinatorial optimization problems, the dynamic attention model [Peng et al., 2020], can find solutions to problems within seconds with deviance of a small percentage from the solutions generated by classical, exact algorithms. This quality of heuristic approaches makes them even more useful for real-world problems, where the importance of fast solutions may supersede optimality.

2.3 Neural Networks

Current state-of-the-art approaches, such as the Dynamic Attention Model [Peng et al., 2020], utilize several different types of artificial neural networks. This section presents an overview of the history and theory behind artificial neural network models to facilitate understanding of the theory of the attention model presented in Sections 2.4, 2.5, and 2.6. First, the biological inspiration for neural networks is presented. Then, a historical description of the first ANN models is given, emphasizing the transition from biology to computational models. Finally, the distinction between ANN and deep learning models is provided.

2.3.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a type of connectivist machine learning model inspired by the brain's inner workings. Here, connectivity describes ANNs as a computational model where small, simple, and independent computational units are combined to create more powerful computations. Neural networks have become a promising field of research due to the remarkable progress these models have achieved in many different areas and problems.

The atomic unit of an ANN is the neuron, which is inspired by the biological neuron. Within the brain, neurons are cells composed of three main parts: a cell body called the *Soma*, a series of terminals that receive electric impulses called the *Dendrites*, and a long appendage that communicates outgoing signals called the *Axon* (see figure 2.2). Neuronal cells communicate information and

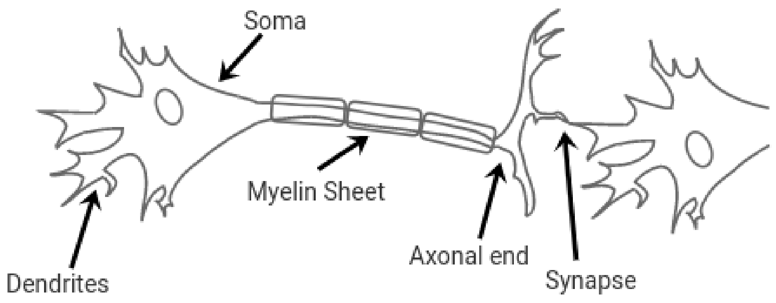


Figure 2.2: Anatomy of a biological neuron.

conduct computations through the emission and reception of electrical pulses, also called *spikes*. These spikes travel from a sending neuron's (pre-synaptic neuron) Soma, down its Axon, and into a receiving neuron (post-synaptic neuron) through a synapse between the dendrites of the post-synaptic and the axon on the pre-synaptic neuron. Through the synapse, a very small space formed between dendrites and an axon, electrical current can be transferred by releasing chemicals from an Axon, creating an electrical charge that accumulates on the Soma of the post-synaptic neuron.

As neurons exchange electrical currents, an electrical charge builds on the neurons' neuronal membrane, accumulating over time. When some specific electrical amount is reached, namely the neuron's *threshold*, the neuron is said to *activate* and release an electrical charge that travels through its axon. The stored electricity in the Soma is depleted, thus bringing the neuron to a state where the electrical charge can accumulate again.

Commonly, in the brain, neurons belong to larger brain structures where neurons process specific stimuli and relay information to other parts of the brain. These interactions can be observed in real-time with the help of an EEG. Activity in the different parts of the brain tends to increase as information flows or is processed. Oftentimes, high activation is related to the "detection" of specific stimuli a brain structure attends to. This hierarchy from the individual, simply-behaved neurons up to entire structures capable of perceiving and processing complex stimuli constitutes the essence of connectivism in biological systems that inspires artificial neural networks. The myriad of chemical reactions and interactions among neurons is often complex and not fully understood, so artificial neural networks resort to *approximating* these processes rather than precisely simulating them.

One of the first computational models developed as a simplification of the interactions within the brain is the *Perceptron*. The perceptron was developed by Rosenblatt [1958]. The core observation behind the model is the preference of neurons for specific pre-synaptic inputs. Specifically, Axons are covered on a fatty substance, the myelin sheet, that aids in transporting electrical charge from the neuron's body. The myelin sheet is variable among Axons, and it has been found that the myelin sheet is most present in axons of neurons that are most crucial for the detection of the targeted stimuli by a neural structure. In this way, spikes arrive faster to post-synaptic neurons from Axons with a thick myelin sheet than from Axons with a thin one. In this way, synapses whose axons have a thick myelin sheet have a more substantial influence on post-synaptic activation. The variability of the myelin sheet works as an indicator of the importance of the input, essentially working as a type of memory based on previous interactions.

Following the idea that the myelin sheet contains information on the input's importance, the perceptron defines a vector of *weights* that indicates the importance of a given input, modeling the myelin sheet. Then, by computing the product of input information with its respective weight, the perceptron obtains a new weighted vector that is summed over to model the post-synaptic activation that a biological neuron would receive. Finally, an activation function is applied to the sum of products. In the perceptron, this activation is the signed function, which outputs 1 if the sum of products is larger than some threshold or zero otherwise.

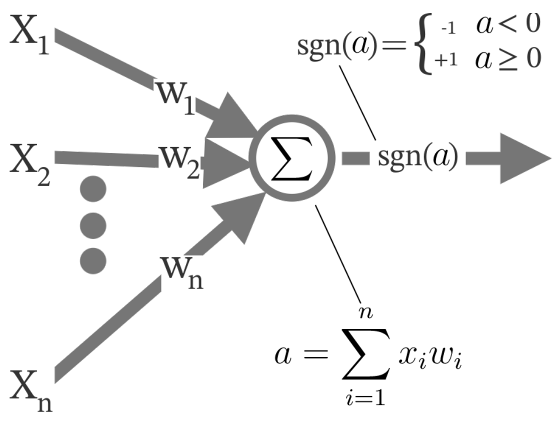

X_1
X_2
X_n
W_1
W_2
W_n
$\sum$
$\mathrm{sgn}(a) = \begin{cases} -1 & a < 0 \\ +1 & a \geq 0 \end{cases}$
$\mathrm{sgn}(a)$
$a = \sum_{i=1}^{n} x_i w_i$

Figure 2.3: Visual representation of the Perceptron model.

While the perceptron demonstrated initial promise to solve challenging problems, excitement disappeared after proving that the model could not correctly model simple, non-linearly separable functions such as an exclusive or (xor) gate. This loss of hope in the model would lead to the ”winter of AI,” where the idea of neural networks was abandoned [Russell and Norvig, 2020].

Fortunately, some researchers kept working to help neural networks improve. Progress arrived once the backpropagation algorithm was developed and popularized. Using multi-layer models had been identified as a possible solution to model non-linear problems, but training such networks was challenging. Backpropagation provided a theoretically-grounded way to train even the largest of neural networks. Additionally, the perceptron activation function was replaced with continuous, differentiable functions to allow neural networks to perform regression. This progress launched artificial neural networks as the leading force behind the AI revolution in the 21st century.

2.3.2 Deep Neural Networks

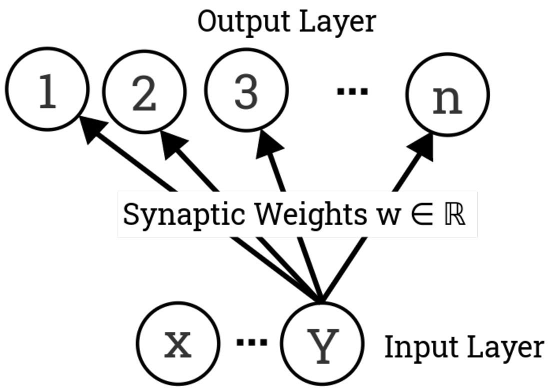


Figure 2.4: Overview of a single neural network layer. A linear transformation is applied to the input information before a summation and activation function are applied. Stacking these layers produces a multi-layered model.

Modern artificial neural networks utilize non-linear activation functions. The most popular activation functions are the logistic sigmoid (often referred to simply as sigmoid), hyperbolic tangent (tanh), and rectified linear (ReLU[1]) functions, given their fast computation and differentiability properties.

Sigmoid	Tanh	ReLU
$\frac{1}{1+e^{-x}}$	$\frac{(e^{x}-e^{-x})}{e^{x}+e^{-x}}$	$max(0,x)$

Table 2.1: Sigmoid, Tanh, and ReLU equations.

Additionally, modern networks are composed of many layers of varying sizes. This characteristic earns them the name of *deep neural networks*, distinguishing these models from their earlier, shallower, and simpler ancestors.

Despite the great success of neural networks, certain classes of problems proved to be still challenging to solve with regular neural networks. It was soon found that the simple computation of dot products passed through non-linear

[1]Although ReLU is not differentiable at 0, a constant value tends to be chosen to allow differentiation as a piecewise function.

functions may be blind to structural information not captured by this process. As a response, research developed new types of deep learning models, such as recurrent neural networks [Jordan, 1986], convolutional neural networks [Goodfellow et al., 2016], and attention [Bahdanau et al., 2015]. This diversification of neural networks has allowed the creation of hybrid models capable of exploiting the best qualities of each sub-type of ANN to push the state-of-the-art on many problems.

2.4 The Transformer: An Attention-based Encoder Decoder

One of the most impactful deep learning architectures is the Transformer. Initially presented in the seminal paper "Attention is all you need" by Vaswani et al. [2017], the authors proposed a new deep learning architecture that achieved state-of-the-art in language translation and modeling. The Transformer was unique in that it deviated from the popular neural network model used for language translation at the time (recurrent neural networks RNNs). Instead, it used a technique called attention, which was shown to help circumvent the many problems of RNNs.

Given the complexity of the transformer, this section will present the background information on the previous research that lead to the Transformer. First, a survey of the transition from RNNs to encoder-decoder models is presented. Then, a study of attention mechanisms is given. In the next section (section 2.5), the transformer model will be presented in its entirety.

2.4.1 Encoder-Decoders

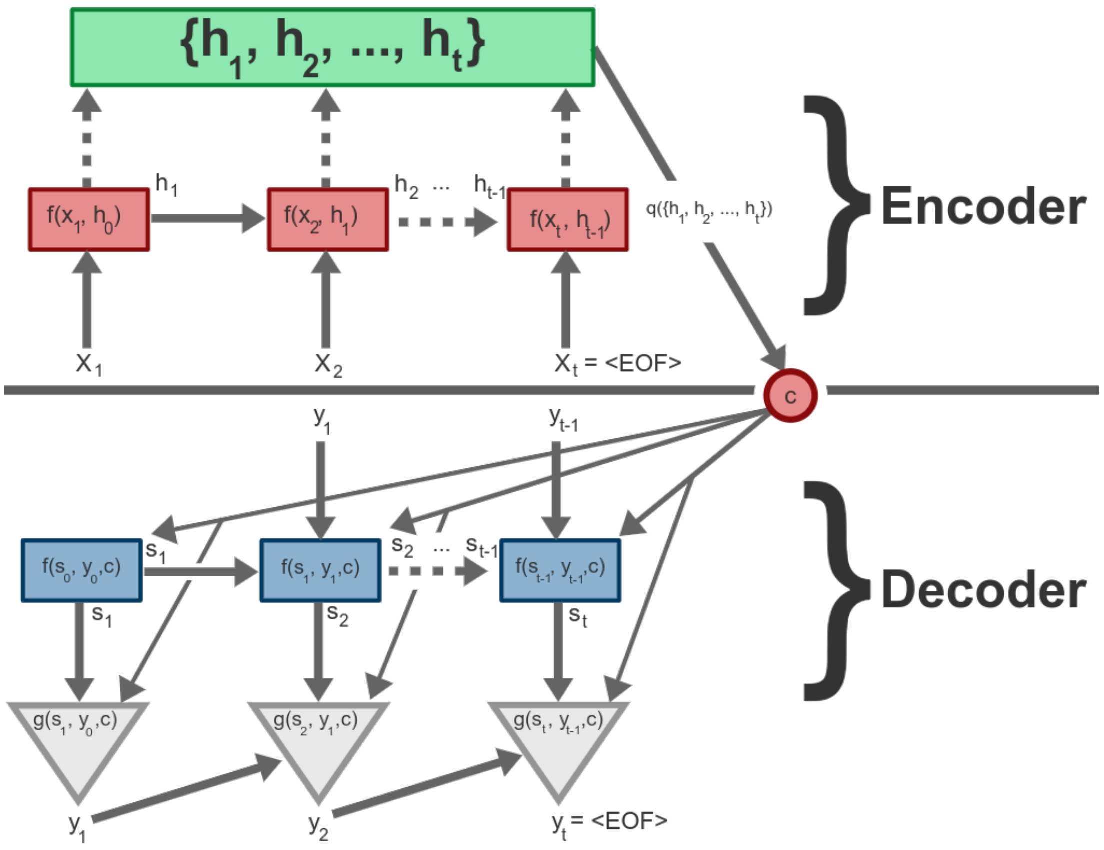


Figure 2.5: The Encoder-Decoder architecture as defined by Cho et al. [2014]. Note that the x values are assumed to come in as embedded inputs; h_i is the hidden state of the encoder; s_i is the hidden state of the decoder; function q is simply the last hidden state. In Cho et al. [2014] q is simply the last hidden state h_t; f is some activation function; g is softmax.

The encoder-decoder architecture was initially presented in Cho et al. [2014], and as its name suggests, it is comprised of two neural networks called an encoder and decoder (Figure 2.5). The architecture developed from research looking to better model the natural language processing problems. The model's core objective in using an *encoder* module is to produce a new representation of the input text that summarizes the whole sentence in some latent space. This latent-spaced context would contain abstract information over the whole input sentence that should be most useful for translation.

The *decoder* in the architecture is designed to model a probability distribution over some vocabulary. Following statistical machine translation (SMT), we can define the translation process as a two-part distribution:

$$p(\mathbf{f}|\mathbf{e}) \propto p(\mathbf{e}|\mathbf{f})p(\mathbf{f}) \tag{2.1}$$

Where the left hand is the *translation model*; the right hand is the *language model*; $\mathbf{e}$ is some sentence in some language; $\mathbf{f}$ is the translation into some other language. In language translation, the interest is in modeling $p(\mathbf{f}|\mathbf{e})$. We can further define this probability as: $\prod_{i=0}^{L_f} p(\mathbf{f_i}|\mathbf{e})$, where L_f is the length of the translation and $\mathbf{f_i}$ is the ith translated word *over the vocabulary.* This product of probabilities defines the individual conditional probabilities on what words are most likely to be in the translation at every point in the translated sentence given some sentence $\mathbf{e}$. Cho et al. [2014] showed that we could model the individual conditional probabilities $p(\mathbf{f_i}|\mathbf{e})$ using a recurrent neural network after encoding the input sentence into some latent space. This encoding, also called the sentence *context*, is computed by the encoder.

The decoder, which models a probability distribution to find words most likely to be the translation, utilizes the information produced by the encoder (context c) to produce conditional probabilities. The probability distribution would be with respect to the vocabulary available to the model, defined during training. First, the RNN computes the hidden state using the function $f(s_{t-1}, y_{t-1}, c)$. This will return a "score of likelihood" on the word most likely to be the translation word. Then, function $g(s_t, y_{t-1}, c)$, commonly $softmax$, is applied to obtain the conditional probability. The word with the highest probability is chosen as the predicted word by the decoder as y_t. As decoding progresses, these produced words, and the hidden states are fed back into the RNN. At some point, the decoder will output some end-of-sentence token (defined when the model was being trained) to indicate the end of decoding/translation.

2.4.2 Attention Mechanisms

Attention mechanisms were first presented by Bahdanau et al. [2015] in their seminal work titled "Neural Machine Translation by Jointly Learning to Align and Translate." This paper became a revolutionary push forward in natural language processing. Since then, attention has evolved and expanded, culminating with the development of self-attention as presented in the Transformer paper [Vaswani et al., 2017].

Bahdanau Attention

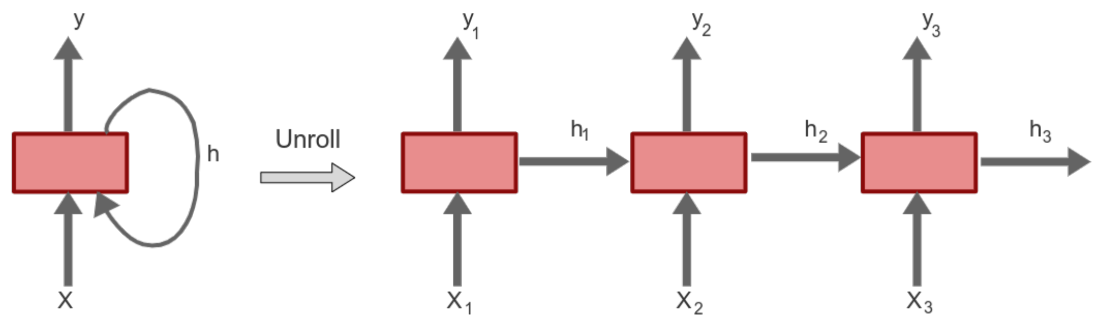


Figure 2.6: Recurrent Neural Network. The inputs $X_1, ...$ can be sequential information such as a word in a sentence. The cell processes them one at a time and maintains a certain amount of information via its recurrent hidden state h. This hidden state is similar to a flip-flop in hardware circuits, except that it is "fuzzy." Since this hidden state is the only means of memory preservation, information from previous inputs may be lost if the input is sufficiently large.

The idea behind additive attention mechanisms developed from the idea that some words within a sentence may have a higher importance in the translation of the sentence than others. The encoder-decoder architecture presented by Cho et al. [2014] only used the context produced at the very end of the encoding process (figure 2.5 context c). A problem that was observed when only using this information was that information computed at the beginning of an input sentence tended to be lost, which caused models to underperform in large sentence translations. The cause for this loss of performance was due to the information pipeline that RNNs use. During encoding, individual words are passed through an RNN network where the previous outputs produced are reused to maintain a sense of memory (see figure 2.6). The problem with this idea is that information has to travel through this hidden state memory, and since its size is fixed by the number of output units of the RNN, data would eventually be lost in a large-enough sentence.

Bahdanau attention was thus introduced to attempt to fix this problem. The authors observed that the final context produced by the RNN encoder is always dependent on the information produced by previous passes through the RNN. Using the idea that some of the previous states may be more important than others, a ”selection” mechanism that weights each state on the importance for the final translation was used. Then, a context would be computed at the end of processing the input sentence, allowing the model to better retain information introduced earlier in the sentence.

Attention in the Encoder/Decoder

Based on the Encoder/Decoder architecture, Bahdanau et al. [2015] modified how the context information from the decoder was computed. First, they defined the context information to be generated by a function $q(h_1, ..., h_{T_x})$ that is dependent on the hidden states produced by the encoder rather than $q(h_1, ..., h_{T_x}) = h_{T_x}$, as Cho et al. [2014] defined it. This allows the function q to have discretion on the information to select, tailoring the context to the decoder's state. This information pathway can be appreciated in figure 2.5:

$$\begin{aligned} c_i &= \sum_{j=1}^{T_x} \alpha_{ij} \cdot h_j \\ s.t.\ \ \alpha_{ij} &= \frac{exp(u_{ij})}{\sum_{k=1}^{T_x} exp(u_{ik})} \\ u_{ij} &= a(s_{i-1}, h_j) \end{aligned} \tag{2.2}$$

Where c_i is the context to be used when producing the i^{th} conditional distribution (output of decoder); h_j is the j^{th} hidden state produced by the encoder; it is also called the *annotation* of the j^{th} input; α_{ij} is the alignment weight of the j^{th} annotation for making the i^{th} translation; u_{ij} is the score of the alignment of the jth annotation concerning the i^{th} translation; $a(s_{i-1}, h_j)$ is some function (an ANN in Bahdanau et al. [2015]) that provides a score on the alignment, or importance, of information. This alignment weight indicates how vital the annotation h_j is for producing the following translation with the highest probability of actually being in the translation. This can also be seen as a primary attention mechanism for identifying the importance of annotations. Using this alignment model, the decoder has discretion on what information to include in the context vector c. The decoder can filter the available information using additive attention, creating a more "relevant" context vector.

2.5 Attention is All You Need (Transformer)

Vaswani et al. [2017] introduced the Transformer alongside the idea of self-attention, an improved type of global, dot-product attention. The architecture managed state-of-the-art performance on language translation problems by using only attention and foregoing recurrent neural networks. The encoder/decoder used dot product global self-attention to obtain information across the entire input encoding. This greatly decreased the computational complexity required to access information across a sentence as constant time complexity.

2.5.1 Overview of Architecture

Figure 2.7 presents a visual overview of the transformer. Here, the input words are fed to an input embedding layer that transforms the word into a vector. Then, information on its position is added by using a positional encoding. The resulting input is then passed through N encoder modules, where each module has a multi-head attention layer whose output is combined with residual information from the input and normalized to then be passed through a feed-forward neural network whose output is also added with residual information from the attention layer's output and normalized. Once encoding is complete, we are left with annotations on the input words, similar to those found in other encoder-decoders. The decoder uses these annotations to produce the output translation.

The decoder first embeds the inputs received and adds positional information via a positional encoding (similar to encoding). Then, the resulting input is passed through N layers (the same number as the encoder), where the information from the annotations is used. Within a decoder unit, the information is first passed through a masked multi-head attention layer where the information from

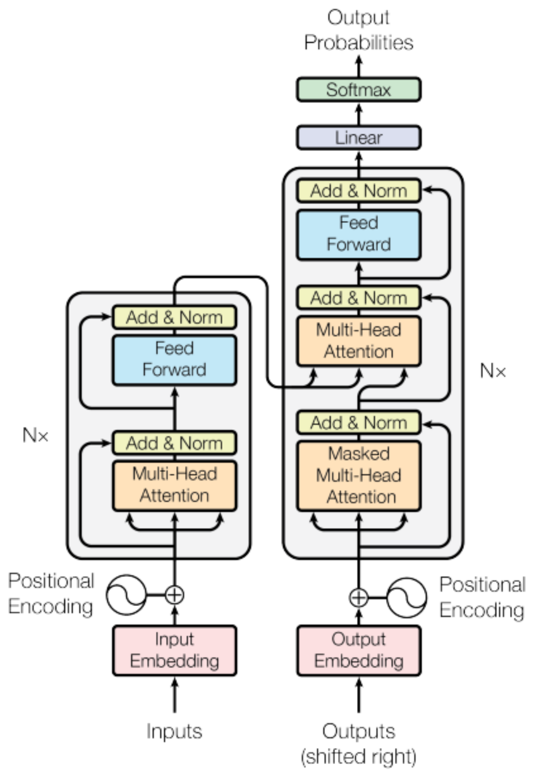


Figure 2.7: The Transformer encoder-decoder architecture. The encoder can be appreciated on the left and the decoder on the right. Note that each of the modules is stacked N times. The words selected by the decoder are re-encoded and fed to the next decoding process until an ”end-of-sentence” token is given. [Vaswani et al., 2017]

future words is masked. Then, this information is combined with the input via residual connection and normalized. Next, the output is passed through another multi-head attention layer where the encoder's annotations are introduced. After adding residual connection and normalizing, the output is passed through a feed-forward neural network to produce a final output (added with residual information and normalized). The output of the decoder stack is linearly transformed (mapping from embedding space to vocabulary space) to produce scores on the most likely vocabulary words. Finally, the output is converted into a probability distribution through a softmax layer, and the most likely word is chosen.

A more detailed explanation of the processing and computation of data is be provided in the next subsections. Each subsection gives detailed input and output matrix and tensor shapes as they are processed. The main objective in introducing such detail is to allow a better understanding on the inner workings of the model, as these computations will be used by the model presented in this thesis. Additionally, this level of detail should allow any reader with knowledge of tensor and matrix computation software to implement their own version of the model presented independently of the provided implementation.

2.5.2 Encoder Operation

Assume that we are given a matrix of dimensions $n_{examples} \times n_{words}$ as input, where $n_{examples}$ is the number of examples to process; n_{words} is the number of words in each sentence. Note that each example may not have the same number of words, and this is fine, assuming that each example is independent of the others. If we want to process paragraphs, these shall all be included in the same example.

Input Embedding

The $n_{examples} \times n_{words}$ matrix is embedded using an embedding layer to produce a tensor of shape $n_{examples} \times n_{words} \times d_{model}$, where d_{model} is the dimension of the embedding, and it is chosen as a hyperparameter. The embedding layer can also be learned or pre-selected.

Positional Encoding

The embedded input is enhanced with positional information by adding a positional encoding. Because the Transformer lacks recurrence and the output translations are dependent on the temporal information of the input, Positional Encodings introduce this information by computing vectors of dimension d_{model}. In the paper, Vaswani et al. define the encoding to be:

$$\begin{aligned} PE_{(pos,2i)} &= sin\left(\frac{pos}{10,000^{2i/d_{model}}}\right) \\ PE_{(pos,2i+1)} &= cos\left(\frac{pos}{10,000^{2i/d_{model}}}\right) \end{aligned} \tag{2.3}$$

The positional encoding information is then added to the embedding information, conserving the shape of the matrix $(n_{examples} \times n_{words} \times d_{model})$. There is no particular reason why the positional information is added rather than introduced with another operation.

Multi-Head Attention

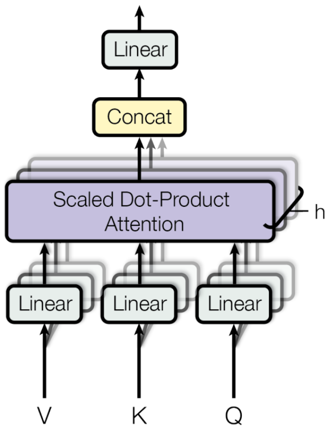


Figure 2.8: Multi-Headed Attention. Note that the module's output will not have any dimension of h since the concatenation drops that dimension. One can think of the h heads as different attention mechanisms that look at different types of information on the inputs. [Vaswani et al., 2017]

After embedding and positional encoding, the input is passed through an attention layer that creates a custom context vector. The input is used as query, key,

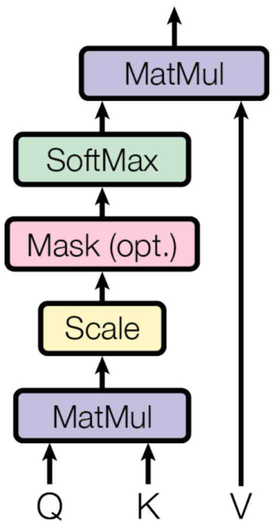


Figure 2.9: Multiplicative attention. This attention is used by multi-headed attention.

and value. The multi-head mechanism is characterized by h stacked dot-product attention modules whose output is concatenated and linearly transformed. This produces a $n_{examples} \times n_{words} \times d_{model}$-shaped matrix. The output is added with a residual connection coming from the input after embedding, positional encoding, and normalization.

Feed Forward Neural Network

The output matrix by the multi-head attention is passed through a fully-connected neural network. The output is then added and normalized as well. This output constitutes the annotations that are commonplace in encoder/decoder architecture.

2.5.3 Decoder Operation

New input of shape $n_{examples} \times n_{words}$ (either an initial decoding token, or the previously-generated encoded word) is embedded into $n_{examples} \times n_{words} \times d_{model}$

and positionally-encoded as in the encoder. After inference, the output of the decoder is reintroduced every time as the input of the decoder.

Masked Multi-Headed Attention

The input is passed through a masked multi-headed attention module as query, key, and value, where a mask is introduced to avoid paying attention to information that is not yet available, ensuring that the ith prediction is dependent only on the previously-generated words. This layer produces a $n_{examples} \times n_{words} \times d_{model}$ that is then added and normalized using the embedded and positionally-encoded input.

Multi-Head Attention

The output from the masked multi-headed attention module is introduced as the query, while the annotations from the encoder provide the key and values. The intuition behind this module is that we can search the collected information produced by the encoder to find information that may be related to the information known for decoding. Since the key and values are the annotations from the encoder, the attention mechanism will search for information most relevant to the input data. The output is added and normalized with respect to the output from the masked multi-headed attention module.

Feed Forward Neural Network

Finally, the output from the attention module is passed through a fully connected feed-forward neural network. The network's output is then added and normalized concerning the output of the multi-headed attention mechanism via a residual connection.

Output Probabilities

The ANN output is linearly transformed into a matrix of shape $n_{examples} \times n_{words} \times d_{vocabulary}$ where $d_{vocabulary}$ is the dimension of the vocabulary vector, which can be seen as a bag of words. The linear transformation essentially provides a score value for the likelihood of being in the translation for each word in the vocabulary. This score is passed through a softmax layer to produce the final conditional probabilities. The Transformer would output the word with the highest probability as the chosen word. This produced word is reintroduced as the next input of the next decoding cycle.

2.6 The Attention Model

The transformer, since its introduction, has influenced different fields of research outside of language translation. An example of this influence is the development of *the attention model* [Kool et al., 2019], which achieved state-of-the-art on combinatorial optimization problems through the adoption of the Transformer architecture to use as part of a Graph Attention (GAT) model. A short survey on the development of graph neural networks, which would catalyze Graph Attention GAT models, is presented. Then, the theory behind the attention model is introduced.

2.6.1 Graph Neural Networks and Graph Attention

Graph neural networks (GNNs) are a subtype of deep neural networks that are capable of solving graph problems [Scarselli et al., 2009]. Traditional DNNs, despite their high performance on many other problems, have shown to be in-capable of generalizing on graph problems. The main reason for this challenge is the requirement to vectorize graph problems, which removes graph structural information that may be important for solving a problem. To solve this problem, Graph Neural Networks (GNNs) emerged as DNN models capable of retaining the graph problem structure information for solving the problem.

Graph Neural Networks

While the idea of GNNs comes from the paper by Scarselli et al. [2009], it was not until later that formalization on the architecture of GNNs was given by Gilmer et al. [2017]. All graph neural networks (GNN) can extract structural graph information by assigning network units to each node of a graph. In doing this, the network topology is used to introduce the graph structure information. Then, through a message-passing protocol, information can be propagated through the network, sharing the information belonging to each node and using incoming information to update the state of a node (modeled as a neuron or a part of the network).

The message-passing protocol is one of the most important parts of a GNN. The simplest description of these protocols follows the equation:

$$F(t+1) = \alpha \, S \, F(t) + (1-\alpha)Y \tag{2.4}$$

where F is the information of some node at time $t+1$; α is some variable weight to control the alignment of the current state of the graph ($F(t)$) and the "ground" initial information (Y); S is some adjacency matrix where the graph structure is encoded, often modeled by a neural network; Y is the initial labels of the nodes. In a nutshell, the equation is a weighted sum that controls the speed at which information is updated, making it "harder" or "easier" to change the values of each node as iterations go by.

Despite the great success of solving graph problems, the fine-tuning of α tends to be a challenging part of training the network. This is mainly caused by the hardness of gauging what parts of a graph are more or less important for each individual neuron, which can be problem-dependent. Graph attention (GAT) networks emerged to solve this problem by using attention to automatically learn the best choice for α. This modification to the original GNN model allowed the model to best focus on the most relevant parts of the input graph, achieving state-of-the-art on many problems.

Graph Attention Network

A different perspective is taken in graph attention models, but the core idea remains. Attention replaces the idea of a message-passing protocol by becoming the protocol itself. Attention layers, as previously discussed, look at incoming information, compute an alignment score, and then, weigh the available information with respect to the incoming input to compute the most relevant information for the given input.

Attention can be seen as a message-passing protocol by observing that the computation of available information dependent on the key/value information gathers the information of the nodes to compute a new state for the query node. This follows equation 2.4, where α is defined as the alignment score. This modification has allowed GAT models to achieve state-of-the-art performance on many graph-related problems.

2.7 Attention Model

In the paper ”Attention! Learn to Solve Routing Problems,” Kool et al. [2019] presented a new Transformer-based GAT model that efficiently processed graph information to solve several combinatorial optimization problems such as traveling salesman, vehicle routing, and orienteering problems, among other variants. The attention model (AM) achieved state-of-the-art performance, surpassing many other heuristic algorithms and closing the gap between classical, exact algorithm and heuristic best performance.

The use of *reinforcement learning (RL)* is special about the AM algorithm. RL is a family of learning algorithms that use rewards to guide learning. Unlike supervised learning, where the solution to a problem is used to guide learning, reinforcement learning merely uses a signal to indicate the model’s fitness. The advantage of this learning style becomes evident when faced with a problem where labeling is non-trivial. This is the case for TSP, where large problems become intractable. Fortunately, computing a fitness function on a problem instance is trivial and quick. This makes RL well-suited for combinatorial optimization problems, which tend to belong to the NP family of problems.

Similarly to the detailed explanation given for the Transformer, the following subsections explain the inner workings of the different parts of the AM. Special note is taken of the shapes of matrices and tensors to allow any reader to implement their own version of the attention model.

2.7.1 Encoder

The encoder presented by Kool et al. [2019] is used to serve a slightly-different purpose from the original transformer for natural language processing: update

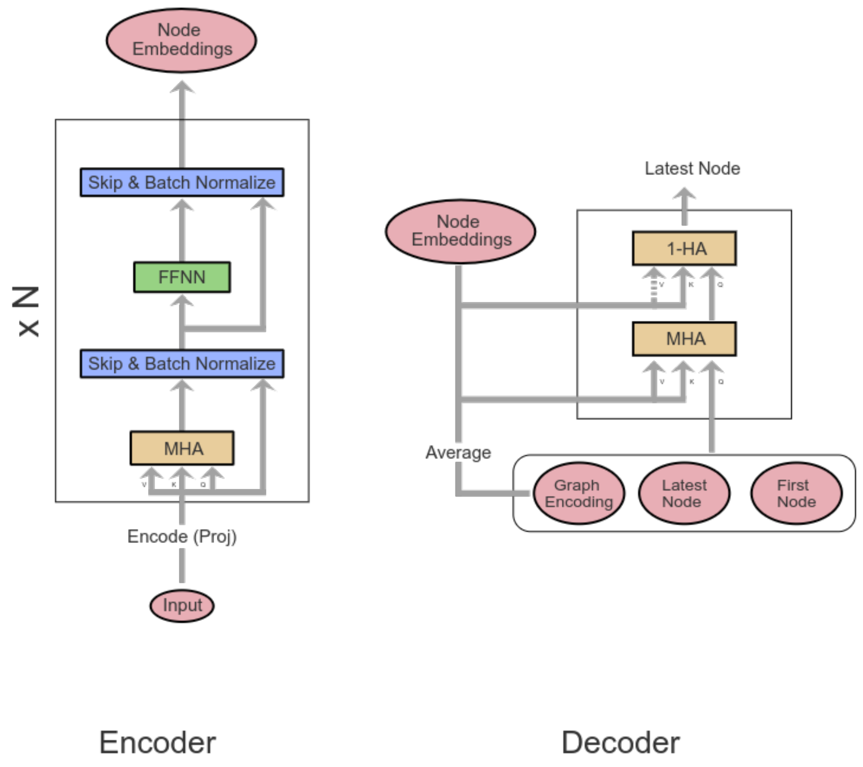


Figure 2.10: Encoder/Decoder architecture used in Attention Model.

node embeddings to contain information with respect to the rest of the graph. The output from the encoder is thus a list of nodes embedded into a different latent space, using attention to serve as the message-passing protocol as described in GNNs.

Input Embedding

The first significant difference between the transformer and the attention model is the absence of a positional encoder. In the Transformer, the positional encoder introduced linear relationships among the words in a sentence to allow the model to operate over the order of the words in the sentence. In the case of combinatorial problems, the order in which graph nodes are processed is irrelevant.

The first step in processing is similar to the transformer in that all the graph nodes are linearly projected into a latent space. Recall that the input problem is represented by a graph $G = \{x_i \in \mathbb{R}^{d_x}\}$. Each node is of dimensions d_x, which in this case is $d_x = 2$ (2D Euclidean). Using the following linear projection, the input is embedded into a vector of dimension d_h.

$$\vec{h}_i^{(0)} = \mathbf{W}^X \vec{x}_i + \vec{b}^x \tag{2.5}$$

such that $\mathbf{W}^X$ are the embedding parameters of dimension $d_h \times d_x$; $\vec{x}_i$ is a column vector from the input matrix of shape $d_x \times 1$; $\vec{b}^x$ is an additional parameter that represents some bias in the embedding of shape $d_h \times 1$; $\vec{h}_i^{(0)}$ is the resulting embedding of shape $d_h \times 1$. This embedding is then fed into the encoder module, where the latent space dimensions will be retained in the output from the encoder.

Attention Modules

The initial embedded nodes first enter the encoder through a Multi-Headed Self-Attention mechanism of 8 heads. Inside the layer, the module computes the *query*, *key*, and *value* vectors to use in computing attention

$$\begin{aligned} q_i^\ell &= \mathbf{W}^Q \cdot \vec{h}_i^d \\ k_i^\ell &= \mathbf{W}^K \cdot \vec{h}_i^d \\ v_i^\ell &= \mathbf{W}^V \cdot \vec{h}_i^d \end{aligned} \tag{2.6}$$

where q_i^ℓ is the query of shape $d_k \times 1$; k_i^ℓ is the key of shape $d_k \times 1$; v_i^ℓ is the value of shape $d_v \times 1$. Because this attention uses self-attention, query, key, and value computations all use the input graph nodes. W^Q is the projection matrix used in calculating the query and has shape $d_k \times d_h$; W^K is the projection matrix of the key and has shapes $d_k \times 1$; W^V is the projection vector of the values and has shapes $d_v \times 1$. Note that the shapes of the query and key vectors are the same. This is necessary for conducting the "query-key comparison."

The query and key vectors compute the weights of each node's message values. First, the compatibility scores $u^{\ell}_{i,j}$ are calculated

$$u^{\ell}_{i,j} = \frac{q_i^{\top} k_j}{\sqrt{d_t}} \tag{2.7}$$

where $q_{(c)}^{\top}$, k_j, and d_t are as described in equation 2.6. This is similar to the Transformer's use of dot-product attention.

Next, the compatibility scores are passed through a softmax layer to produce the weights for each value. These *attention weights* represent the similarity between the information searched for and the similarity of each key/value pair with it

$$a_{i,j} = \frac{e^{u_{i,j}}}{\sum_k e^{u_{i,k}}} = softmax(u_{i,j}, \mathbf{u}) \tag{2.8}$$

where $a_{i,j}$ is the attention similarity weight (a real number) for the compatibility score between i^{th} query and j^{th} key; $\mathbf{u}$ is the matrix containing each comparison weight between all queries and all keys.

Finally, the weighted values are computed and summed up to produce the final node embeddings (or final state if seen from the message-passing perspective)

$$h'_i = \sum_j a_{i,j} \cdot v_j \tag{2.9}$$

the resultant h'_i has dimensions $d_v \times 1$ and is the final node embedding produced by one head. In the case where we have multiple heads, we denote the message as $h'_{i,m}$, being the message produced by the m^{th} head. The respective attention head embeddings (or node states) are combined into one single final embedding by applying a linear projection and summing the resultant

$$\sum_{m=1}^{M} \mathbf{W}_m^O \cdot h'_{i,m} = \mathrm{MHA}_i(h_1, ..., h_n). \tag{2.10}$$

Note that $\mathbf{W}_m^O$ is a matrix of shape $d_h \times d_v$ that essentially maps each head message back to a d_h vector. This matrix is learned and can be seen as the importance of each head in the context of the final produced message.

Residual (Skip) Connection and Batch Normalization

The output vector message of shape $d_h \times 1$ is summed with the input used by the attention module. This is often done to maintain certain information from the raw input and speed up learning (since training begins at identity function). Batch normalization then avoids gradient explosion and speeds up learning

$$\hat{h}_i^\ell = BN^\ell(h_i^{(\ell-1)} + \mathrm{MHA}_i^\ell(h_1^{\ell-1}, ..., h_n^{\ell-1})) \tag{2.11}$$

$$\mathrm{BN}(h_i) = \mathbf{w}^{bn} \odot \overline{BN}(h_i) + b^{bn} \tag{2.12}$$

where $\hat{h}_i^\ell$ maintains its dimensions with respect to the output of the attention layer and residual layer ($d_h \times 1$). Note that a special implementation of batch normalization is used. First, regular batch normalization is computed. Then, an affine transformation is applied using a d_h vector, where the element-wise product is applied. The shape of the input is maintained as a result.

Feed-Forward Layer

A feed-forward neural network is used for computing the final embedding of the N^{th} encoder module. $\hat{h}_i^\ell$ is passed through a fully connected neural network of d_h input and output neurons and one hidden layer of 512 neurons and ReLU activation

$$FF(\hat{h}_i^\ell) = \mathbf{W}^{ff,1} \cdot \text{ReLU}(\mathbf{W}^{ff,0} \cdot \hat{h}_i^\ell + b^{ff,0}) + b^{ff,1} \tag{2.13}$$

where $\mathbf{W}^{ff,0}$ and $\mathbf{W}^{ff,1}$ are the synaptic weights for the input-hidden layer and hidden-output layer with respective bias vectors $b^{ff,0}$ and $b^{ff,1}$. It is important to note that the activation of the hidden layer differs from the activation of the output layer (ReLU and none, respectively). Finally, a residual connection is added to the output of the neural network, and the sum is batch-normalized (in similitude to the output of the attention layer).

2.7.2 Encoder Output

After the embeddings have been passed through all N encoder modules, the final node embeddings are expected to contain information on the individual nodes combined with the information of other nodes dependent on the structure of the input graph. These node embeddings are used by the decoder (as well as other information pertaining to the state of the problem as the model solves it) to predict the solutions to the input problem.

An additional piece of information computed from the node embeddings produced is the *graph embedding.* The graph embedding is defined in the Attention Model as

$$h_{(g)}^{(N)} = \overline{h}^{(N)} = \frac{1}{n}\sum_{i=1}^{n} h_1^\ell. \tag{2.14}$$

This establishes that the graph embedding will be the average of all graph node embeddings, retaining the dimension $d_h \times 1$ (same as node embeddings). The intuition behind this piece of information is that by averaging the graph node embeddings, we obtain a latent representation of the structure of the input graph. This graph embedding is concatenated along with state information on the problem (e.g, initial and current selected node in TSP) during decoding.

2.7.3 Decoder

The decoder significantly differs from the Transformer architecture in that only one module is used (rather than N modules), no batch normalization or residual connections are used, and no feed-forward network layers are used.

Context Embedding

After producing the graph node embeddings and the graph embedding (average of node embeddings), a problem context is created, which contains crucial information for the algorithm to know the state of the problem. The context encoder uses the graph embedding alongside the embedding of the first and last nodes produced by the decoder in the case of TSP. At the beginning of the decoding operation, a random start node is used (which is reused as the last node)

$$h_{(c)}^{(N)} = \begin{cases} \{h_{(g)}^{(N)}, h_{\pi_{t-1}}^{(N)}, h_{\pi_1}^{(N)}\} & t > 1 \\ \{h_{(g)}^{(N)}, h_{\pi_1}^{(N)}, h_{\pi_1}^{(N)}\} & t = 1 \end{cases} \tag{2.15}$$

where the operation $\{.,.,.\}$ represents the concatenation operation. The shape of the context vector is $3 \cdot d_h \times 1$.

Multi-Headed Attention

A multi-headed masked attention module is used where the node annotations serve as key and value while the context vector $h_{(c)}^{(N)}$ is used as a query. The process of computing the query, key, and values is similar to 2.6, where instead of using $\vec{h}_i^d$, we use $h_{(c)}^{(N)}$ for query ($\mathbf{W}^Q$ has dimensions $d_q \times 3 \cdot d_h$) and $h_1^{(N)}, ..., h_n^{(N)}$ for key and value

$$\begin{aligned} q_{(c)} &= \mathbf{W}^Q \cdot h_{(c)}^{(N)} \\ k_i &= \mathbf{W}^K \cdot h_i \\ v_i &= \mathbf{W}^V \cdot h_i. \end{aligned} \tag{2.16}$$

Then, a mask is applied to any node that cannot be visited. For example, in TSP, nodes that have been visited are no longer available. The similarity scores are computed similar to 2.7:

$$u_{(c),j}^{\ell} = \begin{cases} \frac{q_{(c)}^{\top} k_j}{\sqrt{d_t}} & \text{if } j \neq \pi_{t'} \ \forall \ t_0 < t \\ -\infty & \text{otherwise.} \end{cases} \tag{2.17}$$

Finally, the attention weights are computed to obtain the messages for each head as in equation 2.8 and 2.9 and the heads are combined following equation 2.10. Kool et al. refer to this result to be a "glimpse," producing vector $h_{(c)}^{(N+1)}$.

Single-Headed Attention

Once $h_{(c)}^{(N+1)}$ has been computed, a single-headed attention module is (partially) used. First, we reuse equation 2.16 for query, key, and values. Then, we modify eq. 2.17 to "clip" the similarity scores (now interpreted as logits)

$$u^{\ell}_{(c),j} = \begin{cases} C \cdot \tanh(\frac{q_{(c)}^{\top} k_j}{\sqrt{d_t}}) & \text{if } j \neq \pi_{t'} \ \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{2.18}$$

Finally, we compute the attention weights following equation 2.8. We define the attention weights to be $p_i = p_\theta(\pi_t = i | s, \pi_{1:t-1})$, the conditional probability of the next node to produce is i. The produced node will then be used as the last visited node.

2.7.4 Reinforcement Learning

As previously mentioned, the Attention Model is trained using reinforcement learning (RL). The reason for using RL is that combinatorial optimization problems are, as mentioned, computationally intractable. Because of this, traditional supervised learning may be hard to accomplish in large problems. To circumvent this problem, RL allows the use of unlabeled data, using a performance signal to guide learning. This signal can be computed from the function used to establish fitness among problem instances. In the case of TSP, this function is the Euclidean distance of the selected path.

Theory

More precisely, the REINFORCE algorithm with baseline is used for training the Attention Model in Kool et al. [2019], Peng et al. [2020]. Given that the attention model produces a distribution over the actions to take, and the model is a continuous, differentiable function, we can approach training using a policy gradient, approximating Monte Carlo methods.

Within Reinforcement Learning, policy gradient algorithms operate similarly to backpropagation: gradients of reward with respect to model parameters are computed to change the model [Williams, 1992]. These methods are desirable given their better convergence properties over other reinforcement learning algorithms, and the use of neural networks within these methods (as well as the existence of fast neural network derivation libraries) facilitates implementation [Sutton and Barto, 2018].

First, we denote by $\mathcal{L}(\theta|s)$ to be the loss function by which we will optimize our model, defined to maximize reward. As for the reward, we model the collected rewards using the function $R(s_{t-1}, a, s_t)$, which defines the reward collected when the environment is in state s_{t-1} and action a is taken, transitioning the environment to s_t. We derive the gradient of the loss function as

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\theta|s) &= \nabla_\theta \sum_a R(s_0, a, s) \cdot \pi_\theta(s_0, a) \\
&= \sum_a R(s_0, a, s) \cdot \nabla_\theta \pi_\theta(s_0, a) \\
&= \sum_a R(s_0, a, s) \cdot \pi_\theta(s_0, a) \cdot \frac{\nabla_\theta \pi_\theta(s_0, a)}{\pi_\theta(s_0, a)} \\
&= \mathbb{E}\left[R(s_0, a, s) \cdot \frac{\nabla_\theta \pi_\theta(s_0, a)}{\pi_\theta(s_0, a)}\right] \\
&\approx \frac{1}{N} \sum_{j=1}^{N} R(s_0, a, s) \cdot \frac{\nabla_\theta \pi_\theta(s_0, a)}{\pi_\theta(s_0, a)} \\
&= \frac{1}{N} \sum_{j=1}^{N} R(s_0, a, s) \cdot \nabla_\theta \ln(\pi_\theta(s, a_j))
\end{aligned} \tag{2.19}$$

where $L(\pi_\theta)$ is the loss incurred by policy π, parametrized by θ; $\pi_\theta(s, a)$ is the probability of the policy parametrized by θ from taking action a when the environment is at state s. Following the derivation, we obtain the following loss function equation

$$\nabla \mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[L(\pi)\nabla log_e p_\theta(\pi|s)] \tag{2.20}$$

where $p_\theta(\pi|s)$ is the probability of trajectory π to be produced for the combinatorial problem s. Notice that an expected value defines the gradient of the loss. Because of this, Monte Carlo methods are used to approximate this gradient.

Baselines

Due to the nature of Monte Carlo sampling methods, we can expect the expected value of the gradient to have a high variance [Sutton and Barto, 2018]. This occurs because different environments may give significantly different rewards. To address this, Williams [1992] introduced the idea of baselines where a function dependent on each presented state is used to ”even out” these differences:

$$\nabla \mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[(L(\pi) - b(s))\nabla log_e p_\theta(\pi|s)] \tag{2.21}$$

Where $b(s)$ is our baseline function. A good intuition about how the baseline affects learning is that the baseline acts as a function that scores what an expected reward could be. This may be similar to ”grading” the difficulty of a given instance problem. In doing this, it can be easier for the learning algorithm to differentiate gain in performance through as changes occur.

Kool et al. [2019] propose using three types of baselines: exponential, critic, and rollout, of which rollout is favored given its faster convergence. The exponential baseline utilizes the loss function $L(\pi)$ obtained during the first iteration of learning. Then, for every successive learning iteration, the baseline loss is modified as $\beta L(\pi) + (1 - \beta) L(\pi)$, using the latest loss inquired. Critic baseline utilizes a value function $\hat{v}(s, w)$, which can be modeled with a neural network to produce the expected reward on problem s (that being the expected future reward to expect when transitioning to state s). Finally, rollout utilizes the attention model learned so far to obtain the reward collected on the problem s greedily (always choosing the most likely path π). Doing this helps gain a more accurate estimate of how complex a problem is concerning the attention model. The baseline is then "rolled out" when the improved model surpasses the model statistically.

Algorithm 1 REINFORCE with Rollout Baseline

```
1: input: number of epochs $E$, steps per epoch $T$, batch size $B$, significance $\alpha$
2: Init $\theta$, $\theta^{BL} \leftarrow \theta$
3: for epoch = 1, ..., $E$ do
4:     for step = 1, ..., $T$ do
5:         $s_i \leftarrow$ RandomInstance() $\forall\, i \in \{1, ..., B\}$
6:         $\pi_i \leftarrow$ SampleRollout($s_i, p_\theta$)
7:         $\pi_i^{BL} \leftarrow$ GreedyRollout($s_i, p_{\theta^{BL}}$)
8:         $\nabla\mathcal{L} \leftarrow \sum_{i=1}^{B}(L(\pi) - L(\pi_{\theta^{BL}}))\nabla_\theta log_e p_\theta(\pi_i)$
9:         $\theta \leftarrow$ Adam($\theta, \nabla\mathcal{L}$)
10:    end for
11:    if OneSidePairedTTest($p_\theta, p_{\theta^{BL}}$) $< \alpha$ then
12:        $\theta^{BL} \leftarrow \theta$
13:    end if
14: end for
```

2.7.5 The Dynamic Attention Model (AM-D)

Following the work of Kool et al. [2019] on the attention model, Peng et al. [2020] presents an improved Attention Model that removes illegal nodes directly from the problem graph and re-encodes the nodes and graph to maintain "fresh" information. Peng et al. [2020] modified the encoder-decoder inference algorithm to re-encode the graph and nodes dynamically. This change aims to allow the model to better re-focus itself as an input problem becomes partially solved. The *dynamic attention model* (AM-D) was shown to surpass the classical Attention Model (AM) in the Vehicle Routing Problem. Additionally, AM-D achieved competitive results on problems with many nodes by training with only a fraction of the number of nodes in the problem.[2]

Differences to Attention Model (AM)

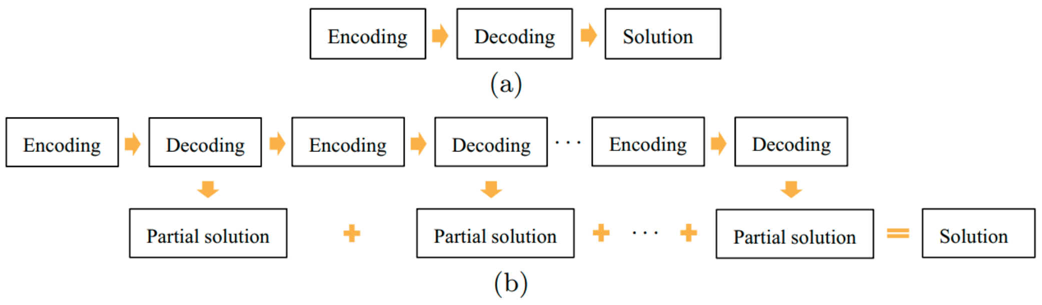


Figure 2.11: Figure from [Peng et al., 2020]. Comparison between AM (a) and AM-D (b). Note that the biggest difference is that a single pass through the encoder and into decoding is done by AM (a). In comparison, AM-D (b) re-encodes and decodes to produce partial solutions added at the end to produce the final trajectory. During re-encoding, illegal nodes are removed, thus reducing the problem's complexity as progress is made.

Most of the architecture presented in Kool et al. [2019] is kept in Peng et al. [2020]; the main difference is the encoder-decoder information flow (Fig. 2.11). During inference, the AM-D architecture encodes each node to produce node embeddings, later used to produce a graph embedding by averaging them out. Then, the graph embedding is concatenated with the currently produced node embedding (from the decoder) and the relevant problem state information. This context vector is then passed through the decoder to produce the next node.

[2]Kool et al. [2019] nor Peng et al. [2020] do not present results to the generalization scores on the vanilla attention model.

Then, the entire graph is re-encoded once the problem reaches a state of partial completeness. An example of an event triggering re-encoding could be the arrival at a goal node from a list of goals to reach during a plane's trajectory. The encoding process will refresh and mask any nodes whose state has changed or have become illegal nodes to traverse (like in TSP, where nodes are only allowed to be visited once). Training occurs using the same reinforcement learning Algorithm presented in algorithm 1.

2.8 Topological Machine Learning and Persistence

A very promising subfield of machine learning that could be useful in solving combinatorial optimization problems is Topological Machine Learning (TML) [Pun et al., 2018]. This subfield of machine learning developed independently from graph neural networks, and research attempted to allow machine learning and neural network models to access information through mathematical analysis of input problems. To perform this analysis, researchers use algebraic topology to convert input data into a format that allows analysis. The flagship tool used from topology is *persistence homology*, where signatures are extracted from the input data, which would in turn allow machine learning to distinguish among input data structures. The use of persistence homology in machine learning is called persistent-homology machine learning (PHML) [Pun et al., 2018].

This section will be separated into several subsections that will build understanding and intuition of the field of topology and topological data analysis, with a particular focus on the concepts of persistent homology. Finally, the challenges of using these tools with machine learning will be introduced, followed by some solutions presented in the literature.

2.8.1 Algebraic Topology

The field of algebraic topology focuses on studying the general shape of objects and what unique characteristics remain as deformations to the object are applied [Carlsson and Vejdemo-Johansson, 2021]. Topology differs from geometry because topology does not perform analysis over the structure defined by some metric (e.g., Euclidean distance). Instead, topology studies the structure of objects through the observation that specific characteristics (e.g., holes or sharp corners) are retained by objects when continuous, linear transformations are applied.

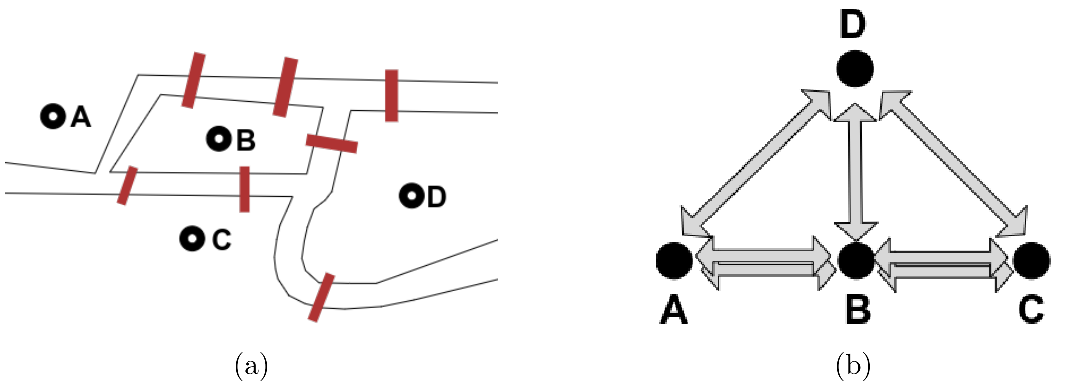


Figure 2.12: Visualization of the seven bridges of Königsberg. (a) Is the original form of the problem. (b) Is the graph form of the problem. Note that in (b), we must traverse from D to/from A, B, and C once, and between A to/from B, and B to/from C twice. No direct bridge exists between A and C.

A good example that provides intuition and motivation for the study of topology is the so-called "Bridges of Königsberg." This problem, which Euler first studied, asks whether a person can travel through all bridges connecting four sections of the city of Königsberg exactly once and ending at the starting point.

This problem can be observed in figure 2.12a. Euler realized that this problem could be re-formulated in infinite ways by keeping certain characteristics consistent, namely the number of bridges and the land they connect. This idea can be pictured in figure 2.12b, where the bridges are represented as legs of a graph connecting four nodes (where direct paths exist among all nodes except between A and C.) All of these representations, although visually different, represent the same original problem where deformation has been applied. Essentially, these problems all belong to the same topological class. Through this perspective, Euler proved no solution exists.

2.8.2 Topological Data Analysis

While Topology is a mature field of mathematics, its application to data has received attention only recently. Since the inception of Topology, much analysis has been reserved for the elemental study of the characteristics of important shapes such as tori and spheres. As computational techniques for topology began to be developed, researchers realized that real-world data could undergo similar analysis by modeling data points (a.k.a. point cloud data) as topological structures. Then, topological analysis tools could be used to analyze these topological structures [Carlsson and Vejdemo-Johansson, 2021].

Simplices and Simplicial Complexes

To model point cloud data as a topological space, the notion of simplices is used to model the atomic units of the data. A *standard n-simplex* is written as

$\Delta^n \subseteq \mathbb{R}^{n+1}$ and it is defined by the set:

$$\{(x_0, x_1, ..., x_n) | \sum_i x_i = 1 \text{ and } x_i \geq 0 \;\; \forall\, i\} \tag{2.22}$$

Where each x_i element is a point of some dimensionality. Note that a 0-simplex represents a single point, a 1-simplex represents a line, a 2-simplex represents a triangle, a 3-simplex represents a tetrahedron, and so on. An example of geometrical representation can be seen in figure 2.13

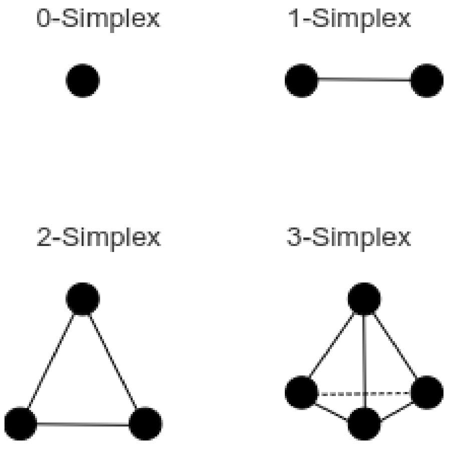


Figure 2.13: Geometrical visualization of different simplices.

In topological data analysis, points from point-cloud data are represented as 0-simplices. Combining all the point-cloud data creates a *simplicial complex.* Note that such a simplicial complex has no connected elements; thus, no higher n-simplices. This can be a problem, as these n-simplices allow us to perform analysis on the point cloud. To solve this problem, several simplicial complex-building algorithms have been developed, each creating simplicial complexes.

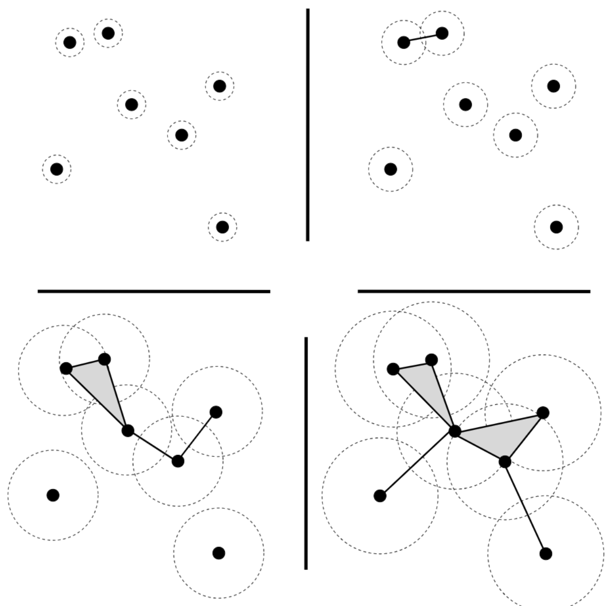


Figure 2.14: Example Čech complex construction at $\epsilon = 0.1$, $\epsilon = 0.2$, $\epsilon = 0.4$, and $\epsilon = 0.5$. As spheres connect, 1-simplices and 2-simplices are created. Note that for a 2-simplex to form, three spheres must connect (see bottom two figures.)

Čech Complex

Let the point cloud be represented by $Z \subset \mathbb{R}^n$. Given a value $\epsilon \in \mathbb{R}$, the Čech complex at scale ϵ is defined by some covering $\mathfrak{U}_{\epsilon}^{\text{Cech}}$ of Euclidean balls over each point with radius ϵ: $\{B(z, \epsilon)\}_{z \in Z}$. This complex is then constructed by finding the nerve of the covering $\mathfrak{U}_{\epsilon}^{\text{Cech}}$, which is defined as the set of non-empty collections $z_{i0} \cap ... \cap z_{is} = \emptyset$

Figure 2.14 demonstrates the complex construction process for some simple point clouds at different ϵ scales. Intuitively, at $\epsilon = 0.0$, the simplicial complex comprises 0-simplices (points). This remains true as we increase ϵ to 0.1 and 0.2. As shown in figure 2.14, when $\epsilon = 0.4$, several 1-simplex, and one 2-simplex are created, shown as lines and a triangle, respectively. As we increase the value for ϵ, more simplices are created.

The Čech complex has the theoretical guarantee that the complex formed by the nerve of the covering, as described previously, is homotopy-equivalent to the topological space it was formed from. This theoretical guarantee establishes the validity of using a formed Čech complex to perform topological analysis, safely assuming that the same topological characteristics of the original point-cloud data will remain intact. This theoretical guarantee is called the *Nerve theorem*, and it is proven by showing that for any covering, there exists a homotopic transformation (a transformation that only bends, shrinks or expands, never cutting) that can convert the complex created by the nerve back to the original topological space of the point-cloud data.

Vietoris-Rips Complex

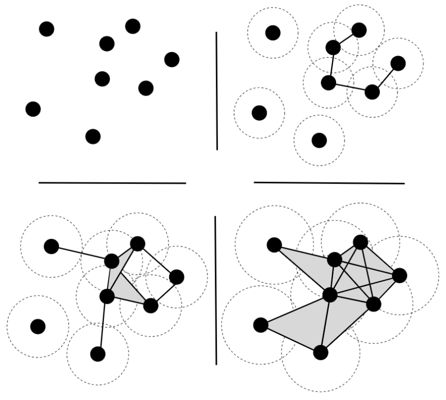


Figure 2.15: Example Rips construction. The values used for ϵ are 0.0, 0.1, 0.3, 0.5.

While the Čech Complex has many desirable theoretical guarantees, it can become computationally expensive to compute the simplices since their construction depends on verifying the non-emptiness of interceptions among spheres. To circumvent this problem, the Vietoris-Rips complex was developed to simplify computation.

The Vietoris-Rips complex on a point cloud Z at some scale ϵ, represented as $VR(X, \epsilon)$, is defined by the k-simplices spanned when:

$$d(z_i, z_j) \leq \epsilon \tag{2.23}$$

where d is some function belonging to a metric space.

The intuition behind the Vietoris-Rips complex, whose construction can be exemplified in figure 2.15, is similar to the construction of the Čech complex. A group of ϵ radius spheres are placed on top of the 0-simplices. Then, a new simplex is formed when the distance between two spheres is less than ϵ. Because distance is used, computational complexity is decreased. Note that Rips complexes do tend to have higher simplex counts.

The main drawback behind Rips complexes is the loss of theoretical guarantees. Specifically, the Nerve theorem does not always hold for Rips complexes. Despite this, theoretical guarantees prove that Čech complexes can be contained within Rips complexes. More exactly, there exists an inclusion as follows:

$$\mathrm{VR}(X, \epsilon/2) \subset C^{\check{\mathrm{C}}\mathrm{ech}}(X, \epsilon) \tag{2.24}$$

While several more complex schemes exist, only Vietoris-Rips and Čech complexes are presented since these are most relevant to this work. Any interested readers seeking to learn more about other complexes are referred to the excellent books by Edelsbrunner and Harer [2010], and Dey and Wang [2022].

2.8.3 Homology

Given a built topological space (either Čech, Vietoris-Rips, or any other), homology is the most common type of topological analysis performed. The core idea behind homology is that topological spaces can be differentiated by looking at the multi-dimensional connected components (informally, holes) these spaces own. Computation of homology requires the definition of a *chain complex*, which essentially encodes the information of the topological space X. Then, homology groups can be formed by looking at the image and kernel of functions defined over the chain complex.

Cycles

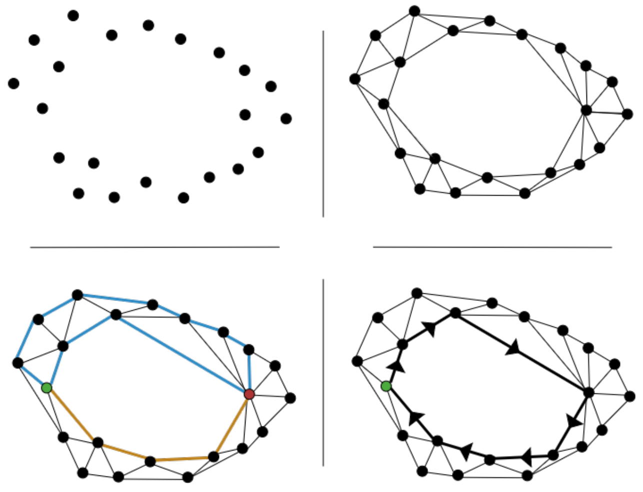


Figure 2.16: Example topological space where cycles are distinguished.

Since homology is interested in the analysis of cycles, it is essential to understand how these concepts operate within the topological space. Consider figure 2.16, where a topological space is constructed from the given data point cloud. It is easy to distinguish the existence of a hole in the center of the space, but defining such a characteristic can be challenging. For example, note in the bottom left figure how two blue segments can represent a segment of the hole. Many more possible segments can represent essentially the same segment of a cycle. For this reason, defining these segments as part of a family or group of segments can be helpful. We achieve this by defining such a family of segments by some mapping that can convert any given segment into any other segment belonging to the family.

Note that while all segments on the upper side of the space connecting the green and red points belong to the same family, any segment from the lower side of the space would not. The brown segment cannot be transformed to fit any of the segments at the top of the shape, including the blue segments, because such a mapping would require the segment to be broken. This raises the notion of direction in segments, where the blue segments have a different direction than the brown segment. This idea can be visualized by the cycle in the bottom right in figure 2.16, where a clockwise cycle can be observed. If one were to create a cycle starting and ending at the green node that is counter-clockwise, it would be easy to see that these cycles would not belong to the same cycle family.

Chain Complexes and Homology

Having defined cycle groups, which can have different dimensionality, chain complexes are used to extract topological information for analysis. A chain complex is defined to operate over some set of abelian groups of modules, connected by a set of maps connecting these groups

$$... \xrightarrow{\partial_{i+1}} C_i \xrightarrow{\partial_i} ... \xrightarrow{\partial_2} C_1 \xrightarrow{\partial_1} C_0 \xrightarrow{\partial_0} = 0 \tag{2.25}$$

These maps ∂_i are referred to as boundary maps and have the special property that $\partial_{i-1} \circ \partial_i = 0$. Within this definition of chain complexes, we can compute the groups of cycles as

$$Z_p = \{c \in C_p : \partial_p(c) = 0\} = Ker(\partial_p). \tag{2.26}$$

Chains also have attached to them a way to compute the boundary objects in the topological space. These can be computed as

$$B_p = \{c \in C_p : (\exists b \in C_p + 1 : \partial_{i+1} = c)\} = Im(\partial_{n+1}). \tag{2.27}$$

Using these definitions of cycles and boundaries, it is possible to compute n^{th} *homology group* of the space, which describes the number of n-dimensional holes that exist within the topological space. Homology is defined as:

$$H_n = Ker(\partial_n)/Im(\partial_{n+q}) = Z_n(\Sigma)/B_n(\Sigma) \tag{2.28}$$

Where Σ is the topological space.

Homology is crucial for topological analysis as this tool permits us to quantify the different-dimensional holes that exist within a topological space. Homology essentially serves as a kind of signature unique for families of topological spaces that share certain characteristics in common.

2.8.4 Persistent Homology

As shown in figure 2.15, different choices of ϵ affect the simplicial complex's characteristics. For example, when $\epsilon = 0.3$, a hole can be appreciated, disappearing later as it gets filled up at $\epsilon = 0.5$. Since homology pays attention to these types of characteristics, this raises the question of how we can choose the right ϵ that is relevant to our analysis. The answer to this problem is to analyze the change in topological characteristics, namely the change in the homology of the space, as ϵ is modified. This idea gives birth to *Persistent Homology.*

First, the filtration concept formalizes that different topological simplicial complexes are generated as ϵ changes. With Z being the topological space over which analysis is to be performed, filtration is defined as

$$\emptyset = Z^0 \subseteq Z^1 \subseteq ... \subseteq Z. \tag{2.29}$$

One of the key uses of filtration is in analyzing changes in homology. Specifically, by studying the homology of topological spaces through filtration, it is possible to capture a sense of persistence in certain topological features. A significant hypothesis within the field is that capturing this information is unique to families of objects and can give rise to a *topological signature.* This analysis is called *persistent homology.*

The main technique used for creating topological signatures from filtration is the *Persistence diagram* (PD). The PD can be created by computing homology through a given filtration, keeping track of the times at which a particular n-dimensional hole appears and disappears. The time at which a hole appears is called its *birth time*, while the time it disappears is called it's *death time.* Plotting this information can provide a visual representation of the topological signature of a topological space.

Figure 2.17 gives a visual example of the computation of a persistence diagram. First, filtration is created through the construction of a Vietoris-Rips

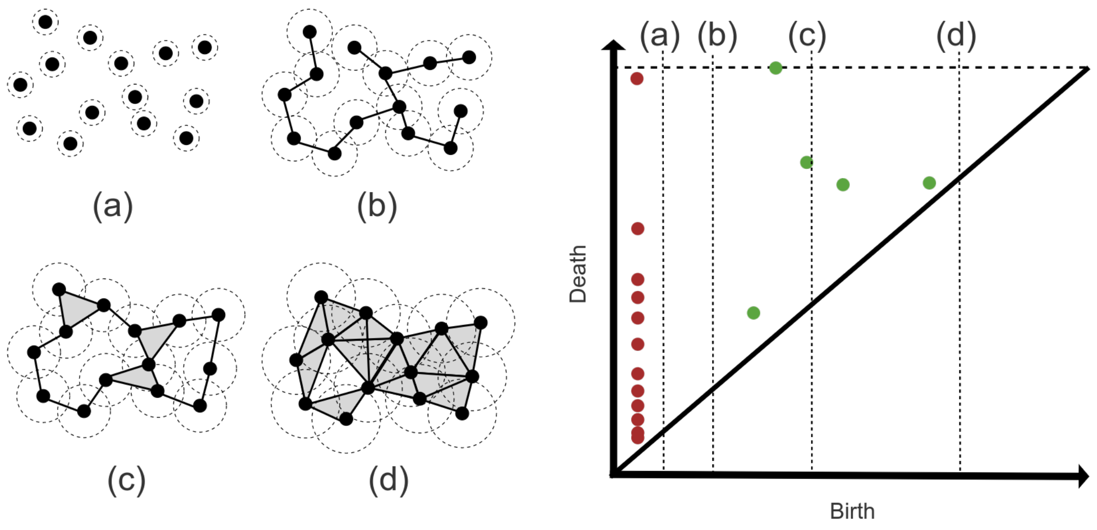


Figure 2.17: Example computation of a persistence diagram through persistent homology. Left figures visually represent the topological space at some filtration value. The right figure is a persistence diagram with some key moments marked by a) through d). Red dots in PD represent 0-cycles (nodes). Green dots represent 1-cycles (holes).

complex. Then, each hole's birth and death times are tracked. At filtration time (a), no holes exist other than nodes. Nodes are 0-cycles, which are represented in the PD as red dots. As the filtration continues, 1-simplices start appearing as legs connecting nodes in the topological space. By time (b), these 1-simplices have not produced any holes. Between time (b) and (c), 1-simplices connect to create holes represented by green dots. Additionally, some 2-simplices appear. Between time (c) and (d), creating more simplices closes one of the holes while some holes suddenly appear and disappear.

The persistence diagram in figure 2.17 is a topological signature. The PD also provides information on the persistence of holes. For example, note that the hole in the left side of the topological space of figure 2.17 persists for a long time as filtration occurs. This characteristic is represented in the diagram by the green dot that is high in the diagram. Dots that sit high in the diagram belong to holes that are most important to the topology of the topological space. Points that sit closer to the diagonal tend to be holes that disappear soon after birth and tend to be related to noise.

Chapter 3

Persistence Attention Model (AM-P)

Inspired by the recent progress of topological machine learning, it seems fitting to study the impact of Persistent Homology signatures within the field of heuristic combinatorial optimization. Because combinatorial optimization often deals with problems that can be presented in graph form, an intrinsic graph structure exists that could be useful for solving problems like Traveling Salesman. As a matter of fact, Persistence has been used for several years to solve problems related to graph labeling or network optimization Pun et al. [2018]. This suggests persistence could be a tool that could aid combinatorial optimization heuristic algorithms.

As presented in the background section, the Attention Model is the current state-of-the-art model capable of solving combinatorial optimization problems. The model uses an encoder-decoder architecture where the encoder takes in a graph problem. Then, through a series of computations by neural networks and self-attention layers, the encoder produces a new representation of the nodes in some latent dimension presumed to contain information about the graph's structure. Finally, the model uses its decoder using the newly-computed graph encoding to produce a probability distribution indicating the nodes with the highest likelihood to be in the optimal path. By obtaining new probability distributions as the problem becomes solved, the model produces a trajectory with the highest likelihood of being the optimal solution to the problem.

While it is known that GNN and GAT models tend to the graph's structure, it is unknown what kind of information is extracted. Understanding the nature of the information computed by these models could shed light to better ways to understand the problems these algorithms solve. Further, this knowledge could help in designing better models that could further close the gap between heuristic and exact algorithms.

3.1 Persistence Attention Model

This thesis proposes a variant of the Attention Model, the Persistence Attention Model (AM-P) (see figure 3.1). The model retains much of the architecture of the AM and AM-D models presented by Kool et al. [2019] and Peng et al. [2020] with the distinction that the graph embedding used by the decoder uses Persistence signatures to replace the average operation employed by the AM/AM-D models.

Including persistent homology information through graph encoding best fits the AM/AM-D architecture theory. The original models use graph encoding to retain global information on the graph's structure throughout decoding [Peng et al., 2020]. Since persistent homology provides a topological signature over the entire graph input, it fits within the context of the graph encoding.

The persistent signature layer comprises three sub-layers: persistence diagram computation, diagram vectorization and concatenation, and projection sublayers

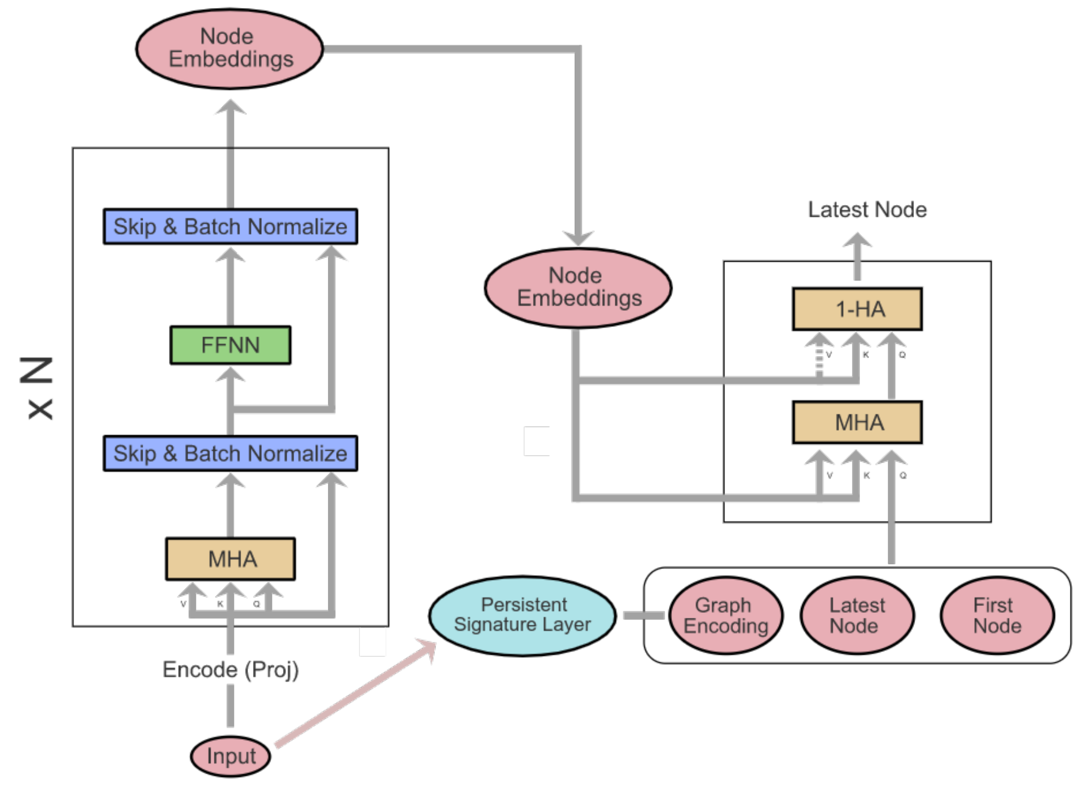


Figure 3.1: Proposed architecture. The main modification includes a *persistence signature layer*, which performs persistent homology analysis and vectorizes the signature to use in the decoder.

(see figure 3.2). In the persistence diagram computation layer, the input graph problem is converted to a topological space by creating a simplicial complex (e.g. Čech, Vietoris-Rips, etc). A filtration is then computed over the simplicial complex, and a *persistence diagram* over the first n-homologies is created.

While a persistence diagram is a topological signature, it cannot be directly passed to machine learning models because the data lack an algebraic structure. Specifically, persistence diagrams are not within a Hilbert or Banach space [Carrière et al., 2019]. This problem arises from the fact that many persistence diagrams can have a different number of points, and some vector space operations are not well-defined [Carrière et al., 2019]. For this reason, a vectorization

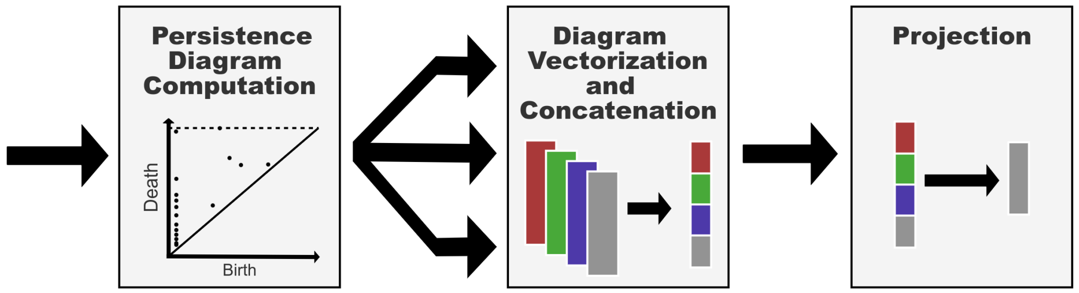

Persistence Diagram Computation
Death
Birth
Diagram Vectorization and Concatenation
Projection

Figure 3.2: Persistent homology signature computation and vectorization layer. The choice taken for the AM-P presented uses Vietoris-Rips complexes to perform persistent homology analysis. Then, Persistent Landscapes are used to vectorize the diagrams. Finally, the vectors are concatenated and projected to d_m.

operation is required to introduce this crucial structure for machine learning.

In the literature, there are different approaches to performing vectorization on persistence diagrams. Approaches can be divided into three branches: *direct vectorization*, *kernel methods*, and *learned vectorization*. Direct vectorization performs a series of transformations to map a persistence diagram into a Banach or Hilbert space. A kernel method utilizes the kernel trick to introduce the diagram information without the computational burden required by direct vectorization. Learned vectorization encompasses the use of machine learning to find a vectorization function.

Due to the sheer number of different approaches to vectorize persistence diagrams, it is left to future work on comparing these approaches with respect to the model performance. After comparing different vectorization methods, the tested implementation in this thesis makes use of a type of direct vectorization technique called *Persistence Landscapes* [Bubenik, 2015].

Persistence landscapes were introduced in Bubenik [2015] and utilize a series of mappings to rotate and obtain ”layers of influence” under the area created by a point in a persistence diagram (see figure 3.3). First, given a persistence

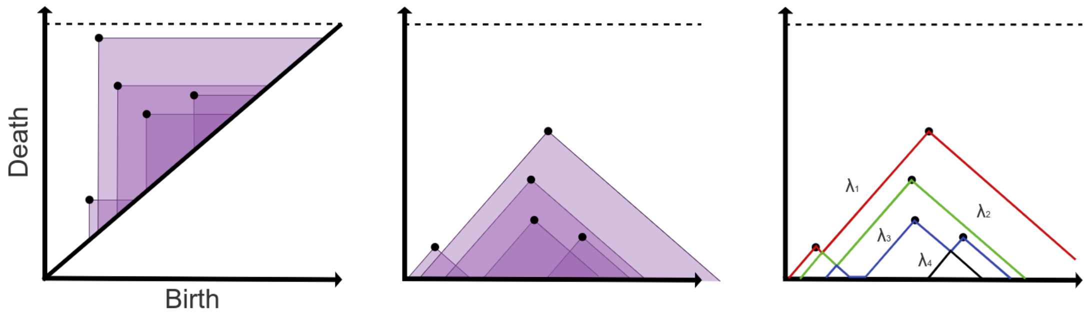


Figure 3.3: Visualization of the process for computing persistence landscapes. A mapping is applied to perform a clockwise rotation on a given persistence diagram. Then, persistence landscapes can be "peeled back using tent functions." In the rightmost figure, 4 landscapes can be appreciated. The vector representation would encompass n samples of each landscape.

diagram defined as $\mathcal{D} = \{(b_i, d_i)\}_{i \in I}$, an auxiliary function can be chosen, which is defined in Bubenik [2015] as $f_{(b,d)}(t) = max\{0, min\{t - b, d - t\}\}$. Then, the *k-th persistence landscape* is defined by the function:

$$\lambda(k, t) = \text{k-max}\{f_{(b_i,d_i)}(t)\}_{i \in I} \tag{3.1}$$

To obtain the vector form, the function is sub-sampled to create a vector of some specified dimension.

After obtaining a vector representation of the persistence diagram, the first k-persistence landscapes in vector form are concatenated and projected to d_m dimensions. This linear projection is intended to maintain a fixed vector length that can be concatenated with the rest of the problem context to run the decoder module.

The computation of the persistence diagrams and the persistence landscapes is done through the open-source library Gudhi [Maria et al., 2014]. While other possible libraries implement similar tools to carry out Topological Data Analysis, Gudhi was found to have a more robust repertoire that best fits the code implementation for AM-P. Additionally, the library's authors used code directly provided from the papers where persistence landscapes were presented.

The model is trained following the same REINFORCE with baseline algorithm presented in algorithm 1. To solve the Traveling Salesman problems, the reward function used was the *negative of the distance traveled.* This choice of reward function follows that chosen in the literature [Kool et al., 2019].

The implementation used for this thesis can be found in the linked github repository.

Chapter 4

Investigating Persistence and Graph Attention

4.1 Goals

The main question investigated through the presented experiments is how persistence homology relates to the attention model. While it is understood the mechanism by which the attention model extracts graph structural information (via GATs), it is unknown what kind of information is used. To learn more about the structure information AM uses, specifically whether information related to homology is extracted, a series of experiments test the performance of the AM with and without access to persistent homology information. It is hoped that by observing the change in performance by the AM we can gain a better understanding about whether GATs extract information similar to Persistent Homology.

4.2 Investigating Impact of Persistence as Graph-Wide Embedding

In the first set of experiments, the original Attention Model and the presented Persistence Attention Model are trained to solve the Traveling Salesman Problem over graph sizes of 10, 20, and 50 nodes. The models are trained with a similar architecture configuration, defined in table 4.1. Additionally, similar hyperparameters are used as those from [Peng et al., 2020], which are presented in table 4.2. Training occurs as defined in algorithm 2, which is a modification from 1 to include computation of validation costs.

Parameter	Value
Encoder Layers (N)	3
Attention Heads (h)	8
Neural Network Layer (d_{ff})	512
Model Latent Dimensions (d_m, d_k, d_v)	(128, 64, 64)
Training Samples	1,280,000
Learning Rate	$1e^{-4}$

Table 4.1: Model and training hyperparameters.

Parameter	Value
Epochs	30
Batch Size	512
Training Data	1,280,000 (per epoch)
Graph Nodes	10, 20, 50 (consistent over experiment)
Architecture	3 encoder layers
Learning Rate	$1e^{-4}$

Table 4.2: Experiment parameters based on Peng et al. [2020] paper.

Once results are collected, statistical testing is performed to see whether the performance and/or overall training behavior of the trained models is statistically similar or different. For this, the validation scores obtained over 10,000 TSP

Algorithm 2 Experiment #1's train function.

```
Require: Optimization function $\varphi$. Model $M$. Baseline $M_b$. Gradient clipping
    value $\rho$. Number of epochs $\varepsilon_{max}$. Baseline $M_b$. Datasets $\mathcal{D}_{train}$ and $\mathcal{D}_{validate}$
 1: Split $\mathcal{D}_{train}$ into $\varpi$ batches ($\mathcal{D}_{train}^{(\varpi)}$)
 2: Evaluate baseline $M_b$ on all batched data $\mathcal{D}_{train}^{(\varpi)}$. Store results into cost matrix
    $C_b^{(\varpi,i)}$, where $i$ is the ith sample from the $\varpi$ batch.
 3: for all batches $\varpi$ in $\mathcal{D}_{train}^{(\varpi)}$ do
 4:     Forepropagate $\mathcal{D}_{train}^{(\varpi)}$ in model $M$.
 5:     Calculate error and cost, and compute gradients using $C_b^{(\varpi,i)}$.
 6:     Clip gradients by global norm to $\rho$.
 7:     Use optimizer $\varphi$ to backpropagate gradients through model $M$.
 8:     Store training error ($\varepsilon_M^{(\varpi,i)}$) and cost ($C_M^{(\varpi,i)}$) for this batch.
 9: end for
10: Compute Costs on validation dataset $\mathcal{D}_{validate}$ by both $M$ and $M_b$, and store
    into $C_{M,val}$ and $C_{b,val}$
11: if p-value($C_{M,val}, C_{b,val}$) $< 0.05$ then
12:     $M_b \longleftarrow M$
13: end if
14: return $M, C_M$, $C_{M,val}$
```

problems of 10, 20, and 50 nodes each are collected.

The statistical tests used to compare the training curves follow those presented in Piater et al. [1998]. The statistical hypothesis testing investigates the existence of the *algorithm effect* and/or the *interaction effect.* The algorithm effect is defined as the situation where the performance of a training curve exhibits superior performance over other training curves. The interaction effect, on the other hand, describes the situation where performance achieved described in a learning curve is dependent on the algorithm used. Intuitively, the algorithm effect looks at statistical differences in the performance among learning curves, while the interaction effect looks at the statistical difference in the overall shape of training curves. A threshold p-value of 0.05 is used following standard statistical testing.

4.3 Investigating Generalization to Different Graph Problem Sizes

An exciting result presented in Peng et al. [2020] shows the ability of the AM to generalize across problem sizes. The authors showed that it is possible to train the AM with smaller problems and achieve good performance on problem sizes much larger than initially trained with. This opens the possibility to accelerate learning by re-using models to solve larger problems. We believe that part of the reason why the model could retain good relative performance on different problem sizes is its ability to use graph structural information.

To study this idea, an experiment is presented to compare the performance obtained by the AM and AM-P models as they are validated on larger or smaller Traveling Salesman Problems. The models are validated over 10,000 TSP problems of 10, 20, and 50 nodes each. The scores for each problem size are then statistically tested with a t-test to accept or reject the hypothesis that the AM-P model's performance and/or overall training behavior differs from that of the original AM model on any of the problem sizes that the tested problems were not originally trained on. Again, a threshold p-value of 0.05 is used following standard statistical testing.

Chapter 5

Results

5.1 Experiment 1

For experiment 1, 10 instances of the AM and AM-P models were trained on traveling salesman problems of sizes 10, 20, and 50. Each training session was run for 30 epochs following the hyperparameters described in table 4.1. During training, the average reward achieved by the models was collected for each epoch. To gain a more accurate representation of the average learning curve, 10 AM and AM-P models were trained for each of the problem sizes. The average learning curve over each experiment for each model can be appreciated in figure 5.1.

A statistical analysis was performed on the data collected. The experiment analysis described by Piater et al. [1998] is used for a randomized ANOVA analysis for statistical hypothesis testing. Two hypotheses are tested:

- H_o^{alg}: The mean performances of the attention model and Persistence attention model are the same (algorithm effect).

- H_o^{int}: The relationship between training time and performance does not depend on the algorithm (interaction effect).

The results from the ANOVA experiment can be seen in tables 5.1, 5.2, and 5.3:

	df	SS	MS	F	P
Algorithm	1	0.0006	0.0006	0.5282	0.4676
Interaction	29	1.1129	0.0038	0.9547	0.1319

Table 5.1: ANOVA table collected over AM and AM-P trained on TSP of 10 nodes.

	df	SS	MS	F	P
Algorithm	1	0.0133	0.0133	1.6288	0.2024
Interaction	29	0.8287	0.0028	0.6522	0.3198

Table 5.2: ANOVA table collected over AM and AM-P trained on TSP of 20 nodes.

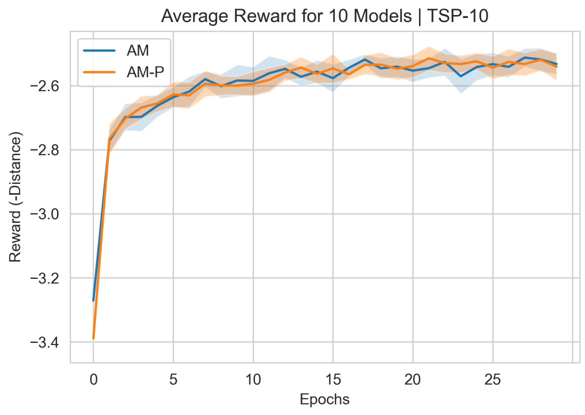
	df	SS	MS	F	P
Algorithm	1	0.0155	0.0155	3.5118	0.1147
Interaction	29	1.4296	0.1183	1.4398	0.0855

Table 5.3: ANOVA table collected over AM and AM-P trained on TSP of 50 nodes.

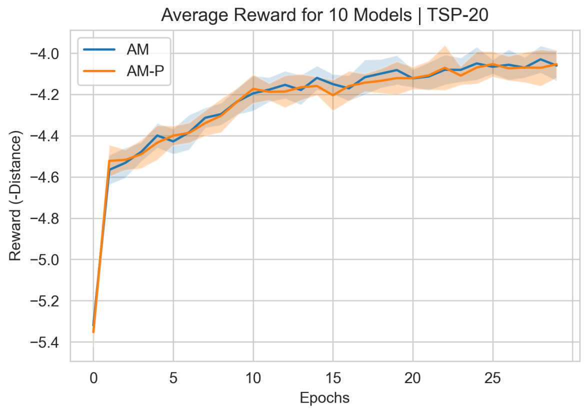
Additionally, to explore for similarities in the solutions chosen by the models, a random model (both AM and AM-P) trained on each problem size (10, 20, and 50) was chosen to undergo visualization. For this, a random problem was created, and the selected models were tasked with solving it. The trajectories generated can be appreciated in figure 5.2.

5.2 Experiment 2

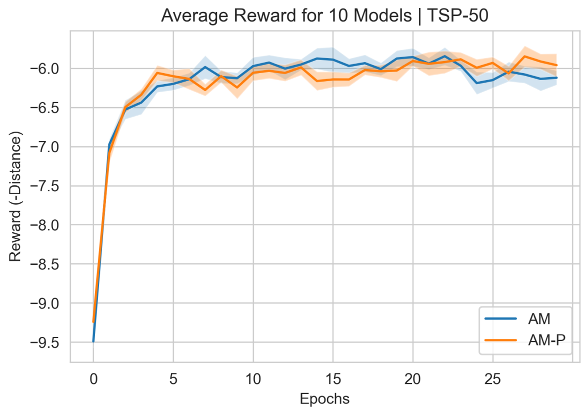
For experiment 2, the trained models were tested to compare how well the AM and AM-P models generalize to different problem sizes, and whether there are any



(a) Experiment on 10 nodes.



(b) Experiment on 20 nodes.



(c) Experiment on 50 nodes.

Figure 5.1: Average training rewards over 30 epochs. The shaded region shows the first standard deviation at each epoch.

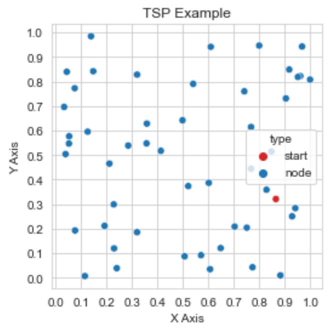
differences in performance. Similarly to the visualization of solutions presented in experiment #1, models trained on 10, 20, or 50 nodes were selected to undergo experimentation. These models were validated over 8,192 randomly-generated TSP problems on different sizes (10, 20, and 50 nodes), and their average reward and standard deviation was computed. Results can be seen in table 5.4. Instead of showing the reward values, the average distance traveled is presented to provide a more intuitive comparison of their performance. Note that the reward function is defined as the negative distance traveled.

Training Problem Size		Validation Problem Size		
		10	20	50
10	AM	2.44 ± 0.0397	3.92 ± 0.0434	6.49 ± 0.0493
	AM-P	2.43 ± 0.0410	3.93 ± 0.0433	6.48 ± 0.0503
20	AM	2.54 ± 0.0406	3.97 ± 0.0389	6.44 ± 0.0434
	AM-P	2.51 ± 0.0415	3.94 ± 0.0391	6.47 ± 0.0446
50	AM	2.86 ± 0.0511	4.16 ± 0.0486	6.02 ± 0.0494
	AM-P	2.85 ± 0.0515	4.11 ± 0.0492	6.05 ± 0.0488

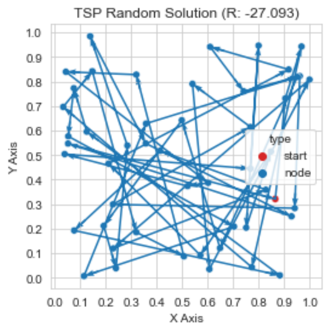
Table 5.4: Average distance traveled across different Traveling Salesman Problem sizes. Models were originally trained on a single size of the traveling salesman problem, and then they were validated to observe how well the model could generalize to other problem sizes. First standard deviations are included.

Validation Problem size	Training Problem Size		
	10	20	50
10	0.6321	0.5351	0.4421
20	0.3234	0.1523	0.2322
50	0.4191	0.2457	0.2000

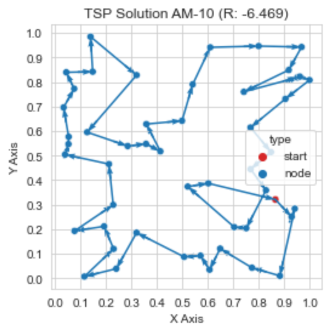
Table 5.5: P-values reported from t-test between the rewards collected between AM and AM-P trained on the same problem size.



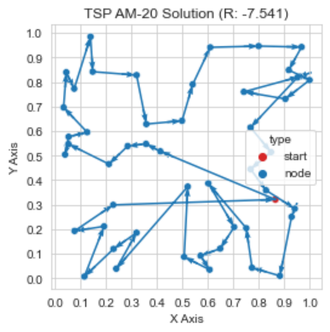
(a) Example validation TSP-50 problem.



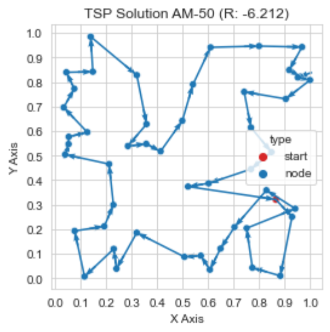
(b) Solution selected by a random model.



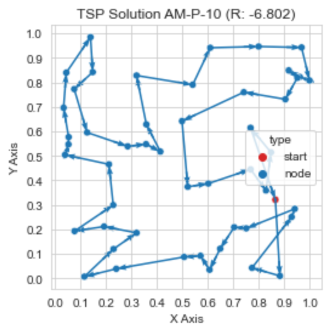
(c) AM trained on 10 nodes.



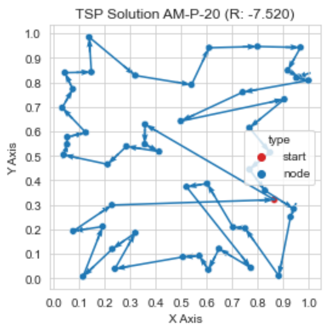
(d) AM trained on 20 nodes.



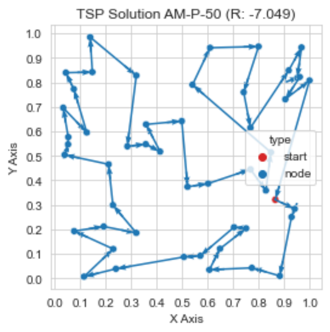
(e) AM trained on 50 nodes.



(f) AM-P trained on 10 nodes.



(g) AM-P trained on 20 nodes.



(h) AM-P trained on 50 nodes.

Figure 5.2: Solutions to Traveling Salesman Problem of 50 nodes chosen by different models.

Chapter 6

Discussion & Conclusions

The results observed in experiment 1 seem to contradict the hypothesis that introducing persistent homology will improve the performance of the attention model. Figure 5.1 presents the behavior of the average learning curve while 10 models are being trained. In the three figures, the average plot over 10 train models (for each model type and problem size) are close together and seem to converge towards a similar value. This behavior is a bit different in figure 5.1c, where the average lines seem to be more random, yet the lines still seem to converge to a close value.

Statistical hypothesis testing was performed to support the aforementioned observations that no statistically significant difference exists in the mean training curves collected. A *randomized ANOVA* as described by Piater et al. [1998] is used for testing. The following hypotheses are tested:

- H_o^{alg}: The mean performances of the Attention Model and Persistence Attention Model are the same. Also called the algorithm null hypothesis.

- H_o^{int}: The relationship between training time and performance does not depend on the algorithm.

The null hypotheses state that the difference between the two algorithms with respect to performance is statistically insignificant with respect to the difference in the means (algorithm effect) and the behavior of the curves caused by training throughout time (interaction effect). We use a threshold p-value of 0.05 for our hypothesis testing.

The resulting statistics from the analysis can be appreciated in tables 5.1, 5.2, and 5.3. The p-values reported for the algorithm effect hypothesis (H_o^{alg}) suggest an inability to reject the algorithm null hypothesis due to lack of evidence; thus, we accept this hypothesis. It is important to note that this only proves that the difference between the training curves was not statistically significant.

With respect to the interaction hypothesis (H_o^{int}), the p-values reported again suggest an inability to reject the null hypotheses due to a lack of evidence. An interesting result can be observed in table 5.3, where the p-value was rather low. It can be observed in figure 5.1c how both training curves do differ but follow a general direction in learning.

As problem size increases, the p-values seem to suggest that the statistical difference between both algorithms increases. While we do not make any claims on whether the AM and AM-P models differ in TSP problems higher than 50 nodes, it could be worthwhile to study whether the models would differ in larger problem settings. This is left for future work.

Further, results from experiment # 2 further show how close the performance between the Attention Model and Persistent Attention Model is (see table 5.4). In the table, it can be appreciated the validation scores for both models stay close even when tested on problems for the models that were not trained originally.

Additionally, the standard deviation reported shows a similar rate of noise within the validation scores obtained. The means of the Attention Model and Persistent Attention Model both fall within the first standard deviation for all of the results, further making the case that the distributions produced by the validation scores are similar.

An additional interesting result is the similarity of the solutions generated by the models. Figure 5.2 shows the trajectory generated by an AM and AM-P model trained on different problem sizes. The models solve a TSP of 50 nodes. It is possible to observe similarities in the overall shape of the generated solutions, only differing in certain locations.

Like experiment # 1, hypothesis testing (a t-test in this case) is carried out to ensure these observations are statistically sounding. The null hypothesis is the following:

- H_o: The mean reward of the Attention Model and Persistence Attention Model is the same when validated over the same TSP.

As 5.5 shows, the p-values indicate no significant differences between the means obtained. for this reason, we accept the null hypothesis. Note that this result only demonstrates there is not a statistically-significant difference between the models.

These results are rather surprising, as they do not follow the trend observed in the literature, where a performance boost to models using persistence has been observed Pun et al. [2018]. There may be many different reasons behind this behavior. A possibility, given that the Attention Model already achieves good performance in TSP, is that persistent homology information is being extracted through the Graph Attention mechanisms used by the Attention Model. The attention model may not compute this information exactly as the persistent homology tools do, but it could be possible that GAT mechanisms approximate this information well enough, that providing the model with such information is not beneficial.

Another reason for the belief that persistent homology may be used by the Attention Model through its GAT mechanisms is the fact that solutions to the Traveling Salesman problem tend to have certain characteristics. Some classical, exact algorithms for solving the TSP problems have noted the usability of the convex hull in finding the optimal solution to a problem [Allison and Noga, 1984]. This has motivated some work to study ways in which the structure of a graph problem, through persistent homology, could help solve TSP problem instances. Some recent work by Carlsson et al. [2022] has explored this idea through the use of TDA to improve the performance of imperfect TSP solutions. The positive results obtained by their approach through the use of persistence information motivate the usability of persistence in solving TSP problems.

While this hypothesis could be likely, the results presented are not enough to make statements past the fact that the attention model, through its persistent attention model variant, does not seem to gain any significant performance increase in Traveling Salesman. It could be possible that the chosen vectorization approach may not capture enough information about the problem. On a similar note, only Vietoris-Rips complexes were used for analysis, and the impact of other simplicial complexes were not assessed. Verification of this hypothesis as well as the impact of other persistent homology representations and tools would be left for future work.

Chapter 7

Future Work

As mentioned in the discussion and conclusion section, it remains uncertain whether persistent homology could be indeed extracted through the GAT mechanisms of the attention model. Initial results hereby presented suggest that the introduction of persistent homology is not as helpful for solving TSP with the attention model, which one possible hypothesis is that the information is partially extracted through the GAT mechanisms of the model. It is uncertain how much of this information is obtained by the GAT mechanisms, and how similar it is to the signatures extracted by existing persistent homology techniques. Exploring the extent to which the Attention Model may utilize such persistence homology information could open the way for mathematical rigor, as it could be possible that some of the theoretical results obtained in other machine learning models using persistence homology could apply to the Attention Model Hensel et al. [2021]

A possible increment to this work could be investigating the algorithm and interaction effects of the AM and AM-P models on larger TSP problems. As noted in tables 5.1, 5.2, and 5.3, p-values seem to decrease as the TSP problem size is increased. It could be possible that a difference among the algorithms exists in larger TSP problem environments. Unfortunately, training the models over TSP problems larger than 50 nodes tends to become computationally expensive, with experiments lasting several days on modern GPUs (used a GTX 1060 ti and Tesla k20m). It would left for future work to investigate whether AM and AM-P algorithms differ in problems larger than TSP-50.

Another possible direction of future research is to investigate whether AM-P could have better resistance to adversarial attacks. An important result from topology is the stability of persistence diagrams with respect to noise [Edelsbrunner and Harer, 2010], [Carlsson and Vejdemo-Johansson, 2021]. Some research into the capability of persistent homology to transfer stability and resistance to adversarial attacks has shown promising results to the applicability of TDA to improve the robustness of machine learning models [Brüel-Gabrielsson et al., 2020]. Within the realm of Graph Neural Networks, of which GAT models are a part of, some adversarial attacks have been shown to derail learning and decrease the performance of Graph Neural Network models [Sun et al., 2020], [Zhang and Zitnik, 2020]. Future research on the AM-P model hereby presented should explore this question and compare the robustness of the model with respect to adversarial attacks that target graph problems direction like node injection as presented in Sun et al. [2020].

Bibliography

Donald C.S. Allison and M.T. Noga. The l1 traveling salesman problem. *Information Processing Letters*, 18(4):195–199, 1984. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(84)90110-8. URL `https://www.sciencedirect.com/science/article/pii/0020019084901108`.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1409.0473`.

Rickard Brüel-Gabrielsson, Bradley J. Nelson, Anjan Dwaraknath, and Primoz Skraba. A topology layer for machine learning. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 1553–1563. PMLR, 26–28 Aug 2020. URL `https://proceedings.mlr.press/v108/gabrielsson20a.html`.

Peter Bubenik. Statistical topological data analysis using persistence landscapes. *J. Mach. Learn. Res.*, 16(1):77–102, jan 2015. ISSN 1532-4435.

Erik Carlsson, John Gunnar Carlsson, and Shannon Sweitzer. Applying topological data analysis to local search problems, 2022. URL `/article/id/621de62b2d80b7479e4357cd`.

Gunnar Carlsson and Mikael Vejdemo-Johansson. *Topological Data Analysis with Applications*. Cambridge University Press, 2021. doi: 10.1017/9781108975704.

Mathieu Carrière, Frédéric Chazal, Yuichi Ike, Théo Lacombe, Martin Royer, and Yuhei Umeda. Perslay: A neural network layer for persistence diagrams and new graph topological signatures. In *International Conference on Artificial Intelligence and Statistics*, 2019.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL `http://arxiv.org/abs/1406.1078`.

Tamal Krishna Dey and Yusu Wang. *Computational Topology for Data Analysis*. Cambridge University Press, 2022. doi: 10.1017/9781009099950.

Herbert Edelsbrunner and John Harer. *Computational Topology - an Introduction.* American Mathematical Society, 2010. ISBN 978-0-8218-4925-5.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 06–11 Aug 2017. URL `https://proceedings.mlr.press/v70/gilmer17a.html`.

Bruce Golden, L. Levy, and Rakesh Vohra. The orienteering problem. *Nav Res Logist*, 34:307–318, 06 1987a. doi: 10.1002/1520-6750(198706)34:3⟨307::AID-NAV3220340302⟩3.0.CO;2-D.

Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics (NRL)*, 34(3):307–318, 1987b. doi: https://doi.org/10.1002/1520-6750(198706)34:3⟨307::AID-NAV3220340302⟩3.0.CO;2-D. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/1520-6750%28198706%2934%3A3%3C307%3A%3AAID-NAV3220340302%3E3.0.CO%3B2-D`.

Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

Felix Hensel, Michael Moor, and Bastian Rieck. A survey of topological machine learning methods. *Frontiers in Artificial Intelligence*, 4, 2021. ISSN 2624-8212. doi: 10.3389/frai.2021.681108. URL `https://www.frontiersin.org/articles/10.3389/frai.2021.681108`.

Holger H. Hoos and Thomas Stützle. *1 - INTRODUCTION*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, San Francisco, 2005. ISBN 978-1-55860-872-6. doi: https://doi.org/10.1016/B978-155860872-6/50018-4. URL `https://www.sciencedirect.com/science/article/pii/B9781558608726500184`.

M I Jordan. Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. 5 1986. URL `https://www.osti.gov/biblio/6910294`.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ByxBFsRqYm`.

Clément Maria, Jean-Daniel Boissonnat, Marc Glisse, and Mariette Yvinec. The gudhi library: Simplicial complexes and persistent homology. In Hoon Hong and Chee Yap, editors, *Mathematical Software – ICMS 2014*, pages 167–174, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44199-2.

Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. *CoRR*, abs/2002.03282, 2020. URL `https://arxiv.org/abs/2002.03282`.

Justus H. Piater, Paul R. Cohen, Xiaoqin Zhang, and Michael Atighetchi. A randomized anova procedure for comparing performance curves. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 430–438, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1558605568.

Chi Seng Pun, Kelin Xia, and Si Xian Lee. Persistent-homology-based machine learning and its applications – a survey. *arXiv*, 2018(0), 2018. URL `http://dml.mathdoc.fr/item/1811.00252`.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 0033-295X. doi: 10.1037/h0042519. URL `http://dx.doi.org/10.1037/h0042519`.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN 9780134610993. URL `http://aima.cs.berkeley.edu/`.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Yiwei Sun, Suhang Wang, Xianfeng Tang, Tsung-Yu Hsieh, and Vasant Honavar. Adversarial attacks on graph neural networks via node injections: A hierarchical reinforcement learning approach. In *Proceedings of The Web Conference 2020*, WWW '20, pages 673–681, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370233. doi: 10.1145/3366423.3380149. URL `https://doi.org/10.1145/3366423.3380149`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL `http://arxiv.org/abs/1312.6199`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL `https://proceedings.neurips.cc/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf`.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

Xiang Zhang and Marinka Zitnik. Gnnguard: Defending graph neural networks against adversarial attacks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.