Universiteit Leiden

Master Computer Science

Travelling Salesman Problem solved using Deep Reinforcement Learning via alternative Attention Mechanisms and Enhanced Context Embedding

Name:	Nathalia Morales Rojas
Student ID:	s3516423
Date:	05/08/2024
Specialisation:	Computer Science: Data Science
1st supervisor:	Dr. Yingjie Fan
2nd supervisor:	Dr. Thomas Back

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Combinatorial optimization problems like the Traveling Salesman Problem (TSP) pose significant challenges in fields such as manufacturing and operations research, prompting continuous efforts to discover more efficient solutions. The application of Machine learning algorithms, particularly Deep Reinforcement Learning (DRL), has emerged as a promising approach by enabling algorithms to autonomously learn solutions without relying on predefined rules.

This research explores the impacts when employing different embedding techniques (simple linear, simple convolutional, enhanced linear with multiple layers and normalization, enhanced linear with additional complexity) and attention mechanisms (Pointer Network and Self Attention) using an Actor-Critic algorithm to solve TSP sizes of 10, 20, 50 and 100 nodes.

Moreover, this research examines the outcomes of different parameters in the model setup, including batch size, epsilon value for action selection, the number of RNN layers, the number of glimpses for the Pointer Network decoder, and the number of heads for the Self-Attention decoder. Through meticulous parameter tuning, it identifies configurations with the attention mechanism and embeddings that offer the best trade-off between performance and computational efficiency. In addition, it provides an analysis of training times, emphasizing the importance of balancing computational resources with model accuracy.

Overall, this research presents an extensive evaluation of the DRL approach implemented. Key findings indicate that the Pointer Network consistently delivers superior performance in comparison to the use of Self-Attention as the attention mechanism across most TSP instances. It also founf that the simple convolutional embedding performed consistently more effectively in the experiments than all other implemented embeddings. While the implemented approach didn't outperform some of the benchmark models, it demonstrated competitive performance, coming specially close to the LKH-3 benchmark results.

Contents

Chapter 1

Introduction

The TSP (Travelling Salesman Problem), considered a ‘classic’ combinatorial optimization problem in the field of operations research [53], entails finding the shortest route possible that visits the whole set of cities [28]. The complexity of the TSP lies in its combinatorial nature, this means that as the number of cities increases, the number of possible routes increases factorially. Thus, making exhaustive search impractical even for relatively small instances. The TSP is an NP-hard problem meaning that there is no known algorithm that can solve all instances in polynomial time [53]. The significance of TSP extends beyond theoretical interests as it’s constantly used to model many real-world problems ranging from delivery logistics and manufacturing [43, 51, 13, 28] to network design [33].

Combinatorial optimization problems can be solved in a variety of ways, like by applying exact, heuristic, or metaheuristic methods [17]. When applying exact methods to solve combinatorial optimization problems we have a guarantee to find the optimal solution. However this is often computationally expensive and impractical for very large instances due to the NP-hard nature of the problem. Meanwhile, solving combinatorial problems through heuristic methods can generate a good enough answer but cannot guarantee an ideal result, making them the next best alternative when exact solutions are computationally infeasible. Finally, metaheuristic methods as solvers for combinatorial optimization problems can balance between exploration and exploitation to find high-quality solutions within reasonable timeframe. They are designed to guide other heuristics to explore the solution space more effectively [17].

In attempt to solve the TSP in the most approximately efficient manner, researchers have explored various heuristic and approximation algorithms such as simulated annealing [12] and ant colony optimization [62], each offering a trade-off between solution quality and computational efficiency. Practical applications of these methods reflect a pragmatic approach, aiming for solutions that are both quick and good enough for real-world use.

Beyond heuristic and approximation algorithms, industry solvers have made significant contributions to the practical resolution of the TSP. Notably, CPLEX, Gurobi and LKH-3 are at the forefront of this effort. Both CPLEX and Gurobi are powerful exact solvers that stand out in solving linear programming, mixed-integer programming, and quadratic programming problems. CPLEX, developed by IBM [30], is a high-performance mathematical programming solver, regarded for its ability to handle complex constraints and large datasets efficiently, making it a staple in industries requiring optimal logistical and routing solutions. Gurobi's solver is recognized for its high performance, and it's often the solver of choice in both academic research and industry applications [26]. Additionally, LKH-3 (Lin-Kernighan-Helsgaun) [22], is an advanced implementation of the Lin-Kernighan heuristic developed by Keld Helsgaun, a solver that incorporates powerful local search techniques and strategic exploration of the solution space.

Common approaches such as exact and heuristic methods are less effective in dynamic and large-scale systems due to their incapacity to learn from data, handle uncertainty, and require substantial manual tuning. Recent advances in solving the TSP problem and its variations have been achieved through the application of ML models [3, 50, 49, 5]. These ML approaches can be seen as metaheuristic methods because they don't guarantee an optimal solution but aim to find high-quality solutions through learning and adaptation. They offer a new way for tackling combinatorial optimization problems by leveraging the ability of ML algorithms to learn patterns and make decisions based on features of the problem instance.

For example, supervised learning techniques can be used to train models on known optimal or near-optimal solutions, enabling the model to generalize to new instances [9, 59, 50, 16]. However, traditional ML methods sometimes struggle with the combinatorial nature and size of problems such as the TSP [5].

To handle these challenges, RL (Reinforcement Learning) has been increasingly been used [14, 59]. This is due to its ability to learn efficient policies for decision-making without requiring explicit instructions on how to perform tasks. RL involves an agent that learns to make a sequence of decisions by interacting with an environment and receiving feedback in the form of rewards [52]. Specifically in context of the TSP, the agent learns to construct routes by selecting the next city to visit based on the current state. The reward for the agent is related to the total (shortest) route length. This approach allows the agent to learn effective policies for constructing high-quality routes over time.

Building on RL, DRL (Deep Reinforcement Learning) has shown even greater promise [23, 4, 15, 44, 14, 65]. DRL combines the decision-making framework of RL with the representational power of deep learning. By using neural networks, DRL can handle the high-dimensional state and action spaces typical of large TSP instances [1, 59]. Once trained, a neural network can generate solutions in a fraction of the time it would take to compute them using traditional optimization methods. For instance, techniques like Deep Q-Networks, Policy Gradient methods, and A-C (Actor-Critic) algorithms [52] have been adapted to the TSP, where the neural network approximates the value function or policy, enabling the agent to learn complex strategies for route construction [23, 4, 15, 44, 14, 65].

Transformers, originally designed for natural language processing, have shown remarkable flexibility and effectiveness in solving combinatorial optimization problems like the TSP [54, 4, 11, 35, 47]. Their integration with DRL has opened new directions of study showing great advancements in solution quality [21, 18, 64, 47, 60]. This is a consequence of the combination of the sequential decision-making process of DRL, with the parallel processing capabilities of transformers. Transformers, with their Self-Attention mechanisms, enable the DRL to consider the entire input sequence context at each step, enhancing the agent's ability to make informed decisions.

Transformers consist of two main parts, an encoder, and a decoder. An encoder processes the input sequence and produces a set of embeddings that capture relationships between entities. The decoder generates the output sequence by focusing on both the encoder's output and the previously generated tokens, using a defined attention mechanism [32]. The embeddings and the attention mechanism are the two main components that can alter the behaviour of a transformer.

The TSP can be modeled as a sequence-to-sequence problem where the input sequence is the list of cities and the output sequence is the ordered route, showing the transformation of one sequence into another optimal sequence through learned representations and attention mechanisms. Embeddings specifically translate the cities of the TSP problem (input data) into continuous vector representations in a high-dimensional space, allowing the model to capture the relative positions and distances between cities.

This continuous representation is needed for the model to absorb information and choose the most optimal route. The attention mechanism allows the model to focus on specific cities that are more relevant at each step of constructing the route. The model can prioritize certain cities over others by calculating attention scores, which allows it to improve the quality of the solutions it generates.

There are several types of attention mechanisms that can be implemented in a transformer [2, 38, 45], this research is centered on the implementation of Pointer Networks and Self-Attention mechanism in order to compare their effectiveness in solving sequence-to-sequence problems like the TSP.

PN (Pointer Networks) [56] use attention to sequentially select elements from the input set, this is ideal for the TSP because its goal is to construct an ordered sequence of visits. The PN points to the next city to visit based on the current state, learning a policy for route construction that minimizes the total distance traveled. Self-Attention, on the other hand, allows the model to consider the relationships between all pairs of cities simultaneously, rather than focusing on a single pair at a time [54]. This holistic view allows the model to capture complex dependencies and interactions between cities [61, 43].

The Self-Attention variation implemented for this research is the multi-head Self-Attention [45]. This type of Self-Attention allows the decoder to compute multiple sets of attention scores in parallel. It allows the model to focus on different parts of the input sequence by computing attention scores across multiple heads, each capturing different aspects of the relationships between tokens. We've added a masking that maintains the autoregressive property required for sequence generation, not allowing the model to regenerate past the current decision point.

The primary distinction between the Self-Attention and PN mechanisms is that the PN concentrates on selecting and ordering elements from the input sequence based on the current state to form the output, essentially producing a reordered subset of the input sequence. Whereas Self-Attention computes attention scores for each token in relation to all other tokens, applies a mask to ensure that only available tokens are considered during prediction. Then, it generates context vectors that incorporate information from relevant preceding tokens and based on all these factors it selects the output sequence. The attention scores between the two mechanisms are also different as Self-Attention computes scores between all pairs of elements within the sequence and the PN computes scores between the decoder state and the encoder outputs to select input elements.

This study aims to demonstrate the strengths and weaknesses of four alternative embedding methods and two attention mechanisms in the context of a DRL implementation to solve the TSP problem. This study provides a thorough analysis on the impact of the embedding and attention mechanism on the model's solution quality, computation efficiency and ability to generalize. To ensure its robustness and relevance, it analyzes the previously mentioned approaches against three industry-standard benchmarks, presenting an understanding of how the methodologies compare to commonly accepted standards in the field.

Building on the various experiments designed for this research, it also presents an investigation into the impacts of parameter tuning per TSP instance implemented (10, 20, 50, 100). Therefore observing the experiment's generalization capabilities in different instance size of the TSP problem with the different configurations and in the most optimal setting to compare them to the benchmark models.

In the next sections we will continue by exploring the related work in Chapter 2. This is followed by Chapter 3, Methodology, which explains the details of the methodology implemented. Chapter 4, Experiments and Results explores the experimental setup along with the analysis of the results obtained. Chapter 5; Conclusion, opens a discussion referring back to the results, discusses the limitations and summarizes the contributions of the study.

Chapter 2

Literature Review

The TSP (Travelling Salesman Problem) and its variants have long been important topics of research [53] due to their critical importance in obtaining solutions for logistic and transportation systems [17, 6]. Understanding and solving these problems is of academic and practical relevance, as more dependable, scalable, and efficient solutions are required to address the complex and dynamic nature of today's routing challenges [65, 58, 47, 41]. These long-standing problems keep finding new approaches because of the rapid evolution in fields like ML (Machine Learning) [14, 28, 43, 51].

The application of ML has led to significant advancements in optimization techniques. It has introduced new heuristic and metaheuristic models that leverage data-driven insights to solve instances of the TSP and its variants more efficiently [2]. These models range from training on historical data to recognize patterns and predict near-optimal solutions, to using techniques like supervised learning, reinforcement learning, and clustering algorithms that present improvement in initial solution guesses and enhance local search strategies [57, 50, 3].

The ability of DL (Deep Learning) to model complex, high-dimensional data has transformed the method of solving the TSP. Specifically, DRL (Deep Reinforcement Learning) has shown promise as it combines RL (Reinforcement Learning)'s decision-making framework with DL's powerful representation learning [8]. Models like Deep Q-Networks (DQN) and policy gradient methods have been employed to train agents that can iteratively improve TSP solutions through trial and error, guided by reward signals that encourage shorter route lengths [19, 20, 24]. These approaches have achieved remarkable success in solving large and complex TSP instances by learning to make sequence choices that generalize well to unseen problems [39, 60].

Alongside DRL methods, the introduction of transformers has marked a new era in tackling the TSP. Introduced by Vaswani et al. in 2017 [54] transformers were initially designed for natural language processing tasks. But because they use Self-Attention mechanisms to capture long-range dependencies in sequential data they have been implemented in combinatorial problems such as the TSP due to their ability to model global interactions. Transformers are particularly suitable for solving the relationship between cities since it needs to be considered as a whole in order to produce more optimal results. Recent studies have applied transformer architectures to the TSP, using their capacity to handle variable-length input sequences and capture intricate patterns [5, 60, 47].

There are two main components that affect the behavior of a transformer. Firstly, the embeddings, because they provide a dense, low-dimensional representation of high-dimensional data, capturing the underlying relationships between different entities [54], Secondly, the attention mechanism, that let's the transformer focus on other elements within the same sequence [32].

In the context of the TSP, embeddings represent the cities or nodes in a way that preserves their spatial and relational properties. By embedding the cities into a continuous vector space, transformers can process and understand the complex interactions and dependencies between them [44]. This allows the model to encode important features such as distances, connectivity, and other relevant attributes that influence the optimization process. Consequently, embeddings facilitate the transformer's ability to generalize and learn from smaller instances of the TSP and apply this knowledge to solve larger, more complex instances, enhancing both the efficiency and accuracy of the solutions generated, as seen in recent studies [31].

The second main component in the decision making process of the transformer is the attention mechanism [2], specifically the Self-Attention mechanism. The Self-Attention mechanism allows each element of the input sequence to dynamically focus on other elements within the same sequence to compute its representation. Altering the attention mechanism can significantly change the behavior of the transformer by modifying how dependencies between elements are captured and processed. For example, variations in the attention mechanism can influence the model's ability to prioritize certain inputs over others.

Recent studies have explored multiple modifications to the attention mechanism to improve performance on combinatorial problems and DRL tasks [45]. For example, Kool et al. [32] introduced the attention model for solving the TSP, demonstrating that fine-tuning the attention mechanism could lead to more efficient solutions. Similarly, research by Bello et al. [4] showed that combining Self-Attention with RL significantly improved the performance of neural combinatorial optimization models.

The introduction of multi-head attention in transformers, as detailed by Vaswani et al. [54], allows the model to attend to different parts of the sequence simultaneously, capturing a richer set of dependencies and improving overall performance. Studies like that of Xu et al. [60] have further refined these techniques, exploring the impacts of varying the number and configuration of attention heads on the model's effectiveness in handling complex sequences.

Other than the Self-Attention mechanism there's also the PN (Pointer Networks) which are a type of neural network architecture specifically designed to address problems where the output is a sequence of positions in the input [56]. Unlike traditional sequence-to-sequence models, which generate output tokens from a fixed vocabulary, PN produce output by pointing to elements in the input sequence.

PN are a promising venue of research for solving the TSP using DRL given their architecture is inherently well-suited for tasks where the output sequence needs to directly correspond to elements in the input, such as the case in the TSP. They effectively learn to point to the next city to visit based on the current state, dynamically constructing the route.

Also, integrating PN with DRL combines the strengths of both approaches as DRL enables the model to learn optimal policies through interactions with the environment, providing a framework for learning effective strategies for route construction. This is particularly beneficial for TSP, as it allows the model to improve its performance through trial and error, guided by a reward signal that incentivizes shorter routes.

Research has shown that combining PN with DRL can lead to significant performance improvements. For instance, Bello et al. [4] demonstrated that a model trained with a combination of PN and RL outperforms traditional methods on various combinatorial optimization tasks, including the TSP [4]. This approach builds on the sequential decision-making capability of RL and the input-output alignment of PN, resulting in a powerful model for solving TSP instances.

Nazari, et al. [44] addressed some limitations of traditional PN in handling dynamic environments. They proposed a method made up of an integration of a RNN (Recurrent Neural Network) with an advanced attention mechanism attempting to dynamically update the state of the network based on changes in customer demands and vehicle status, specifically made to solve the VRP problem. The discussion on computational efficiency and scalability in their paper is limited, encouraging future research to expand on these aspects by exploring the computational overhead caused by the recurrent and attention-based components of the model.

Recent developments in DRL show a promising direction towards addressing these limitations. Below, in Table 2.1, is the comparison of the State-of-the-Art papers on the subject. For example, the use of advanced neural architectures that incorporate dynamic graph neural networks (GNNs) and attention mechanisms can potentially enhance the model's responsiveness to real-time changes in problem parameters as used in Liang et al. [34]. These models adapt to the changing state of the problem and also learn from a diverse range of scenarios, improving their generalization capabilities across different instances of the problem.

Additionally, the integration of metaheuristic components with DRL frameworks has shown potential in navigating the vast solution spaces more effectively [17, 7]. Those hybrid approaches use the strengths of heuristic optimizations to provide initial good-quality solutions and refinements. DRL continuously adapts and improves the decision-making process based on interactive learning from the environment [17].

Innovative embedding techniques have also emerged, improving the way input data is represented and processed by these models. Positional encoding, a common feature in transformer models, has been refined to better capture the spatial relationships inherent in the TSP [11]. Additionally, graph-based embeddings [55] have been introduced, where cities are treated as nodes in a graph, and edges represent the distances between them. This representation allows the model to naturally incorporate the structure of the problem.

Recent developments in DRL have introduced sophisticated techniques such as Proximal Policy Optimization (PPO) [48] and Soft Actor-Critic (SAC) [27], which have shown great promise in stabilizing training and improving the performance of DRL agents. These methods have been adapted to work in conjunction with transformer architectures to tackle the TSP more effectively. For instance, PPO's clipping mechanism helps in maintaining stable policy updates, which is crucial when training large models like transformers. Similarly, SAC's entropy maximization encourages exploration, ensuring that the agent doesn't get stuck in local optima.

Table 2.1: Comparison of Papers on state-of-the-art methods for solving TSP and its variants, all but the first paper have been published from 2020 onwards. This table highlights the algorithm variant to be solved, along with the algorithm used to solve it and some of the inner components that are contained in the attention mechanism, embedding layers and the transformer's decoder and encoder layers. If there was no explicit information found in the paper about the structure implemented the cell contains a '-' symbol.

Paper	**TSP/VRP Variant(s)**	**Field**	**Algorithm**	**Attention**	**Embedding**	**Decoder**	**Encoder**
Nazari et al. [44]	TSP, CVRP, SVRP, Split Delivery Problems	RL	A-C	Glimpse + PN/Beam Search	1-D Conv.	Linear RNN	No
Gao, et al. [23]	VRP, CVRPTW	RL	A-C	GAT	Node and Edge	GRU-based	Mod. GAN
da Costa, et al. [14]	TSP	RL	PG	PN	LSTM	Custom	GCN
Zhao, et al. [65]	VRP, VRPTW	RL	Adaptive A-C	Basic	Yes	LSTM	Yes
Wu, et al. [59]	TSP, CVRP	RL	A-C	Self	Linear + Position	Node pair selection	No
Xu, et al. [61]	TSP, CVRP, SDVRP, OP, PCTSP	RL	PG	Self	Batch normalization + Gate aggregation	Attentive Aggregation Module	No

Continued on next page

Table 2.1: Comparison of Papers on state-of-the-art methods for solving TSP and its variants, all but the first paper have been published from 2020 onwards. This table highlights the algorithm variant to be solved, along with the algorithm used to solve it and some of the inner components that are contained in the attention mechanism, embedding layers and the transformer's decoder and encoder layers. If there was no explicit information found in the paper about the structure implemented the cell contains a '-' symbol. (Continued)

Paper	**TSP/VRP Variant(s)**	**Field**	**Algorithm**	**Attention**	**Embedding**	**Decoder**	**Encoder**
Liu, et al. [36]	TSP for ALMRRC	ML	Seq2Seq	-	Word2Vec	-	-
Mo, et al. [43]	TSP for ALMRRC	ML	Custom Attention	ASNN for Pair-Wise + PN	Seq2Seq	LSTM	LSTM
Zhao, et al. [66]	MOTSP	RL	GPN	Self + PN	Feature	Yes	Pointer and Graph
Liang, et al. [34]	MTSP, MinMax-MTSP, Bounded-MTSP	RL	B&B and BiGNN	Self	-	Yes	GNN
Goh, et al. [24]	TSP	RL	REINFORCE	Self	1-D Conv.	Sinkhorn	Basic Transformer
Fellek, et al. [20]	CVRP	RL	PPO	GATv2	Node and Edge	Yes	No

Continued on next page

Table 2.1: Comparison of Papers on state-of-the-art methods for solving TSP and its variants, all but the first paper have been published from 2020 onwards. This table highlights the algorithm variant to be solved, along with the algorithm used to solve it and some of the inner components that are contained in the attention mechanism, embedding layers and the transformer's decoder and encoder layers. If there was no explicit information found in the paper about the structure implemented the cell contains a '-' symbol. (Continued)

Paper	**TSP/VRP Variant(s)**	**Field**	**Algorithm**	**Attention**	**Embedding**	**Decoder**	**Encoder**
Chen and Luo [8]	TSP, CVRP	RL	PPO	Synthetic	Node	Yes	Yes
Zhang, et al. [63]	TSP	RL	REINFORCE	Sparse	Node	No	No
Fellek, et al. [19]	TSP	RL	PPO	Gated Graph	Node and Edge	Yes	GNN
Wang, et al. [58]	VRP, CVRP	RL	Adaptive A-C	MHA	Node and Edge	Yes	GAN
Ouyang, et al. [46]	TSP	RL	eMAGIC	Custom	-	Yes	GNN and MLP
Fellek, et al. [21]	VRP	RL	PPO	EEMHA	Node and Edge	Yes	Yes
Vo, et al. [57]	TSP	RL	Branch-and-Cut	No	MLP	No	GNN
Fang, et al. [18]	TSP, CVRP	RL	PG	Custom	Dynamic	Multi-view	Nested-view

Continued on next page

Table 2.1: Comparison of Papers on state-of-the-art methods for solving TSP and its variants, all but the first paper have been published from 2020 onwards. This table highlights the algorithm variant to be solved, along with the algorithm used to solve it and some of the inner components that are contained in the attention mechanism, embedding layers and the transformer's decoder and encoder layers. If there was no explicit information found in the paper about the structure implemented the cell contains a '-' symbol. (Continued)

Paper	**TSP/VRP Variant(s)**	**Field**	**Algorithm**	**Attention**	**Embedding**	**Decoder**	**Encoder**
Ma, et al. [40]	TSP, CVRP	RL	NeuOpt	Custom	MLP	RDS	Yes
Ke, et al. [31]	TSP, CVRP	RL	Custom	E-GAT and E-MHA	Edges	KL	E-GAT and E-MHA
Cheng, et al. [10]	TSP	RL	PG	MHA and SHA	Yes	SHA	MHA
Luo, et al. [37]	TSP, CVRP	ML	SIL	Yes	Node	Yes	Yes
Min, et al. [42]	TSP	ML	GNN	SAG	Yes	GNN	GNN
Zhao, et al. [64]	TSP	RL	gSaS	MHA	Yes	Yes	Yes
Grinsztajn, et al. [25]	TSP, CVRP, KP, JSSP	RL	Poppy	Yes	Yes	Yes	Yes
Hottung, et al. [29]	TSP, CVRP, JSSP	RL	EAS	Yes	Yes	Yes	Yes

Continued on next page

Table 2.1: Comparison of Papers on state-of-the-art methods for solving TSP and its variants, all but the first paper have been published from 2020 onwards. This table highlights the algorithm variant to be solved, along with the algorithm used to solve it and some of the inner components that are contained in the attention mechanism, embedding layers and the transformer's decoder and encoder layers. If there was no explicit information found in the paper about the structure implemented the cell contains a '-' symbol. (Continued)

Paper	**TSP/VRP Variant(s)**	**Field**	**Algorithm**	**Attention**	**Embedding**	**Decoder**	**Encoder**
This paper	TSP	RL	A-C	Multi Head Self Attention and Pointer Network	4 embedding variations	Yes	No

Chapter 3

Methodology

This section defines the methodologies used for the Traveling Salesman Problem, the Actor-Critic Deep Reinforcement Learning algorithm, and the architectural elements in charge of the decision-making process, which together constitute the proposed approach in this study. First, in Section 3.1 we define the TSP (Travelling Salesman Problem). Section 3.2 describes the Actor-Critic algorithm, including the precise DRL methodology used to solve the TSP and the evaluation metrics of the approach. The significance and concept of embeddings are explained in Section 3.3. Section 3.4.1 focuses on the Pointer Network, explaining its functionality and architectural details. Section 3.4.2 explores the function and underlying architecture of Transformers and Self-Attention. Finally, Section 3.5 examines the overall architectural framework as well as the data flow in the decision making process.

3.1 The Travelling Salesman Problem

The TSP is a classic optimization problem in combinatorial optimization. It seeks to determine the shortest possible route that visits a complete set of cities (nodes) exactly once and returns to the origin depot (base location). We use the definition for the TSP provided by Nazari [44] for this study. The problem is defined as follows:

Definition 3.1 (TSP (Travelling Salesman Problem))**.** The main objective of this problem is to minimize the total length of the route. The input comprises a order-invariant sequence (set) of N nodes representing cities to be visited $s = \{x_1, x_2, \dots, x_N\}$ located in a 2-D space (latitude and longitude), each node (city) represented by 2-D coordinates. The output a is a permutation of these nodes $(a_1, a_2, \dots, a_N)$, where a_i identifies the index of the ith node in the sequence. The task starts at an arbitrary depot node x_{a_1}, and the 'salesman' or agent must visit each node exactly once. The exception is the depot node, which the salesman will return to last. The minimization equation is defined as:

$$\text{Minimize} \quad L(a|s) = |x_{a_N} - x_{a_1}|^2 + \sum_{i=1}^{N} |x_{a_{i+1}} - x_{a_i}|^2 \tag{3.1}$$

In the context for this study, the problem is modeled as a Reinforcement Learning scenario where the input is a specific problem instance, and an episode consists of an encoding-decoding cycle that produces the complete solution for the provided sequence, named the node permutation a.

Before the start of an episode, the nodes are encoded into fixed node embeddings, which we will explain how in Section 3.3, for the duration of that episode. The decoding process unfolds over a set number N of timesteps. Where at each timestep t, the agent selects a node that has not yet been visited, with the action a_t representing the index of this node in the provided sequence. The state s_t contains the set of nodes s and information about the transition dynamics, such as a mask, of what nodes have been visited already. The mask for visited nodes is explained in the Environment Subsection 3.2.1.

The reward, defined and allocated at the conclusion of each episode, is given by the negative Euclidean distance of the chosen sequence, denoted as $-L(a|s)$. Given the state s, the goal is to develop a policy that can produce the node permutation a to optimize the episodic reward $-L(a|s)$. This negation of the reward facilitates the objective of achieving the sequence with the shortest possible distance between nodes. The policy is parameterized by θ and is expressed as follows:

$$p_\theta(a|s) = \prod_{t=1}^{N} p_\theta(a_t|s_t, a_{1:t-1}) \tag{3.2}$$

3.2 The Actor-Critic Algorithm

The Actor-Critic algorithm, originally defined and created by Sutton and Barto in 1992 [52] is a type of RL that combines both policy-based and value-based approaches. Policy-based methods directly learn a policy that maps states to actions, while Value-based methods focus on learning the value function without explicitly learning a policy [52]. This is specially important for a problem like the TSP as the search space can grow exponentially with the number of cities and this algorithm allows for continuous updates to both the policy and the value estimate. The Actor-Critic algorithm allows the agent to adapt more fluidly to changes in the problem dynamics compared to purely policy-based or value-based methods.

There are two main components of the Actor-Critic algorithm, the *'actor'* and the *'critic'*. Both components make up the agent in our approach. They work simultaneously to improve the decision-making of the agent. The *'actor'* network proposes actions, in this case a sequence of selected nodes (cities) in a route, while the *'critic'* network evaluates these actions by estimating the value function, which reflects the expected reward of taking an action given a particular state [52]. This approach improves learning efficiency and decision quality by continuously updating policies based on feedback from the *'critic'*, helping to identify optimal routes.

In our implementation of the Actor-Critic algorithm, both the actor and critic components are parameterized using deep neural networks. It uses the features of DL to effectively manage complex feature interactions within the TSP, such as geographical distributions and route dependencies. On Algorithm 1 we have defined the algorithm as implemented in this study.

Algorithm 1 starts by setting up of the environment and parameters, such as the data generator and reward function. We see then the iterative computation of action probabilities using the actor network, followed by action selection and environment state updates. These steps allow the agent to explore and exploit the solution space effectively. After taking an action, the critic network computes the TD error (called advantage in Figure 3.1), that measures the difference between predicted and actual outcomes, guiding the network weights' updates. Both actor and critic iterate until they reach a terminal state.

There are different processes involved in training and evaluating the model. While the evaluation phase evaluates the model's performance on test data, the training phase concentrates on optimizing the network parameters. As a result, the agent learns how to construct short routes by validation of the learned policies.

Algorithm 1 Reinforcement Learning Agent for TSP using Actor-Critic Architecture

```
1: Input: a differentiable policy parameterization $\pi(a|s,\theta)$
2: Input: a differentiable state-value function parameterization $\hat{v}(s,w)$
3:
4: Initialize:
5:     Environment, Data Generator, Reward Function, Training or Evaluation Mode
6:     Actor and Critic networks using deep neural network layers $\theta$ and $w$
7:     Optimizers for actor $\mathcal{O}_\theta$ and critic $\mathcal{O}_w$
8:     Embedding and attention mechanisms parameterized
9:
10: Actor-Critic Model:
11:     Convert input to embeddings
12:     Repeat for each decision step:
13:         Compute action probabilities $\pi(a|s,\theta)$
14:         Select action $a$, update environment state $s'$
15:         Calculate TD error $\delta = r + \gamma\hat{v}(s',w) - \hat{v}(s,w)$
16:         Update $\theta$ and $w$ using gradients $\nabla_\theta\delta$ and $\nabla_w\delta$
17:         Record rewards $r$, actions $a$, and states $s$
18:         until terminal state is reached
19:     Return rewards and state evaluations
20:
21: Training Step:
22:     Obtain predictions from Actor-Critic Model
23:     Calculate advantages, actor loss $\mathcal{L}_\theta$, and critic loss $\mathcal{L}_w$
24:     Update model parameters $\theta$ and $w$ via backpropagation
25:     Apply updates using $\mathcal{O}_\theta$ and $\mathcal{O}_w$
26:
27: Evaluation Method:
28:     Use batch of data sets for model evaluation through Actor-Critic Model
29:     For each batch:
30:         Predict actions and states
31:         Compute and log rewards $r$
32:         Optionally update model based on performance metrics
33:
34: Execution Loop:
35:     While task not completed:
36:         Fetch data, update model through Training Step, evaluate performance with
    Evaluation Method
37:         Return results and update logs
```

3.2.1 Environment

The implemented environment is specifically designed to set up and manage the state dynamics of the TSP. It is initialized with parameters such as the number of nodes (cities) and their dimensions (coordinates). The data generator creates two-dimensional coordinates randomly to imitate the geographical structure of cities. It's parameterized by batch size. Allowing for the simultaneous analysis of multiple problem cases while improving computing efficiency. The state class is used to store a mask tensor that tracks the nodes visited during the exploration process. At each step the model updates the state based on the agent's actions.

3.2.2 Early Stopping Mechanism

We incorporated an early stopping mechanism in the agent's training to increase both the efficiency and adaptability of our approach.

This mechanism continuously monitors the model's performance and terminates training if there is no significant improvement in the solution quality after a predefined number of iterations. The early stopping criteria is based on the model's performance and it's assessed at regular intervals. If the model's performance plateaus, showing no further improvement in the rewards obtained, the training stops to prevent overfitting and conserve computational resources.

The best model is always saved if a better score is obtained. This guarantees that the training process is both time-efficient and resource-efficient, focusing computational efforts only on iterations that contribute to the model's learning. The early stopping mechanism is particularly helpful in handling large-scale TSP instances, where prolonged training times are impractical, thereby making the DRL model more adaptive and suitable for real-world applications. We specifically saw an improvement of up to 85% in time reduction once implemented, allowing us to train certain TSP model instances for as little as a few minutes. We show the results in Section 4.

3.2.3 Action selection

At the action selection we have two priorities: actions need to be both explored and exploited efficiently. Our implementation starts by converting the action logits into a probability distribution using the softmax function (seen in Equation 3.3), which makes the logits sum to one, forming a valid probability distribution over possible actions. We obtain the record of the previously the visited positions to prevent reselection of previous nodes. We return the adjusted probabilities and the indices of the selected actions.

We support two strategies for action selection: epsilon-greedy and stochastic. An epsilon-greedy algorithm primarily selects the action with the highest probability but based on an epsilon value it occasionally picks a random action, the epsilon value decreases over time to shift from exploration to more exploitation of the known best actions. Conversely, the stochastic method selects actions based on the probability weights directly, allowing for proportional exploration of the action space.

Definition 3.2 (Softmax function)**.** In this function z_i is the i-th element of the input vector z. K is the total number of elements in vector z. e is the base of the natural logarithm. The output of the equation is a probability distribution over K different possible outcomes.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.3}$$

3.2.4 Evaluation metrics

As the model needs to award high scores to the shortest routes and bad scores for longer routes we defined the reward for the selected route by the agent as the negative value of the total (euclidean) distance traveled. This approach for calculating the reward inversely ties the length of the route with the actor's reward, favoring the use of shorter, more efficient routes.

Definition 3.3 (Reward Function Definition)**.** Given a route selected by the agent, the reward is calculated based on the total Euclidean distance traveled. We denote the sequence of coordinates that define the route as $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where (x_i, y_i) is the coordinate of the i-th point in the route, and there are n points in total.

The Euclidean distance between two consecutive points (x_i, y_i) and (x_{i+1}, y_{i+1}) is given by:

$$d(i, i+1) = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

The total distance D for the entire route is the sum of the distances between all consecutive points. The reward R for the route is defined as the negative value of the total distance:

$$R = -\sum_{i=1}^{N-1} d(i, i+1) \tag{3.4}$$

Embedding 1 Simple Linear Embedding

```
1: Initialization: Number of input features, Embedding dimension
2: Define a linear layer to project input features to the embedding dimension
3: Forward Pass:
4:     Flatten input to apply the linear layer
5:     Return output
```

Embedding 2 Simple Convolutional Embedding

```
1: Initialization: Embedding dimension, kernel size
2:     Define a 1D convolution layer with specified in-channels, out-channels, and kernel size
3: Forward Pass:
4:     Apply single convolutional layer
5:     Return output
```

3.3 Embedding input data

Embedding the input data is important as it allows the model to transform raw data into a more interpretable representation that captures inherent patterns present. Learning from higher-dimensional spaces often involves complex features that can be challenging to abstract. However, the process of embedding the input data simplifies their representation, enhancing the model's ability to effectively extract and utilize the relevant information. Embeddings help reduce the dimensionality of the input space without significant loss of information, enabling models to train faster and often with better generalization to new, unseen data.

We have chosen to evaluate four embedding techniques to determine the best conditions for our model. Our testing approach includes two basic embeddings and two advanced. In the basic embeddings, the input is processed through a single linear or convolutional layer. The advanced methods, on the other hand, involve embedding the input data through multiple neural network layers and incorporating batch normalization for potential enhanced performance.

On our first approach illustrated in Embedding 1 we created a simple linear embedding that offers a straightforward, less computationally intensive alternative. This embedding option linearly transforms the input features into an embedding space, prioritizing speed and simplicity over the extraction of complex features. While it may not capture as detailed information as the multi-layer approaches, it serves as a baseline to assess the necessity and impact of more complex embedding methods.

In Embedding 2 we define an alternate version for the simple linear embedding

Embedding 3 Enhanced Linear Embedding with Multiple Layers and Normalization

```
 1: Initialization: Number of channels, Embedding dimension
 2: Define three convolutional layers with padding
 3: Each convolution followed by batch normalization and ReLU activation
 4: Implement dropout after final convolution
 5: Add residual connection to adjust dimensions if necessary
 6: Forward Pass:
 7:     Apply convolutions, ReLU activations, and batch normalization sequentially (3 times)
 8:     Apply dropout
 9:     Add output from the last convolutional layer to the residual path output
10:     Return output
```

method but instead of a linear embedding we apply a single-layer 1 dimension convolution, transforming the input data into a higher-dimensional space to capture local dependencies within the sequence. The purpose of this approach is also to serve as a baseline to observe the impact of the following more complex methods.

For the third and fourth embedding methods, we employ multiple convolutional layers in an effort to identify the most efficient approach that provides enhanced contextual understanding of the data.

Embedding 3 shows our first enhanced embedding approach, this approach uses multiple convolutional layers with padding, batch normalization, and ReLU activations to enhance the feature extraction process. Padding is added because it ensures that the spatial dimensions of the feature maps are preserved after each convolution operation, this helps in capturing fine details from the input features throughout the network. Batch normalization is applied after each convolutional layer to normalize the output, allowing for higher learning rates and improving the overall robustness and performance of the model. ReLU activations introduce non-linearity to the model, enabling it to learn complex patterns and representations and it helps mitigate the vanishing gradient problem. A dropout layer is included to mitigate the risk of overfitting, and a residual connection is added to facilitate training of deeper networks.

Embedding 4 presents the second version of our enhanced embedding approach, this version simplifies the structure compared to the first but retains significant complexity. It includes ReLU activations after convolutional layers and a residual connection that helps maintain the network's learning capability over deeper architectures. This allows the network to learn more complex features without significant information loss during propagation through the network layers.

Embedding 4 Enhanced Linear Embedding with Additional Complexity

```
Initialization: Number of channels, Embedding dimension
    Initialize the first convolution layer with ReLU activation
    Prepare second convolution layer and a residual connection for dimension adjustment
Forward Pass:
    Pass through the first layer and ReLU activation
    Apply the second convolution layer
    Combine with the output from the residual connection
    Return output
```

We have summarized the implemented embeddings once more in Table 4.2 for the reader's convenience and to facilitate easier recall of the embedding techniques used.

3.4 Attention Mechanism

Attention mechanisms help the RL agent make better-informed, context-aware decisions, specially important when trying to solve the TSP. The use of attention mechanisms allows for efficient processing of complex, variable-length input sequences and helps agents learn and generalize from diverse problem instances. In this next section we will introduce the two approaches we implemented and experimented in order to obtain better results with more contextual decision making as well as enhanced learning of complex patterns for solving the TSP.

3.4.1 Pointer Networks

Unlike traditional approaches that might use heuristics or manually defined rules for selecting the next city, PN (Pointer Networks) [56] with attention mechanisms learn to solve the problem end-to-end by pointing to elements in the input sequence. This is achieved through a softmax layer that computes a probability distribution over the input positions, effectively selecting elements from the input as the output. In the context of the TSP, PN leverage the attention mechanism to sequentially select cities based on the current partial route i.e., the sequence of cities already visited. At each step, the model points to the next city to visit by computing attention scores over the cities that have not yet been visited. The attention mechanism dynamically adjusts to focus on different input cities as the sequence grows, determining the most logical next city based on the current context.

PN can often find near-optimal solutions much faster than classical methods, especially as the problem size grows. But they may require substantial computational resources for training. Another great advantage for which we have implemented the PN as one of the attention mechanism approach is that a trained model with PN can adapt to different instances of TSP (varying numbers of cities, distances, etc.) without needing changes in the algorithm or reprogramming, as long as the model is robustly trained on a diverse set of problems.

Glimpses

As an additional feature for the PN implementation, we have decided to implement Glimpses. A glimpse in the context of a PN is the result of the attention mechanism applied at a particular decoding step. It computes a weighted sum of the features of all input elements, where the weights are determined by the attention scores. These scores measure how much the current output (at a given step of decoding) should 'attend' to each input element. The attention mechanism itself is often realized through a soft attention model, which allows for differentiable operations and end-to-end training of the network.

At each step in the decoding process, the PN calculates attention scores for each element in the input sequence. This can be seen in line 13 of Algorithm 1 where we obtain the action probabilities, or action scores. These scores are derived based on the current state of the decoder and the entire input sequence. This involves a compatibility function that assesses how well the features of the input at a given position match the current decoder state.

Inside the PN, the glimpse is computed as a weighted sum of the input features, where the weights are the attention scores computed in the previous step. Each feature of the input contributes to the glimpse proportionally to its attention score, allowing the decoder (PN in this case) to focus more on relevant parts of the input.

3.4.2 Transformers and Self-Attention

Although originally designed for natural language processing tasks, transformers can be used for handling sequential data. Specifically for combinatorial problems like the TSP as they can effectively process sequences of cities, learning the dependencies and context necessary for optimizing travel routes. The main component of transformers is the attention mechanism, specifically Self-Attention, which allows the model to focus on different parts of the input sequence when making predictions. In the context of the TSP, this means a transformer can dynamically decide which cities (nodes) are most relevant when planning the next step in a route. Consequently understanding complex relationships and dependencies between cities, such as distance, cost, or other constraints.

Contrary to a 'normal' attention mechanism as presented by Bahdanau, et al. [2], the Self-Attention mechanism introduced by Kool, et al. [32] allows each element of the input sequence to dynamically focus on other elements within the same sequence to compute its representation. This is done by computing a weighted sum of the other elements, where the weights are determined by a learned similarity measure.

We are using multi-head Self-Attention to capture different aspects of the relationships between tokens. Transformers use heads which are multiple Self-Attention mechanisms to operate in a different subspace of the input representations. Their outputs are concatenated and linearly transformed. This allows the model to learn multiple relationships in the data. Another great quality of transformers is that they are highly scalable because of their architecture and the use of Self-Attention. This makes them well-suited for handling larger datasets and more complex TSP scenarios, where the number of cities and the relationships between them can increase for more complex problem solving.

It is important to note that the Self-Attention mechanism in transformers evaluates the entire input sequence simultaneously, which means the model learns to consider global context and long-range dependencies among cities. This is very important in TSP because the optimal route might depend on understanding the entire network of cities rather than just local neighbors.

3.5 Architectural overview

We've used the paper published by Nazari et al. [44] as inspiration for our implementation, but we have included multiple changes, including the early stopping mechanism, the optimization of the architecture (embeddings and attention mechanisms) as well as a transition from the TensorFlow 1 library to the more robust and flexible PyTorch framework, which we believe improved the model's handling of dynamic data structures and increase computational efficiency.

3.5.1 Dataset

We used the data generator created by Nazari, et al. [44] to generate and maintain the datasets required for the TSP solution testing and training. A single batch of randomly selected training data can be produced using this generator, with each batch being defined by predetermined parameters such batch size and number of cities (nodes). Each data entry within the batch is composed of randomly generated coordinates in a two-dimensional space, effectively simulating various city (node) locations. The generator can also compile datasets that customized for testing or training. These datasets are created according to predetermined criteria, such as the quantity of problems and whether the data will be used for testing or training.

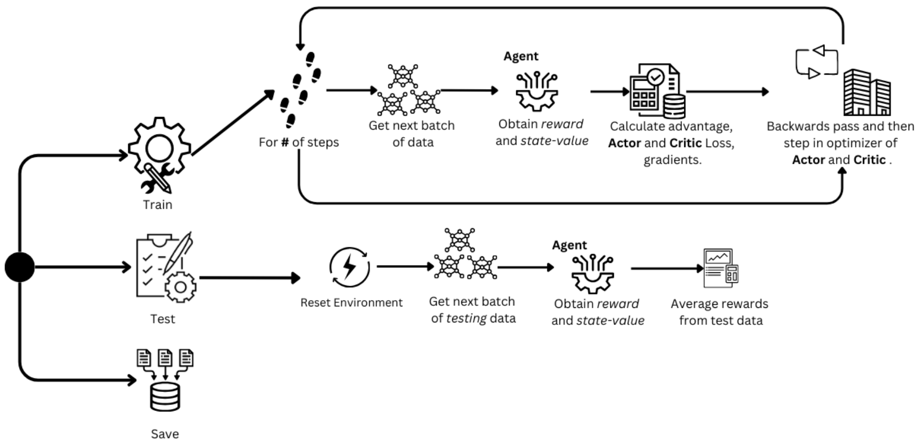
The creation process of the dataset involves options to either generate new data or load existing data for TSP scenarios, this is because we create new random data for the training mode while we load a dataset for the testing stage in order to make sure that the test data remains unchanged across various experimental runs, provided that the same seed and parameters are used.

If a new dataset is required for testing, it is produced with random two-dimensional coordinates for each node across all problem instances. A seed parameter has been incorporated in order to control the randomness, ensuring the reproducibility of data generation and results.

3.5.2 Overview of the Data Flow

Our approach[1] as shown in Figure 3.1 runs through several scenarios, these are training, testing and saving. During training we go through the defined number of training steps or until the early stopping gets activated. In each step of the training loop, the reinforcement learning agent begins by retrieving the next batch of data, which represents different scenarios within the TSP. The agent then interacts with the environment to perform actions and observes the results in terms of both immediate rewards and new state transitions. From these interactions, the agent obtains measurements of rewards and estimates of state-value functions. The rewards provide direct feedback from the environment on the effectiveness of the actions taken, while the state-value functions, generated by the critic, offer an estimate of the expected

[1]This study has relied on approaches that are replicable. Given the documentation and descriptions provided we're certain that the experiments conducted are therefore verifiable by external sources and that the improvements to current frameworks implemented can be reproduced by other researchers for future research. The code developed for this research is available at: `https://github.com/nathsmo/Main-Code/`



Agent
For # of steps
Get next batch of data
Obtain reward and state-value
Calculate advantage, Actor and Critic Loss, gradients.
Backwards pass and then step in optimizer of Actor and Critic .
Train
Agent
Reset Environment
Get next batch of testing data
Obtain reward and state-value
Average rewards from test data
Test
Save

Figure 3.1: This figure outlines our approach, encapsulating the key phases of training, testing, and model saving. During the training phase, the agent retrieves batches of TSP scenarios, interacts with the environment to execute actions, and observes results to gain rewards and state-value estimates. These interactions help the agent calculate advantages and adjust actor and critic models to enhance decision accuracy. Testing intervals allow for performance evaluations without learning, ensuring the policy's effectiveness. Model parameters are periodically saved to secure progress and facilitate future training or deployment in similar TSP settings. The defined procedure for the Agent is shown in Figure 3.2

.

returns from the current state under the agent's policy.

Using the observed rewards and the critic's state-value estimates, the agent calculates the advantage, indicating how much better an action is compared to the baseline performance predicted by the critic. This advantage informs the calculation of losses for both the actor and the critic. The actor loss encourages actions leading to higher than expected returns, and the critic loss is minimized to refine the accuracy of the state-value predictions. Following this, the agent performs a backward pass to compute gradients, applies gradient clipping to stabilize training, and updates the parameters using optimizers for both the actor and critic.

The training process is interrupted periodically to evaluate the agent's performance using test data, as well as to keep record of the steps without improvement for the early stopping mechanism. The evaluation begins with resetting the environment to ensure consistency in testing. The agent processes the test data without performing any learning updates, generating rewards and state-value estimates. The average rewards over the test batch are then calculated to assess the effectiveness of the agent's current policy.

To protect the progress made during training and allow for further analysis or continued training in the future, the agent's model parameters are saved at established intervals. This involves saving the weights of the neural networks that make up the actor and the critic, enabling the model to be reloaded later to either continue training or deploy the trained model in a similar TSP setting.

The agent operates in a series of steps until all nodes in the problem space are visited. Initially, the agent embeds the input using one of four possible embedding techniques, indicated in Figure 3.2 with a star. This embedding is meant to transform data into a representation that captures essential features necessary for decision-making.

Once the input is embedded, the agent resets the environment. This reset is needed at the start of each new episode to ensure the environment is in a standard initial state, free from remains of previous episodes' data. Following the reset, the agent updates its internal mask to keep track of visited nodes, ensuring the route remains valid under TSP constraints.

The core of the agent's decision-making process involves generating action logits, which are probabilities indicating the next best node to visit. This is achieved through either a PN or a Self-Attention mechanism both which have been previously explained, as they are part of our approach they have been indicated by a yellow star symbol too in Figure 3.2.

Using the action logits and the current mask the agent selects its next action. This action selection involves applying a policy, such as ϵ-greedy or stochastic, balancing exploration and exploitation. After completing its route, the agent receives a reward calculated based on the total route created from the chosen actions. The reward typically reflects the efficiency of the route, with shorter routes receiving higher rewards. This reward signals to the agent how the actions and the constructed route performed.

Simultaneously, the critic component of the agent assesses the actions by creating a state-value function result from the last hidden layer of the actor's network. This state-value function estimates the expected return from the current state after taking the action.

Finally, the agent returns both the obtained reward and the estimated state-value. This dual output helps in fine-tuning the agent's decision-making abilities by adjusting the actor and critic networks based on the discrepancies between expected and actual returns, thus enhancing the agent's ability to solve the TSP scenario more efficiently over time.

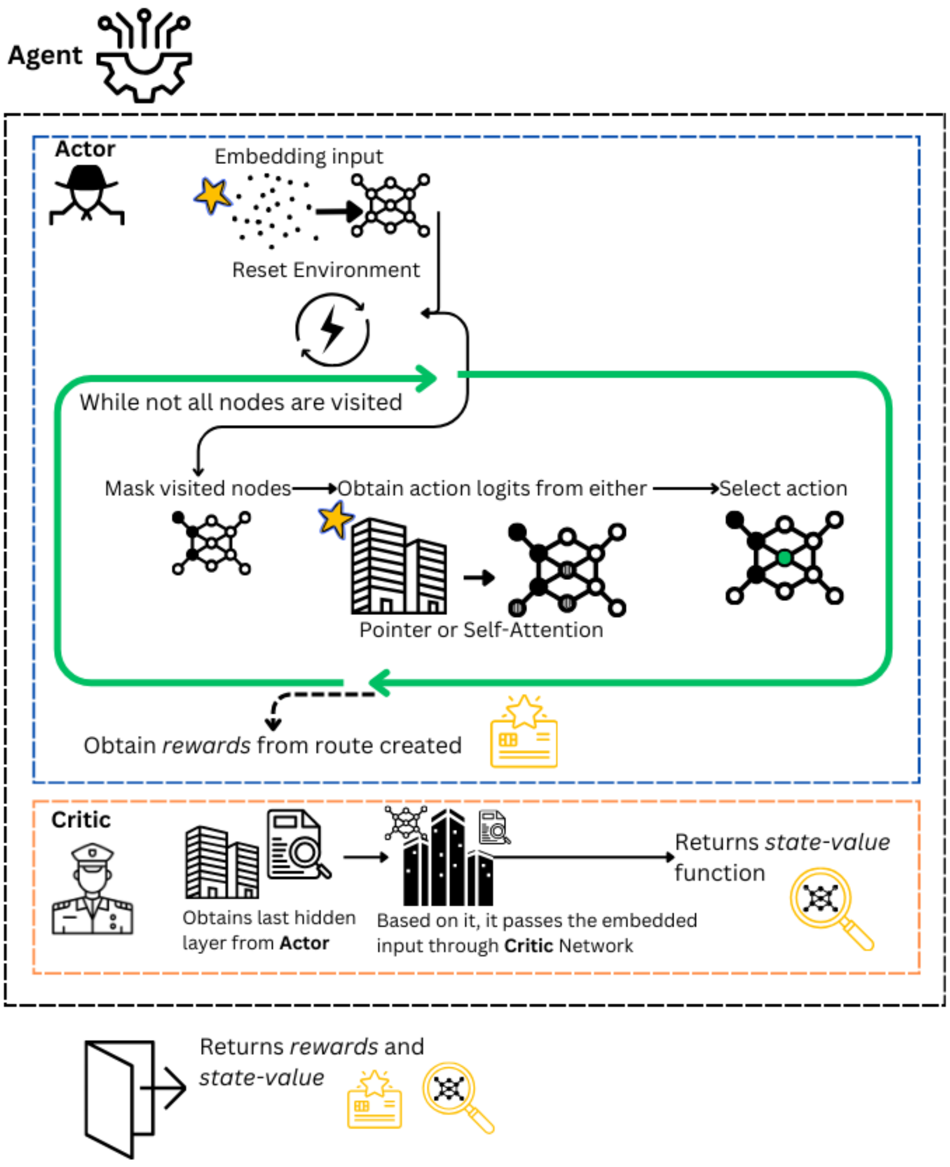

Agent
Actor
Embedding input
Reset Environment
While not all nodes are visited
Mask visited nodes
Obtain action logits from either
Select action
Pointer or Self-Attention
Obtain *rewards* from route created
Critic
Obtains last hidden layer from **Actor**
Based on it, it passes the embedded input through **Critic** Network
Returns *state-value* function
Returns *rewards* and *state-value*

Figure 3.2: Workflow of the Reinforcement Learning Agent implemented in this study. This diagram outlines the sequential steps undertaken by the agent, from input embedding and environment reset to node visitation tracking, decision-making through attention mechanisms, action selection, reward acquisition, and state-value estimation by the critic. The process iterates until all nodes are visited, emphasizing the agent's strategy to optimize route efficiency under TSP constraints. The stars denote the altered parts for this study.

Chapter 4

Experimental setup and results

In this chapter, we go into the experimental setup and procedures used to evaluate the performance of our reinforcement learning agent. The process for the parameter selection is also included here. A series of baseline tests, experimental setups, and result analysis are presented to compare our approach compared to the benchmark models selected in the last sections.

4.1 Baselines and Implementation evaluation

In order to evaluate the performance of our approach, we obtained the performance from external benchmark models that currently adhere to industry standards. They include both exact and heuristic methods. Observing the performance of this benchmarks is important as they are key to placing the performance of our approach within the larger framework of industry standards. The results were generated from three separate software programs well-known for their ability to solve TSP cases: Gurobi Optimizer [26], the IBM ILOG CPLEX Optimization Studio [30] and the Elkai [22] Python library, which implements the LKH 3 algorithm.

The Gurobi Optimizer [26] and IBM ILOG CPLEX Optimization Studio [30] are known for their powerful linear, integer, and quadratic programming capabilities. They were used to build a baseline with its advanced algorithms. Using them for our comparison helps validate our approach efficiency against a leading mathematical optimization solver highly optimized for performance. The third solver implemented was the Elkai Python library which is an implementation of the LKH-3 solver. It's an open-source heuristic solver for the TSP well-known for providing near-optimal solutions to big, difficult routing problems. The Elkai [22] version of LKH-3 used offers a more heuristic benchmark in addition to Gurobi and CPLEX's exact methods. The results of the benchmark models according to the TSP instance size are summarized in Table 4.1.

The evaluation framework of our study and the implemented benchmarks are made up of three key aspects. First, comparing the results of the experiments from this study to those produced by the classical heuristical solvers such as Gurobi and IBM's CPLEX. Secondly, by comparing the optimality gap between them, representing how close the implemented approach can get to the best possible solution under given constraints. Lastly, by comparing the computational time needed for each experiment. We aim to evaluate our approach given these metrics.

Model	TSP 10	TSP 20	TSP 50	TSP 100
CPLEX	2.85	3.84	5.69	7.77
Gurobi	2.85	3.84	5.69	7.77
LKH3	3.50	6.20	15.20	30.24

Table 4.1: Benchmark performance comparison for TSP instances.

4.2 Experimental setup

In our experimental setup, we experimented with several variations to identify the most critical parameters in our model and therefore to obtain the most optimal configurations. Specifically, we investigated the following parameters: the batch size, the epsilon value for action selection, the number of RNN layers, the number of glimpses for the pointer network decoder, and the number of heads for the self-attention decoder. While we know it's not feasible to test every possible parameter. However, we have carefully selected the parameters we believe to be the most influential on the model's performance. Furthermore, we observed a notable impact on the model's performance as a result of this parameter tuning.

In all of our experiments we tested the performance in training and testing of both attention mechanisms (Pointer Network and Self-Attention) with our 4 embeddings for all TSP instance sizes (10, 20, 50 100). For the sake of clarity and convenience, we have included Table 4.2 that summarizes the embeddings discussed in the Section 3.3.

Embedding	Name
1	Simple Linear Embedding
2	Simple Convolutional Embedding
3	Enhanced Linear Embedding with Multiple Layers and Normalization
4	Enhanced Linear Embedding with Additional Complexity

Table 4.2: Summary of the different Embedding types presented in Section 3.3.

4.2.1 Batch size

We started the parameter tuning with the batch size of the model as it can influence the stability and speed of the training process which is very important for our study as, given the amount of tests we must perform we want to make sure that all the parameters tuned can push the model to be the most optimal it can be in a quick amount of time. Adjusting the batch size is also important for managing memory usage, as larger batches use more memory and smaller batches use less. The batch size also affects the model's ability to generalize, as smaller batches can introduce more noise during training, acting as a regularizer. All of the experiments were tested with various batch sizes (64, 128, 256) to determine their impact on the performance and training efficiency of both attention mechanisms with different embeddings in all TSP instance sizes (10, 20, 50 100).

Table 4.3 shows the performance of the algorithm with for TSP 10 instance when implemented with the Pointer Network as the attention mechanism. We can see in Table 4.3 that batch size of 64 with embedding 2 provides the best training and testing results, with a moderate training time. In batch size 128, although embedding 2 has the best training average, embedding 1 achieves the best testing result. For batch size 256, embedding 1 has the best testing result. In terms of training times we can observe from these results that there is a general trend that smaller batch sizes (64) tend to have shorter training times but slightly higher variability in results compared to larger batch sizes (256).

For this experiment embedding 2 achieved the best average training result (3.95) in batch size 128 but there was a 0.15 reward reduction in embedding 1 showing that maybe a batch size large can generalize better. We also obtained a near-optimal testing result (3.85) in this same a batch size, coming very close to the benchmark model of LKH-3 as seen in Table 4.1.

Batch size of 128 produces competitive training and testing results with less variability than the other batch sizes. For example, the gap between results in batch size 64 can vary from .20 to .40 points while in batch 128 they vary from 0.08 to 0.35.

Decoder	Pointer Network				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	4.43	**3.69**	4.33	4.20
	Test R.	4.00	**3.86**	4.64	3.82
	Training Time	0:05:06	0:04:30	0:04:28	0:04:45
128	Train R. (avg)	4.30	**3.95**	4.44	4.22
	Test R.	**3.85**	3.88	4.60	4.05
	Training Time	0:06:09	0:06:02	0:06:37	0:06:08
256	Train R. (avg)	4.47	**4.20**	4.54	4.25
	Test R.	**3.91**	3.95	4.53	4.06
	Training Time	0:10:55	0:10:13	0:10:59	0:10:43

Table 4.3: Training and Testing Results for TSP 10 with Various Embeddings and Batch Sizes for Pointer Network as attention mechanism.

Decoder	Self Attention				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	4.32	**4.28**	4.34	4.31
	Test R.	4.85	**4.81**	4.85	4.86
	Training Time	0:08:33	0:07:12	0:08:30	0:08:27
128	Train R. (avg)	**4.40**	4.44	4.47	4.41
	Test R.	4.67	4.64	4.61	**4.59**
	Training Time	0:09:30	0:09:55	0:07:46	0:08:03
256	Train R. (avg)	4.52	4.53	4.55	**4.49**
	Test R.	4.64	**4.58**	4.61	4.64
	Training Time	0:10:34	0:10:19	0:09:13	0:10:48

Table 4.4: Training and Testing Results for TSP 10 with Various Embeddings and Batch Sizes for Self Attention as attention mechanism.

The training times for a batch size of 64 are also shorter compared to larger batch sizes, almost all training in under 5 minutes, significantly less than the training times observed for batch sizes of 128 and 256. We chose batch size of 128 to continue the experiments for the TSP 10 with Pointer Network as it showed consistent performance, a moderate training time efficiency, and a consistent model generalization.

Table 4.4 shows the performance of the algorithm with the Self-Attention as the attention mechanism for the TSP 10 instance. The average training results for batch size 256 show the lowest reward obtained during testing, specifically when using embedding 2, this embedding also has the best results of the model when using batch size 64. From the general results obtained between all batch sizes we can observe a lower variability in results obtained using batch size 256.

From test results with batch size 256, embedding 2 achieves the best testing result (4.58), followed by batch size 128 with embedding 4. When comparing the results of the other batch sizes, batch size 256 consistently provides better a more stable and better testing performance. The training times for batch size 256 are the highest

Decoder	Pointer Network				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	9.13	**7.31**	9.47	8.50
	Test R.	7.73	**7.67**	9.73	7.81
	Training Time	0:08:04	0:09:11	0:06:29	0:10:27
128	Train R. (avg)	8.58	**7.75**	9.50	8.38
	Test R.	7.51	**7.49**	9.83	7.80
	Training Time	0:11:54	0:12:27	0:16:03	0:13:35
256	Train R. (avg)	9.24	**8.76**	8.90	8.90
	Test R.	7.67	**7.65**	9.08	8.20
	Training Time	0:25:34	0:19:33	0:22:32	0:22:32

Table 4.5: Training and Testing Results for TSP 20 with Various Embeddings and Batch Sizes for Pointer Network as attention mechanism.

from all the tested batch sizes but yet they are still manageable. While batch size 64 sometimes shows shorter training times (e.g., 2 at 0:07:12), it does not significantly outperform in terms of test results, making batch size 256 a better trade-off between time and performance.

Seeing its effectiveness in producing a model that generalizes well to unseen data we decided to keep proceed with batch 256 for the experiments of the TSP 10 instance that use the Self-Attention as the attention mechanism.

Table 4.5 shows the results of the different batch sizes implemented for the TSP 20 instance using the Pointer Network as the attention mechanism. From this table we can observe that batch size might not influence as much the results as the type of embedding applied to it. It seems that embedding 1 produces a gap from 1.04 to 1.57 between the training and the testing rewards. While all other embeddings don't produce as big gaps between the testing and the training results. The batch size nonetheless does change the training time for the model, as small batch sizes (64) have lower training times than bigger batch sizes (256), pattern that gets inversed when using Self-Attention as the attention mechanism as we'll see in a moment.

Seeing as the results of the individual embeddings don't vary a lot our choice for the batch size was guided by the training time and the better testing performance obtained with that time, therefore choosing the proceed with a batch size of 128 for the TSP 10 instance using Pointer Network as the attention mechanism.

Table 4.6 shows the results of the different batch sizes implemented with the Self-Attention as the attention mechanism for the TSP 20 instance. As we noted before it seems that when using the Self-Attention as the attention mechanism the batch size influences considerably the training time as small batch sizes (64) have higher training times than bigger batch sizes (256).

Decoder	Self Attention				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	**9.31**	9.38	9.43	9.39
	Test R.	8.80	8.89	8.79	**8.77**
	Training Time	0:34:19	0:42:37	0:32:00	0:33:15
128	Train R. (avg)	9.48	9.47	**9.41**	9.49
	Test R.	7.78	**6.51**	7.83	6.80
	Training Time	0:19:39	0:19:04	0:19:33	0:17:14
256	Train R. (avg)	**9.58**	9.60	9.64	9.62
	Test R.	6.67	**6.45**	8.38	8.20
	Training Time	0:15:32	0:18:58	0:13:39	0:12:30

Table 4.6: Training and Testing Results for TSP 10 with Various Embeddings and Batch Sizes for Self Attention as attention mechanism.

We can also see in the table that two out of the three batch sizes' best results come from embedding 2. Both in batch size 128 and 256 they get very close to the benchmark model LKH-3 as seen in Table 4.1. Which we'll discuss in later sections.

The variablility shown in the results of Table 4.6 show that all testing results are considerably lower than the training reward obtained. Specifically batch size 128 and 256 seem to have 2 out of 4 embeddings with results very close to an optimal result. Seeing as the lowest training time and best results can be found in batch size 256 we decided to implement size 256 as the optimal batch size to continue testing for TSP 20 instance with Self-Attention as the attention mechanism.

Table 4.7 shows the results from the TSP 50 instance using the Pointer Network as the attention mechanism. In order to determine the most optimal batch size to use in this model configuration we took a look at the training times from each batch size implemented. As we can see the bigger the batch size implemented in the model that used Pointer Network, the higher the training time becomes.

From Table 4.7 we can also see a very obvious behaviour, as embedding 2 always obtains the most optimal results. No matter the batch size the testing results seem to have very little difference between them, what does have a great difference specifically for this embedding (2) is that the bigger the batch size, the higher the training rewards. Embeddings 3 and 4 seem to also have very consistent training and testing results across all batch sizes. Embedding 1 is the only one that breaks the pattern.

From the results obtained in Table 4.7 we decided that the most optimal implementation for the TSP 50 instance with Pointer Network as the attention mechanism is the batch size 64. This choice balances good performance metrics with the testing results and lower training times.

Table 4.8 shows the implementation of the TSP 50 instance with Self-Attention as

Decoder	Pointer Network				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	21.43	**16.94**	23.81	22.61
	Test R.	18.77	**18.25**	24.20	19.75
	Training Time	0:23:19	0:21:37	0:26:12	0:27:39
128	Train R. (avg)	21.32	**17.33**	23.79	22.61
	Test R.	19.37	**18.57**	23.40	19.58
	Training Time	0:46:09	0:48:56	0:43:54	0:54:23
256	Train R. (avg)	22.20	**19.72**	24.21	22.08
	Test R.	18.40	**18.30**	23.88	19.81
	Training Time	1:22:10	1:45:34	1:39:30	1:13:34

Table 4.7: Training and Testing Results for TSP 50 with Various Embeddings and Batch Sizes for Pointer Network as attention mechanism.

Decoder	Self Attention				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	24.67	24.71	**24.56**	24.67
	Test R.	19.97	20.08	**19.69**	20.92
	Training Time	0:57:03	0:57:42	0:52:10	0:53:42
128	Train R. (avg)	24.90	**24.86**	24.93	24.92
	Test R.	**19.27**	19.45	19.55	19.29
	Training Time	1:05:01	1:29:09	1:22:34	1:29:40
256	Train R. (avg)	**25.06**	25.09	25.12	25.16
	Test R.	19.48	19.47	**19.44**	19.61
	Training Time	1:52:08	1:56:14	1:34:36	1:34:11

Table 4.8: Training and Testing Results for TSP 50 with Various Embeddings and Batch Sizes for Self Attention as attention mechanism.

Decoder	Pointer Network				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	50.12	**49.87**	51.34	50.76
	Test R.	46.55	**38.34**	44.85	48.17
	Training Time	0:34:23	0:46:25	0:59:32	0:59:05
128	Train R. (avg)	46.18	**32.11**	46.74	46.85
	Test R.	36.25	**32.41**	44.70	43.07
	Training Time	2:39:50	1:55:35	3:02:46	2:59:45
256	Train R. (avg)	43.70	**35.46**	49.29	46.28
	Test R.	36.75	**35.15**	44.27	41.60
	Training Time	3:16:09	4:44:11	3:40:37	5:13:10

Table 4.9: Training and Testing Results for TSP 100 with Various Embeddings and Batch Sizes for Pointer Network as attention mechanism.

Decoder	Self Attention				
Batch Size	Embedding	1	2	3	4
64	Train R. (avg)	**49.51**	50.28	53.44	51.23
	Test R.	**46.55**	48.34	46.85	48.17
	Training Time	2:03:53	1:53:25	1:54:20	1:30:20
128	Train R. (avg)	50.76	**50.58**	50.68	50.73
	Test R.	45.52	45.59	45.42	**45.29**
	Training Time	2:16:48	2:25:30	2:29:43	2:48:27
256	Train R. (avg)	51.03	**51.01**	50.93	51.06
	Test R.	45.78	45.65	45.67	**44.68**
	Training Time	2:55:08	3:22:53	3:35:03	4:34:25

Table 4.10: Training and Testing Results for TSP 100 with Various Embeddings and Batch Sizes for Self Attention as attention mechanism.

the attention mechanism. In this table we see the same pattern as in when the model is implemented with Pointer Network as a decoder. As the batch size increases, so does the training time. From the results of the table se can see that the training time more than doubled between batch size 64 and batch size 256.

From the training performance in batch size 64 we can see that all embeddings show similar performance (around 24.56 to 24.71). This is a pattern that also follows in batch 128 (around 24.86 to 24.93), and batch 256 (around 25.06 to 25.16). The best testing result comes from the implementation of the model with batch size 128 with embedding 1 with a score of 19.24, followed twice by embedding 3 that has the best results in batch size 64 and batch size 256. The training times also mostly remain consistent throughout the different batch sizes.

The best testing scores as a group appear to be from batch size 128, and having observed that the other parameters remain consistent across batch sizes we have chosen batch size 128 to continue experimenting the TSP 50 instance with Self-Attention as the attention mechanism.

Table 4.9 shows the implementation for the TSP 100 instance with the Pointer Network as the attention mechanism. From the table we can see that the performance does tend to get better as the batch size increases, this can be seen constantly in all of the embeddings implemented. The time for training also increases as the batch size increases, which is not optimal if we want a balance between time for training and results.

Batch size 128 with embedding 2 shows the best balance of performance and training time. Even though the results do change as the batch size increases, we do see a bigger jump from the results found in batch size 64 to the results in batch 128, especially for embeddings 1 and 2. Given that the improvement in performance with batch size 256 is marginal it doesn't justify the significantly longer training times. Considering the balance between performance and training efficiency, we have chosen to continue training with batch 128 for the TSP 100 instance when using the Pointer Network as the attention mechanism.

Table 4.10 shows the results for the implementation of Self-Attention as the attention mechanism for the TSP 100 instance. There seems to be a similar performance across all embeddings and batch sizes. The best result from batch size 64 is from embedding 1 with 49.51 during training and 46.55 during testing. The best results in all other batch sizes seem to be in training for embedding 2 and for testing for embedding 4.

The training time seems to constantly grow as the batch size also increases. but the testing performance overall happens in batch size 128 were all the embedding had similar performance (45.29 to 45.59). Batch size 256 had significantly longer training times, with 1 being the shortest (2:55:08) and 4 the longest (4:34:25).

Observing that batch size 128 shows similar performance to batch size 256 but with significantly shorter training times. This marginal improvement in performance with batch size 256 doesn't justify the substantial increase in training time, and so therefore we can conclude that batch size 128 offers good efficiency with the best performance and shorter training times for the next experiments for TSP instance size 100 with Self-Attention as the attention mechanism.

We present a summary of the chosen the batch sizes per TSP instance size in Table 4.11.

	Batch size			
Model	TSP 10	TSP 20	TSP 50	TSP 100
Pointer Network	128	128	64	128
Self-Attention	256	256	128	128

Table 4.11: Summary of the optimal batch sizes for the TSP instance sizes implemented.

4.2.2 Epsilon

The epsilon variable is a parameter that measures the exploration vs exploitation factor in learning algorithms as we saw in Section 3.2. In order to avoid overfitting, we want to select an epsilon value attempting to balance the model's capacity to generalize from training data to testing data. A higher epsilon value promotes greater exploration in the early phases of training, enabling the model to attempt a wide range of actions and collect a variety of outcomes. An epsilon value of 0 means that the RL agent always exploits the best possible action.

We hope that by having a moderate exploration value the model is able to keep away of local optima and learn more about the environment around it. In our implementation we gradually lower the epsilon as training continues encouraging the agent to use the learnt policy to improve and maximize performance. The epsilon parameter makes sure that the model efficiently learns the optimal policy by applying both the exploration of new actions and the exploitation of known actions.

We have implemented 3 epsilon values in our experiments (0-no exploration, 0.1-moderate exploration, 0.3-high exploration). As we have selected the batch size previously, from this point onwards we utilized the selected batch size as seen in the summary Table 4.11.

In Table 4.12 we can see the performance of the model with different epsilon values for the implementation of the Pointer Network as the attention mechanism for TSP 10 instance. We can observe that embedding 2 always performs the best during the training but as the epsilon increases embedding 4 beats embedding 2 during the testing of the model. The training times of all three implementations with different epsilon show similar training time ranging from as little as 2:24 to 6:33 minutes.

The performance presented on Table 4.12 demonstrates that for the TSP 10 instance, as the epsilon increases so does the testing score, performing worse than the experiments with higher epsilon values. Given that the training time remains consistent through the epsilon values and the the performance worsens we have chosen to opt for the epsilon value 0 for the TSP 10 instance with the Pointer Network, therefore

Decoder	Pointer Network				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	4.43	**3.69**	4.34	4.20
	Test R.	3.90	**3.70**	4.24	3.79
	Training T.	0:05:02	0:04:48	0:06:33	0:05:08
0.1	Train R. (avg)	4.43	**3.69**	4.34	4.20
	Test R.	4.01	3.86	4.65	**3.82**
	Training T.	0:05:06	0:05:23	0:06:00	0:05:18
0.3	Train R. (avg)	4.43	**3.69**	4.34	4.20
	Test R.	4.10	4.00	4.55	**3.97**
	Training T.	0:03:57	0:04:48	0:06:10	0:02:24

Table 4.12: Training and Testing Results for TSP 10 with Various Embeddings and Epsilon Values for Pointer Network as attention mechanism

Decoder	Self Attention				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	**4.40**	4.44	4.47	4.41
	Test R.	4.71	4.67	4.68	**4.59**
	Training T.	0:08:24	0:09:35	0:07:26	0:08:05
0.1	Train R. (avg)	**4.40**	4.44	4.47	4.41
	Test R.	4.67	4.64	4.61	**4.59**
	Training T.	0:08:28	0:11:07	0:07:27	0:07:24
0.3	Train R. (avg)	**4.40**	4.44	4.47	4.41
	Test R.	4.62	4.53	4.55	**4.50**
	Training T.	0:08:06	0:10:14	0:06:26	0:07:08

Table 4.13: Training and Testing Results for TSP 10 with Various Embeddings and Epsilon Values for Self Attention as attention mechanism

completely prioritizing the exploitation of known actions rather than exploration.

Table 4.13 shows the performance of the TSP 10 instance when using the Self-Attention as the attention mechanism. From the performance it appears that all results appear constant no matter the epsilon value applied. There is therefore a common trend where embedding 1 obtains the best result during the training while embedding 4 obtains the best result during the testing.

The testing results do vary from the different epsilon values applied, we can even notice that the test performance is actually higher as the epsilon value is greater. This shows that specifically for the implementation of the Self-Attention as the attention mechanism in the model, this attention mechanism actually performs better when it explores beyond the know best value. As for the training time, it has a downwards trend as the epsilon value increases.

From this results we can conclude that the optimal choice for the epsilon value for the TSP 10 instance with the Self-Attention mechanism as the attention mechanism is the 0.3 value, as it enhances the most the performance of the model.

Decoder	Pointer Network				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	8.46	**7.29**	9.51	7.63
	Test R.	7.18	**7.14**	8.40	7.57
	Training T.	0:13:58	0:13:16	0:14:35	0:14:00
0.1	Train R. (avg)	8.46	**7.29**	9.51	7.63
	Test R.	7.67	**7.44**	9.07	8.04
	Training T.	0:15:54	0:15:26	0:19:38	0:20:30
0.3	Train R. (avg)	8.46	**7.29**	9.51	7.63
	Test R.	8.33	**8.23**	9.65	8.70
	Training T.	0:22:02	0:21:23	0:23:35	0:22:44

Table 4.14: Training and Testing Results for TSP 20 with Various Embeddings and Epsilon Values for Pointer Network as attention mechanism

On Table 4.14 we present the performance of the model for TSP 20 instance when using the Pointer Network as the attention mechanism. In the table we can see that the training rewards again seem to remain constant across all epsilon values implemented. Specifically, we see that embedding 2 consistently outperforms other embeddings in both training and testing performance across all epsilon values. We must note that have been seeing this pattern for the implementation with Pointer Network as the attention mechanism from past experiments, this could indicate that the 2 embedding is particularly effective for the Pointer Network in solving TSP instances.

From table 4.14 we observe that as the epsilon value increases, there is a noticeable decline in testing performance, indicating that higher epsilon values might negatively impact the generalization ability of the model. The training times also increase with higher epsilon values. This could be due to the model requiring more iterations or adjustments to converge when epsilon values are higher, therefore the early stopping mechanism won't be used as the model slowly reaches the best performance.

We can notice that embedding 1 and 3 have consistently higher training and testing errors compared to 2 and 4, showing that they may not be as suitable for this problem or model configuration. We will talk more about this in the later sections of this study. We also note that ehe differences in performance between embeddings are more pronounced at lower epsilon values, with embedding 2 significantly outperforming others. At higher epsilon values, performance differences between embeddings become less different.

From all this we can conclude that the optimal epsilon value for the implementation of the TSP 10 instance with Pointer Network as the attention mechanism appears to be 0, particularly with the 2 embedding, as it provides the best balance of best training and testing results with relatively shorter training times.

Decoder	Self Attention				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	**9.47**	9.54	9.48	9.52
	Test R.	8.78	**8.76**	8.84	8.90
	Training T.	0:19:03	0:19:57	0:20:06	0:21:33
0.1	Train R. (avg)	**9.47**	9.54	9.48	9.52
	Test R.	**7.75**	7.76	7.80	7.88
	Training T.	0:16:01	0:16:36	0:16:20	0:14:48
0.3	Train R. (avg)	**9.47**	9.54	9.48	9.52
	Test R.	**6.57**	6.68	7.55	7.60
	Training T.	0:14:06	0:14:56	0:13:55	0:11:04

Table 4.15: Training and Testing Results for TSP 20 with Various Embeddings and Epsilon Values for Self Attention as attention mechanism

On Table 4.15 we can see the performance of the model for TSP 20 instance with Self-Attention as the attention mechanism. We can observe that across all epsilon values, the performance metrics for training and testing are quite similar among the different embeddings. This could be because no single embedding drastically outperforms the others for the Self-Attention mechanism on the TSP 20 instance.

We can observe that as the epsilon value increases from 0 to 0.3, there is a slight but consistent decrease in the testing performance, indicating that a higher epsilon value might help the model generalize better, specifically for the Self-Attention mechanism. This is a pattern we've seen in previous tables.

In the training times we see a decrease as epsilon values increase. This suggests that higher epsilon values may allow the model to converge faster during training, potentially making them more efficient in terms of computational resources and time. Embedding 2 shows shorter training times compared to other embeddings, indicating it might be more computationally efficient. The optimal configuration in terms of testing performance seems to be at epsilon 0.3 with the 4 embedding, which has the lowest testing value. For training performance, all embeddings perform similarly with no significant difference across epsilon values.

In Table 4.16 we can see the performance of the model for the TSP 50 instance when using the Pointer Network as the attention mechanism. In this table we can see that the training times increase with higher epsilon values. This repeats itself as we also saw with TSP 20 instance. From the training performance we see that embedding 2 embedding consistently shows the best training performance across all epsilon values, achieving the lowest training values (16.90 to 17.05). As for the testing we see that for epsilon value 0, embedding 2 obtains the best testing performance with a value of 17.06. For epsilon values 0.1 and 0.3, embedding 1 obtains the best testing values

Decoder	Pointer Network				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	18.77	**17.05**	22.91	20.41
	Test R.	20.49	**17.06**	21.79	25.93
	Training T.	0:29:59	0:22:08	0:25:39	0:29:42
0.1	Train R. (avg)	19.03	**17.04**	23.67	19.02
	Test R.	**18.36**	18.79	19.81	20.12
	Training T.	0:48:29	0:51:17	0:59:28	0:35:11
0.3	Train R. (avg)	19.03	**16.90**	22.91	20.41
	Test R.	**20.86**	21.38	23.57	22.56
	Training T.	1:08:55	1:07:20	1:14:30	0:53:55

Table 4.16: Training and Testing Results for TSP 50 with Various Embeddings and Epsilon Values for Pointer Network as attention mechanism

Decoder	Self Attention				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	24.55	24.53	**24.48**	24.67
	Test R.	19.84	**19.80**	20.01	20.12
	Training T.	0:35:21	0:54:34	0:41:55	0:31:34
0.1	Train R. (avg)	24.55	24.53	**24.48**	24.67
	Test R.	20.11	**19.85**	20.00	20.10
	Training T.	0:32:32	0:36:23	0:30:28	0:30:08
0.3	Train R. (avg)	24.55	24.53	**24.48**	24.67
	Test R.	19.95	19.88	**19.60**	19.85
	Training T.	0:20:16	0:17:16	0:18:49	0:18:04

Table 4.17: Training and Testing Results for TSP 50 with Various Embeddings and Epsilon Values for Pointer Network as attention mechanism

(18.36 and 20.86 respectively).

From the general performance we can see that the change in the epsilon value signal that a moderate exploration value (epsilon 0.1) obtains the best results for all the embeddings. The training time increases as the epsilon increases. we can also note from the table that there is high consistency in training performance across all embeddings from all epsilon values, perhaps indicating robustness to changes in epsilon during training.

From all this we opted to choose the optimal configuration in terms of both training and testing performance with an epsilon value of 0. While it might not be the most efficient parameter in terms of training time it does appear to set a balance that produces better results than the alternative epsilon values. Therefore concluding that the epsilon value 0 is the chosen value for the TSP 50 instance that uses the Pointer Network as the attention mechanism.

On Table 4.17 we observe the implementation of the model when using the Self-Attention as the attention mechanism for the TSP 50 instance and experimenting with the epsilon values. From this table we can note that the training performance remains consistent across different epsilon values, showing that the use of the Self-Attention decoder leads to stable results when training, even with changes in the epsilon. From this stable results we can say that embedding 3 consistently shows the best training performance across all epsilon values, achieving the lowest training result (24.48).

On the other hand we can see that the testing results show a different story as the results from the implementation of epsilon values 0 and 0.1 show that embedding 2 has the best testing performance, but at epsilon 0.3, 3 achieves the best result (19.60), showing it adapts well to a higher epsilon. It also shows that as much as we obtain stable results when training, the that the epsilon value does affect the network architecture when seeing untested data. We can also observe that as epsilon increases, there is a general trend of improved testing performance, this can be particularly seeing with embedding 3 at epsilon 0.3. Suggesting that higher epsilon values might help the model generalize better.

As for the training times we can see that they significantly vary among the embeddings. Embedding 4 generally has the shortest training times across all epsilon values but the general trend is that the higher the epsilon values the shorter the model takes to train. For example, at epsilon 0.3, training times are the shortest across all embeddings, which could be due to faster convergence and therefore a faster activation of the early stopping mechanism implemented.

Given that this epsilon value achieves the best testing performance with relatively efficient training times, we can assume that the implementation for the TSP 50 instance with Self-Attention as the attention mechanism is most efficient with a higher exploration value, therefore using epsilon value 0.3.

Table 4.18 show the performance of the model when using the Pointer Network as the attention mechanism for the TSP 100 instance. from the results of the table we can see that the testing performance shows more variability, particularly with higher epsilon values. But the training performance remains constant across different epsilon values for all embeddings, suggesting that the training process is robust to changes in epsilon.

The training times generally increase with higher epsilon values, showing it in the results of all the implemented embeddings. From the testing results we can see that as the epsilon value increases, there is a noticeable deterioration in testing performance for all embeddings also. For example, the testing score for embedding 2 increases from 31.84 at epsilon 0 to 38.74 at epsilon 0.3, suggesting that higher epsilon values might negatively impact generalization specifically for when implementing the Pointer

Decoder	Pointer Network				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	36.63	**32.59**	45.81	37.01
	Test R.	32.66	**31.84**	40.00	34.81
	Training T.	2:01:09	1:51:06	2:26:09	2:53:01
0.1	Train R. (avg)	36.63	**32.59**	45.81	37.01
	Test R.	36.87	**32.04**	44.94	40.88
	Training T.	2:39:23	1:55:03	3:02:08	2:56:00
0.3	Train R. (avg)	36.63	**32.59**	45.81	37.01
	Test R.	42.84	**38.74**	49.19	45.01
	Training T.	2:30:33	2:04:00	3:09:34	3:04:17

Table 4.18: Training and Testing Results for TSP 100 with Various Embeddings and Epsilon Values for Pointer Network as attention mechanism

Decoder	Self Attention				
Epsilon	Embedding	1	2	3	4
0	Train R. (avg)	50.71	**50.58**	50.68	50.76
	Test R.	**45.03**	45.58	45.60	45.18
	Training T.	2:05:42	2:03:07	2:04:40	2:03:39
0.1	Train R. (avg)	50.71	**50.58**	50.68	50.76
	Test R.	**45.08**	45.64	45.60	45.21
	Training T.	2:07:28	2:07:07	2:15:14	2:31:08
0.3	Train R. (avg)	50.71	**50.58**	50.68	50.76
	Test R.	**45.19**	45.55	45.51	45.24
	Training T.	2:09:23	2:20:07	2:31:44	3:01:22

Table 4.19: Training and Testing Results for TSP 100 with Various Embeddings and Epsilon Values for Self Attention as attention mechanism

Network as the attention mechanism.

Embedding 2 consistently achieves the best training performance across all epsilon values, with the lowest training value (32.59) and also the lowest testing value with scores ranging from 31.84 to 38.74. This shows that embedding 2 can be very effective in both training and generalization. This is a pattern that we have also been seeing in other TSP instances of the study.

Given all of this we consider that the optimal configuration in terms of both training and testing performance appears to be the epsilon value of 0. As the implementation of the Pointer Network as the attention mechanism seems to obtain its best scores when it exploits the known best actions in the model. This epsilon also has the lowest training time which is advantageous when dealing with an instance size such as this one.

On Table 4.19 we can see the results of the implementation of the Self-Attention as the attention mechanism for the TSP 100 instance. From the highlighted results its obvious that embedding 2 outperforms other embeddings in terms of training errors,

	Epsilon value			
Model	TSP 10	TSP 20	TSP 50	TSP 100
Pointer Network	0	0	0	0
Self-Attention	0.3	0.3	0.3	0

Table 4.20: Summary of the epsilon results for all TSP instance sizes.

making it a strong candidate for the Self-Attention mechanism. But that embedding 1 shows the best performance in testing errors, showing the potential of the embedding when applied to the model.

From the training performance we can say that it remains constant across different epsilon values for all embeddings. And also as we've seen before that its only the testing performance that changes, showing small variations as the epsilon value increases. We've seen this pattern in other TSP instances. While the testing values remain relatively stable across different epsilon values there does seen to be a detereorating effect on some embedding such as embedding 2 and 4, but the change is very small to be impactful. The optimal configuration in terms of both training and testing performance appears to be the the epsilon value of 0. As this value achieves a strong training performance with the most efficient training times.

In Table 4.20 we summarize the chosen parameters for the epsilon value across all the TSP instances implemented in this study.

4.2.3 RNN layers

The number of RNN layers can significantly impact the model's ability to capture complex temporal dependencies. Adding more layers can help the model learn intricate sequential data, making it better suited for handling complex tasks. However, this comes with potential drawbacks. More layers can lead to overfitting, longer training times and higher resource consumption. In this section we go through the process of finding the right balance of the amount of RNN layers in the model.

The anlysis is separated by the type of attention mechanim used. In the use of the Pointer Network as the attention mechanism we present the analysis of the number of RNN layers and the number of glimpses. For the use of Self-Attention as the attention mechanism we present the analysis of the number of RNN layers with a number of heads. The subsequent experiments are designed and conducted based on the optimal configurations selected from the prior experimental results.

	Pointer Network				
RNN	Embedding	1	2	3	4
1	Train R. (avg)	4.43	**3.69**	4.34	4.20
	Test R.	3.90	**3.70**	4.24	3.79
	Training T.	0:05:09	0:04:39	0:05:28	0:04:50
2	Train R. (avg)	4.39	**3.73**	4.35	4.01
	Test R.	**3.78**	3.83	4.77	3.90
	Training T.	0:06:23	0:06:01	0:07:09	0:06:42

Table 4.21: Training and Testing Results for TSP 10 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with no additional glimpses.

4.2.4 Glimpses for the Pointer Network

Similarly to the RNN layers the number of glimpses can affect the model's capacity to focus on different parts of the input sequence. Therefore optimizing this parameter is very important as it can augment the model's accuracy by ensuring it adequately attends to relevant input features.

On Table 4.21 we can see the performance of the model when using the Pointer Network as the attention mechanism for the TSP 10 instance with no additional glimpses (0). From the table we can observe that embedding 2 consistently achieves the best training performance across both RNN configurations, with the lowest training reward (3.69 and 3.73). The performance during testing does change between embedding 2 and 1 which suggest that simpler embeddings are more suitable for this configuration of the model.

We can notice that the number of RNN does provoke an increase in the values during testing. Training times also increase with higher RNN layers across all embeddings. This trend shows that more complex RNN configurations require more computational time for convergence. The optimal configuration given this factors is the use of a single RNN layer when using no glimpses with the Pointer Network as the attention mechanism in the TSP 10 instance.

On Table 4.22 we present the results of the model with the use of the Pointer Network as the attention mechanism for the TSP 10 instance with 3 additional glimpses. Again we can see that for the training performance embedding 2 consistently achieves the best results no matter the RNN layer number. But that it is embedding 1 that obtains the best results during testing.

There does appear to be a slight impact again with the training times, as they generally increase with more complex RNN configurations across all embeddings. Still embedding 2 consistently has shorter training times compared to other embeddings,

	Pointer Network				
RNN	Embedding	1	2	3	4
1	Train R. (avg)	4.23	**3.78**	4.40	4.22
	Test R.	**3.75**	3.90	4.59	3.99
	Training T.	0:09:03	0:04:40	0:09:56	0:08:50
2	Train R. (avg)	4.32	**3.89**	4.32	4.12
	Test R.	**3.70**	3.75	4.65	4.07
	Training T.	0:10:57	0:09:19	0:10:47	0:11:01

Table 4.22: Training and Testing Results for TSP 10 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with 3 additional glimpses.

making it more efficient in terms of computational resources. Contrary to the training performance the testing results become better with an added RNN layer but at slightly longer training time.

The performance therefore suggests that the optimal configuration in terms of both training and testing performance is the use of 2 RNN layers is slightly more effective for generalization with this configuration of the TSP 10 instance with Pointer Network with 3 glimpses.

On Table 4.23 we can see the performance of the model when using the Pointer Network as the attention mechanism for the TSP 20 instance with no additional glimpses (0). We again can notice the slight increase in training time with the additional RNN layer across all embedding's results. For both the training and testing performance the table shows how embedding 2 consistently achieves the best results across both RNN configurations. From the training performance we can observe that while the performance does increase there is a noticeable improvement in the testing results, showing that the use of 2 RNN layers is more effective for generalization of this instance.

For this experiment also embedding 1 has shorter training times compared to other embeddings, but it doesn't perform as well in terms of training and testing results. The optimal configuration in terms of both training and testing performance is therefore the use of 2 RNN layers when employing the Pointer Network as the attention mechanism with no glimpses for the TSP 10 instance, as the slight increase in training time is compensated by the enhancement of the model scores.

On Table 4.24 we present the results of the model with the use of the Pointer Network as the attention mechanism for the TSP 20 instance with 3 additional glimpses. We see again in this table the same pattern as in Table 4.23 as embedding 2 achieves the best training and testing performance across both RNN configurations. We can also notice again that the training times generally increase with additional RNN layers.

	Pointer Network				
RNN	Embedding	1	2	3	4
1	Train R. (avg)	7.62	**7.23**	9.30	7.67
	Test R.	7.19	**6.85**	8.63	7.81
	Training T.	0:08:30	0:10:38	0:11:07	0:09:11
2	Train R. (avg)	7.79	**7.28**	9.37	7.70
	Test R.	7.02	**6.65**	9.97	7.63
	Training T.	0:11:16	0:12:28	0:14:23	0:10:48

Table 4.23: Training and Testing Results for TSP 20 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with no additional glimpses.

	Pointer Network				
RNN	Embedding	1	2	3	4
1	Train R. (avg)	8.02	**7.34**	8.84	8.58
	Test R.	7.34	**7.13**	8.99	8.07
	Training T.	0:18:02	0:19:48	0:20:04	0:19:39
2	Train R. (avg)	7.52	**7.15**	9.35	7.64
	Test R.	7.59	**6.87**	9.39	7.32
	Training T.	0:19:09	0:25:45	0:21:56	0:23:30

Table 4.24: Training and Testing Results for TSP 20 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with 3 additional glimpses.

From the results we can see that the training performance for embedding 2 improves slightly from 7.34 to 7.15 and in testing from 7.13 to 6.87, a patter that is only repeated with embedding 4 but with results that are slightly less favorable. Embeddings 1 and 3 seem to get worse results when the RNN layers are increased.

The optimal configuration for this TSP 20 instance with Pointer network as the attention mechanism and the use of 3 additional glimpses was more in favour of the marginally better results from embeddings 2 and 4 as the results suggest that 2 RNN layers is generally better for testing performance.

On Table 4.25 we can see the performance of the model when using the Pointer Network as the attention mechanism for the TSP 50 instance with no additional glimpses (0). In this model configuration we see again that the best training and testing performance comes from embedding 2 across both RNN configurations. The training time again increases with additional RNN layers although given the size of the TSP instance it does increase even more considerably than in previous experiments with smaller TSP instances.

The training performance for the 2 embedding improves slightly from 17.30 to 17.04 when moving from 1 to 2 RNN layers, while the performance of other embeddings shows more variability as they tend to have better training results but worse testing

	Pointer Network				
RNN	Embedding	1	2	3	4
1	Train R. (avg)	18.33	**17.30**	22.21	20.32
	Test R.	17.83	**17.02**	20.73	23.43
	Training T.	0:25:08	0:22:42	0:29:23	0:30:45
2	Train R. (avg)	17.38	**17.04**	21.47	17.97
	Test R.	18.99	**18.63**	21.04	19.02
	Training T.	0:36:55	0:38:53	0:40:26	0:37:20

Table 4.25: Training and Testing Results for TSP 50 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with no additional glimpses.

	Pointer Network				
RNN	Embedding	1	2	3	4
1	Train R. (avg)	**17.27**	19.17	23.52	21.33
	Test R.	**18.18**	18.97	24.35	21.63
	Training T.	1:06:13	1:41:42	1:36:35	1:31:09
2	Train R. (avg)	17.48	**16.28**	23.95	18.13
	Test R.	18.25	**18.04**	24.21	18.97
	Training T.	1:17:18	1:45:29	2:04:58	1:34:37

Table 4.26: Training and Testing Results for TSP 50 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with 3 additional glimpses.

results. Given the noticeable increase in the testing value for 2 from 17.02 to 18.63 when moving from 1 to 2 RNN layers, and the increase in training time we have chosen to use 1 RNN layers as the configuration for this instance. This is because we have seen that even if the testing performance shows more variability, particularly with embedding 2, it still consistently achieves the best testing results and has the potential to optimize the final model configuration.

On Table 4.26 we present the results of the model with the use of the Pointer Network as the attention mechanism for the TSP 50 instance with 3 additional glimpses. As we've seen before the training times increase, even if just slightly, with more complex RNN configurations across all embeddings.

This table has some changes in the performance that we've seen in previous instances as embedding 2 only obtains the best testing and training results when the model consists of 2 RNN layers. Instead when the model only has 1 RNN layer embedding 1 is the better choice. We can see that the training performance for embedding 2 improves significantly from 19.17 with RNN configuration 1 to 16.28 with RNN configuration 2, showing the benefit of the second configuration. Even though this pattern doesn't follow for the other embeddings we do see an improvement in the scores for three out of the four embeddings (2, 3, 4) when using 2 RNN layers.

Pointer Network					
RNN	Embedding	1	2	3	4
1.0	Train R. (avg)	36.63	**34.59**	45.81	37.01
	Test R.	31.63	**30.08**	32.78	32.74
	Training T.	1:13:35	1:53:36	1:19:17	1:23:05
2.0	Train R. (avg)	**34.48**	35.19	45.46	45.29
	Test R.	31.77	**30.92**	31.33	33.82
	Training T.	2:43:56	2:41:59	2:57:18	2:35:00

Table 4.27: Training and Testing Results for TSP 100 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with no additional glimpses.

Pointer Network					
RNN	Embedding	1	2	3	4
1.0	Train R. (avg)	36.23	**35.95**	42.86	39.11
	Test R.	31.76	**30.86**	33.04	34.19
	Training T.	4:58:50	4:00:15	4:57:15	3:17:12
2.0	Train R. (avg)	**34.40**	35.00	46.82	36.99
	Test R.	31.82	**30.73**	40.72	33.54
	Training T.	4:09:10	4:36:40	4:00:45	3:17:21

Table 4.28: Training and Testing Results for TSP 100 with Various Embeddings and RNN configuration when using the Pointer Network as the attention mechanism with 3 additional glimpses.

The optimal configuration in terms of both training and testing performance appears to be with the implementation of 2 RNN layers for the TSP 50 instance with the Pointer Network as the attention mechanism using 3 glimpses.

On Table 4.27 we present the results of the model with the use of the Pointer Network as the attention mechanism for the TSP 100 instance with no additional glimpses. We can see that embedding 2 achieves the best performance in testing, with the both RNN layer configuration. For training embedding 1 performs best with 2 layers and 2 for the single RNN layers. On the other side, the other embeddings, 3 and 4, show higher variability in performance.

Training times are longer across when the 2 RNN layers are implemented. The one single RNN layer generally produces better training results compared to the 2 RRN layer configuration. Also, the single RNN layers configuration also shows slightly better or comparable testing performance. Given all of this, the most optimal configuration for the TSP 100 instance with the Pointer Network as the attention mechanism with no additional glimpses is therefore the configuration of a single RNN layer.

On Table 4.28 we present the results of the model with the use of the Pointer Network as the attention mechanism for the TSP 100 instance with 3 additional glimpses. From the table we can see the same pattern repeated as on Table 4.28. As embed-

Pointer Network	TSP 10	TSP 20	TSP 50	TSP 100
0 glimpses	1	2	1	1
3 glimpses	2	2	2	1

Table 4.29: Summary of RNN layers for different TSP instances with 0 and 3 glimpses using Pointer Network.

ding 2 obtains again the best performance during testing on both configurations and embedding 1 obtains the best results for the training for the 2 RNN layer configuration.

From the training times we can note that they are somewhat balanced across the embeddings, with embedding 3 having the shortest training time with a single RNN layer and embedding 4 having the shortest training time for the 2 RNN layer configuration. The single RNN layer configuration generally produces better training results compared to the second configuration, just as we've seen before. The optimal configuration for the TSP 100 instance when using the Pointer Network as the attention mechanism with 3 additional glimpses is the use of the a single RNN layer configuration.

On Table 4.29 we present a summary of the selected RNN layers per experiment when using the Pointer Network with glimpses as the attention mechanism for all TSP instances implemented.

4.2.5 Heads for Self-Attention

The number of heads in the self-attention mechanism can determine the diversity of input representations the model can learn simultaneously. This can improve the model's overall performance by capturing various aspects of the data. As we obtained the results for the comparative analysis to determine the impact of varying the number of attention heads (1, 2 and 4) on the model's performance, our findings showed that the variation in the number of attention heads had no significant effect on the results. Consequently, the performance metrics, including training and testing errors and the training time remained consistent across these configurations.

Given this consistency, we have elected to present the results for the configuration using a single attention head. This choice is made for clarity and simplicity, as the additional heads did not produce any observable differences in performance. The consistency across different head configurations suggests that the benefits of multi-head attention, might be inherently captured by the self-attention mechanism even with a single head in this specific application.

On Table 4.30 we can obsrve the performance of the model when using Self-

RNN	Self Attention				
	Embedding	1	2	3	4
1	Train R. (avg)	**4.40**	4.44	4.47	4.41
	Test R.	4.67	4.64	4.61	**4.59**
	Training T.	0:07:42	0:07:03	0:06:36	0:06:42
2	Train R. (avg)	4.44	**4.42**	4.43	4.45
	Test R.	4.68	4.68	4.72	**4.64**
	Training T.	0:06:47	0:06:04	0:07:37	0:07:13

Table 4.30: Training and Testing Results for TSP 10 with Various Embeddings and RNN configuration when using the Self-Attention as the attention mechanism with a single head.

Attention as the attention mechanism for the TSP 10 instance with one attention head. From the results we can observe that the training times are relatively stable, suggesting that both RNN configurations are scalable and efficient for larger datasets or more complex problems.

Both the training and testing performance remains relatively stable across different RNN configurations for each embedding, with slight variations. Embedding 2 is the one that shows slightly shorter training times, especially in RNN configuration 2 (0:06:04). But overall, training times do not change drastically. This could mean that the complexity of the number of RNN layers has a limited impact on training times for this specific setup.

The results from the table suggest that given the lack of change from the amount of RNN layers that the configuration with 1 RNN layer is generally better for testing performance. Therefore we will use the 1 RNN layer when using Self-Attention as the attention mechanism in TSP 10 instance with only 1 head.

On Table 4.31 we can see the performance of the model when using Self-Attention as the attention mechanism for the TSP 20 instance with one attention head. Just like we saw for instance TSP 10 we can see that the same patterns emerge as the training time, training and testing results are very similar and remain constant across all RNN configurations.

From the results we can see that embedding 1 is the best when training but still the results seem to be very close together in both the testing and training results. The training time also remains constant as the model probably converges easily no matter the RNN configuration. Given all this, we've chosen to select the 1 RNN layer as the metric when implementing the TSP 20 instance with Self-Attention as the attention mechanism with a single head.

On Table 4.32 we can see the performance of the model when using Self-Attention as the attention mechanism for the TSP 50 instance with one attention head. The

RNN	Self Attention				
	Embedding	1	2	3	4
1	Train R. (avg)	**9.47**	9.54	9.48	9.53
	Test R.	8.67	8.72	**8.46**	8.73
	Training T.	0:16:59	0:16:06	0:17:16	0:14:01
2	Train R. (avg)	**9.46**	9.48	9.53	9.47
	Test R.	8.78	8.79	8.77	**8.68**
	Training T.	0:18:39	0:14:28	0:16:27	0:16:34

Table 4.31: Training and Testing Results for TSP 20 with Various Embeddings and RNN configuration when using the Self-Attention as the attention mechanism with a single head

RNN	Self Attention				
	Embedding	1	2	3	4
1	Train R. (avg)	24.91	**24.89**	24.96	24.92
	Test R.	**19.13**	19.33	19.44	19.47
	Training T.	0:38:13	0:52:57	0:54:08	0:44:34
2	Train R. (avg)	24.92	**24.89**	24.93	24.93
	Test R.	**19.42**	19.49	19.44	19.46
	Training T.	0:47:29	0:52:28	0:56:40	0:44:24

Table 4.32: Training and Testing Results for TSP 50 with Various Embeddings and RNN configuration when using the Self-Attention as the attention mechanism with a single head

results, both in training and testing, are very close together between all the embedding and across all RNN configurations.

During the training embedding 1 has obtained the best results and embedding 2 has obtained the best testing results, but still the overall results remain constant across all configurations and without a big different. We do notice that in the training time there is a slight increase as the model goes from 1 RNN layer to 2. Taking all of this into account we've chosen to run the model that uses the Self-Attention as the attention mechanism for TSP 50 instance with a single RNN layer.

Finally, on Table 4.33 we can see the performance of the model when using Self-Attention as the attention mechanism for the TSP 100 instance with one attention head. We see the same patterns as the previous tables emerge. The training performance stays constant across the different RNN layer configuration. The testing performance has a big jump from the training results but the results overall remain constant across all embeddings. We can notice in the table though that the training time does increase as the number of RNN layer increases, perhaps requiring more time to let the model converge or to activate the early stopping mechanism. From this results we can conclude that the best configuration for the model when using Self-Attention as the attention mechanism with 1 head for TSP 100 instance is the use of a single RNN layer.

RNN	Self Attention				
	Embedding	1	2	3	4
1	Train R. (avg)	50.71	**50.58**	50.68	50.76
	Test R.	**45.03**	45.57	45.60	45.17
	Training T.	1:47:30	2:08:54	2:11:24	2:21:17
2	Train R. (avg)	50.72	50.77	**50.65**	50.80
	Test R.	45.77	45.55	**45.42**	45.70
	Training T.	2:05:50	3:18:06	3:04:11	2:45:31

Table 4.33: Training and Testing Results for TSP 100 with Various Embeddings and RNN configuration when using the Self-Attention as the attention mechanism with a single head

Self-Attention	RNN layers			
	TSP 10	TSP 20	TSP 50	TSP 100
1 Head	1	1	1	1

Table 4.34: Summary of RNN layers for different TSP instances with Self-Attention - 1 Head.

On Table 4.34 we summarize the results of the chosen parameters for the model when using Self-Attention as the attention mechanism for all TSP instances and any size head. We have to make note again, that as all results remain constant when implemented with different heads in the Self-Attention we therefore have only presented the results with 1 head to simply for the reader.

4.2.6 Concluding thoughts

In this subsection, we presented the model's performance for each type of embedding with both attention mechanisms: Pointer Network and Self-Attention. We started by evaluating the most optimal batch size as this parameter helps to stabilize the training process by averaging gradients over a batch of samples, leading to improved convergence. Smaller batch sizes can provide robust updates but increasing training time, while larger batches accelerate training at the risk of reduced generalization. We then continued by analyzing the epsilon value, obtaining the most optimal balance between exploration and exploitation in order to improve the model's performance according to the tested attention mechanism.

While the Self-Attention often prefers a balance between exploration and exploitation, the Pointer Network consistently performed better when employing solely an exploitation approach in action selection. After, we examined the performance from the use of either glimpses for the use of the Pointer Network or attention heads for Self-Attention as the attention mechanism. Our findings showed that the Pointer Network without glimpses performed better than with three glimpses, while the performance

Instance	TSP 10	
Attention Mechanism	Pointer Network	Self Attention
Batch size	128	256
Epsilon value	0	0.3
RNN layers - 0 glimpses	1	NA
RNN layers - 3 glimpses	2	NA
RNN layers - 1 Head	NA	1

Table 4.35: Summary of Parameter Configuration Selection for the TSP 10 Instance

Instance	TSP 20	
Attention Mechanism	Pointer Network	Self Attention
Batch size	128	256
Epsilon value	0	0.3
RNN layers - 0 glimpses	2	NA
RNN layers - 3 glimpses	2	NA
RNN layers - 1 Head	NA	1

Table 4.36: Summary of Parameter Configuration Selection for the TSP 20 Instance

of Self-Attention remained consistent regardless of the number of heads used.

As we continue to examine and optimize each feature for each TSP instance, we applied the previously selected features to evaluate the model in its most optimal configuration. By identifying the optimal configuration of these parameters, the model can be fine-tuned to achieve the best possible performance. From this process we aspire to enhance the effectiveness and efficiency of the model individually per TSP instance. We present a summary of the chosen parameters in Table 4.35 for the TSP 10 instance, Table 4.36 for the TSP 20 instance, Table 4.37 for the TSP 50 instance, Table 4.38 for the TSP 100 instance accordingly .

This parameter tuning has helped us to identify configurations that offer the best trade-off between performance and computational resources. Particularly important in practical such as ours as the training time and resource consumption is limited. By recognizing the optimal configuration of these parameters, the model is now fine-tuned to achieve the best possible performance, allowing for a more meaningful comparison

Instance	TSP 50	
Attention Mechanism	Pointer Network	Self Attention
Batch size	64	128
Epsilon value	0	0.3
RNN layers - 0 glimpses	1	NA
RNN layers - 3 glimpses	2	NA
RNN layers - 1 Head	NA	1

Table 4.37: Summary of Parameter Configuration Selection for the TSP 50 Instance

Instance	TSP 100	
Attention Mechanism	Pointer Network	Self Attention
Batch size	128	128
Epsilon value	0	0
RNN layers - 0 glimpses	1	NA
RNN layers - 3 glimpses	1	NA
RNN layers - 1 Head	NA	1

Table 4.38: Summary of Parameter Configuration Selection for the TSP 100 Instance

with the established benchmark models.

4.3 Summary of performance

We've summarized the performance from each attention mechanism and embedding on Table 4.39 for the TSP 10 instance, Table 4.40 for the TSP 20 instance, Table 4.41 for the TSP 50 instance, and Table 4.42 for the TSP 100 instance.

In Table 4.39 we can see that for the TSP 10 instance both best results came from the use of the Pointer Network as the attention mechanism, specifically from the use of embedding 1 with the use of 3 glimpses and from embedding 2 without any glimpses. In Table 4.40 we can see that again the use of the Pointer Network as the attention mechanism obtains the best performance with a score of 6.65 and with the use of embedding 2. On Table 4.41 we see that the pattern continues as the best performance is also from the use of the Pointer Network as the attention mechanism and again with embedding 2. Finally, on Table 4.42 we see that again the use of the Pointer Network as the attention mechanism with 0 glimpses obtains the best result, specifically with the use of embedding 2.

Another important factor to take into account is the training times. Computational resources are limited and cost efficiency is an important factor when selecting an approach to implement in both research and industry scenarios. As efficient resource management is important for us we present a summary of training times, the highlighted times show the best results obtained from each instance, just as in the previous summary tables. Training times are presented in Table 4.43 for the TSP 10 instance, Table 4.44 for the TSP 20 instance, Table 4.45 for the TSP 50 instance, Table 4.46 for the TSP 100 instance accordingly.

From Table 4.43 for TSP 10 instance we can see that there is a big gap between the training time of the experiments that used the Pointer Network as the attention mechanism with and without glimpses, more than double the training time difference.

TSP 10			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	3.90	**3.70**	4.67
2	**3.70**	3.75	4.64
3	4.24	4.65	4.61
4	3.79	4.07	4.59

Table 4.39: Summary of the performance for all embeddings and attention mechanisms on TSP 10 instance.

TSP 20			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	7.02	7.59	8.67
2	**6.65**	6.87	8.72
3	9.97	9.39	8.46
4	7.63	7.32	8.73

Table 4.40: Summary of the performance for all embeddings and attention mechanisms on TSP 20 instance.

TSP 50			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	17.83	18.25	19.13
2	**17.02**	18.04	19.33
3	20.73	24.21	19.44
4	23.73	18.97	19.47

Table 4.41: Summary of the performance for all embeddings and attention mechanisms on TSP 50 instance.

TSP 100			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	31.63	31.76	45.03
2	**30.08**	30.96	45.57
3	32.78	33.04	45.60
4	32.74	34.19	45.17

Table 4.42: Summary of the performance for all embeddings and attention mechanisms on TSP 100 instance.

TSP 10			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	0:05:09	**0:10:57**	0:07:42
2	**0:04:39**	0:09:19	0:07:03
3	0:05:28	0:10:47	0:06:36
4	0:04:50	0:11:01	0:06:42

Table 4.43: Summary of Training Time for All Embeddings in the TSP 10 Instance, highlighted in bold we can see the best performance.

TSP 20			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	0:11:16	0:19:09	0:16:59
2	**0:12:28**	0:25:45	0:16:06
3	0:14:23	0:21:56	0:17:16
4	0:10:48	0:23:30	0:14:01

Table 4.44: Summary of Training Time for All Embeddings in the TSP 20 Instance, highlighted in bold we can see the best performance.

TSP 50			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	0:25:08	1:17:18	0:38:13
2	**0:22:42**	1:45:29	0:52:57
3	0:29:23	2:04:58	0:54:08
4	0:30:45	1:34:37	0:44:34

Table 4.45: Summary of Training Time for All Embeddings in the TSP 50 Instance, highlighted in bold we can see the best performance.

TSP 100			
Embedding	Pointer Network	Pointer Network	Self Attention
	0 glimpses	3 glimpses	1 Head
1	1:13:35	4:58:50	1:47:30
2	**1:53:36**	4:00:15	2:08:54
3	1:19:17	4:57:15	2:11:24
4	1:23:05	3:17:12	2:21:17

Table 4.46: Summary of Training Time for All Embeddings in the TSP 100 Instance, highlighted in bold we can see the best performance.

From Table 4.44 we can see that this difference gets smaller although the general time increase for all other embeddings remains almost again double between the Pointer Network configurations. Table 4.45 and Table 4.46 show the same pattern in training time increase. From all the summary tables we can also see that for TSP instances 10, 20 and 50 the Self Attention experiments have quicker times very close to those of the use of the Pointer Network as the attention mechanism with any glimpses.

On Table 4.43 and 4.45 we can see that the best performances are the shortest in their experiment group, while Tables 4.44 and 4.38 show the opposite. Inference times were not reported because the evaluation of all approaches, including our own and the benchmark methods, demonstrated negligible latency. Taking into account when choosing the training time when choosing the optimal configuration is crucial for efficient resource management. But balancing training times with model accuracy is key to developing practical, high-performing models.

4.4 Results and Analysis

Table 4.47 presents the comparison of the best performance obtained from our approach and the performance obtained from the benchmark models. The average rewards obtained across different TSP problem sizes using our approach showed a significant difference in performance compared to the benchmark models. Notably, IBM's CPLEX and Gurobi optimizers outperform all other baselines across all tested TSP instances. Our approach however, shows a consistent and considerable closeness with the results obtained from the LKH3 model, even beating it in the TSP 100 instance.

There is a pronounced gap between the results from the LKH3 model and the CPLEX and Gurobi model. This can be because LKH3's heuristic methods prioritize computational efficiency and scalability, providing near-optimal solutions quickly. In contrast, CPLEX and Gurobi's exact optimization techniques ensure optimal solutions but require significantly more computational resources and time, especially for larger instances. The disparity between the performance of the model increases as the TSP instance increase accordingly.

From the presented performances we know that all of the best model performances in our approach come from embedding 2, showing that it's the most optimal embedding for solving all of the TSP instances presented. This shows that while embedding 2 is minimalist in design it demonstrates substantial power in capturing complex patterns and relationships within the data. From the training times we know that our approach increases its training time as the TSP instance increases, this can be a potential

	TSP 10	TSP 20	TSP 50	TSP 100
CPLEX	2.85	3.84	5.69	7.77
Gurobi	2.85	3.84	5.69	7.77
LKH3	3.50	6.20	15.20	30.24
Our approach	3.70	6.65	17.02	30.08

Table 4.47: Performance Comparison of Our Approach and Benchmark Models

weakness when scaling our model into greater instance sizes.

From the bigger TSP instances (50, 100) we also saw the trend of a stability in average rewards suggesting that that our embeddings robustly handle increased problem complexity, or possibly, that this problem size represents a threshold beyond which the benefits of different embedding strategies converge. Interestingly, for embeddings 3 and 4 with additional features, such as additional heads, don't produce shorter route lengths despite having greater complexity and sometimes high training times. This shows that increased model complexity does not equal improved performance.

We can conclude that while our approach did not outperform CPLEX and Gurobi, it demonstrated competitive results by coming very close to the performance of LKH3. This proximity to LKH3, a well-regarded heuristic solver for TSP, highlights the potential of our method. The promising results suggest that, with further refinement and optimization, our approach could become a viable alternative.

Chapter 5

Conclusion

In this study, we've explored the application of a DRL algorithm with architecture modifications to solve the TSP, a classic combinatorial optimization problem that has significant implications in fields like logistics and network design. Our approach used the Actor-Critic framework, enhanced by various embeddings and two attention mechanisms and an early stopping mechanism. The experiments were structured to identify the most optimal parameter combination, aiming to optimize the model's performance and training efficiency. The Pointer Network and Self-Attention mechanisms were tested with different embeddings across TSP instances of varying sizes (10, 20, 50, and 100).

In Chapters 1 and 2 we presented some of the latest directions taken with Reinforcement Learning in the context of solving complex combinatorial problems, specifically for the TSP. Current DRL approaches have experimented with different architectures in the hopes of obtaining better results than heuristic methods which can help us understand which features are best to implement to obtain more accurate results in a shorter time and with varying instance sizes.

From the experiments and results from Chapter 4 we explored the implications of different attention mechanisms where the Pointer Network performed better than Self-Attention as the attention mechanism in the model. The experiments highlighted the nuanced trade-offs between different parameter settings. We can say specifically that embedding 2 emerged as the most consistently strong performer, particularly with the Pointer Network.

We presented the impact of the batch size as smaller batch sizes (64) often resulted in shorter training times but showed higher variability in results. While larger batch sizes (128, 256) generally improved stability and performance but increased training times significantly. For the TSP 100 instance with Pointer Network specifically, batch size 128 balanced performance and training time well.

During the exploration of the most optimal epsilon values we observed that lower epsilon values (0) tended to produce better results for the Pointer Network, emphasizing the importance of exploitation in this context. Higher epsilon values (0.3) improved the performance of Self-Attention mechanisms by encouraging exploration. From the experiments with the amount of RNN layers we discovered that the number of RNN layers had a notable impact on performance. For example, 2 RNN layers generally improved results for configurations with 3 glimpses in the Pointer Network. However, single-layer RNNs often sufficed for configurations of the Pointer Network without additional glimpses.

The use of the Self-Attention as the attention mechanism showed stable performance across different RNN configurations, with 1 RNN layer being generally sufficient, yet it never produced better results than those from Pointer Network. Self-Attention mechanisms generally had quicker training times compared to Pointer Network with 3 glimpses, making them more resource-efficient. Finally, in general we saw an increase in training times with higher batch sizes and more complex RNN configurations.

In conclusion, this thesis has explored the application of DRL techniques to solve the TSP using advanced attention mechanisms and a series of embeddings for encoding. Through a comprehensive series of experiments, we identified optimal configurations for batch size, epsilon values, RNN layers, and attention heads. The results demonstrated that the Pointer Network, particularly without additional glimpses, consistently outperformed other configurations, while the Self-Attention mechanism exhibited stable performance across various TSP setups. Despite our approach not surpassing established solvers like CPLEX and Gurobi, it closely matched the performance of LKH3, demostrating its potential. We hope that our findings have contributed valuable insights into the development of efficient and effective deep reinforcement learning models for combinatorial optimization problems, opening new avenues of research.

Bibliography

[1] Charu C Aggarwal. Neural networks and deep learning a textbook. *Neural Networks*, 2023.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[3] Ruibin Bai, Xinan Chen, Zhi-Long Chen, Tianxiang Cui, Shuhui Gong, Wentao He, Xiaoping Jiang, Huan Jin, Jiahuan Jin, Graham Kendall, Jiawei Li, Zheng Lu, Jianfeng Ren, Paul Weng, Ning Xue, and Huayan Zhang. Analytics and machine learning in vehicle routing research. *International Journal of Production Research*, 61(1):4–30, December 2021.

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning, 2017.

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[6] Ann Melissa Campbell, Dieter Vandenbussche, and William Hermann. Routing for relief efforts. *Transportation Science*, 42(2):127–145, May 2008.

[7] Marco Caserta and Stefan Voß. A hybrid algorithm for the dna sequencing problem. *Discrete Applied Mathematics*, 163:87–99, 2014. Matheuristics 2010.

[8] Jianlin Chen and Jianping Luo. Enhancing vehicle routing solutions through attention-based deep reinforcement learning. In *2023 5th International Conference on Data-driven Optimization of Complex Systems (DOCS)*, pages 1–9. IEEE, 2023.

[9] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in neural information processing systems*, 32, 2019.

[10] Hanni Cheng, Haosi Zheng, Ya Cong, Weihao Jiang, and Shiliang Pu. Select and optimize: Learning to aolve large-scale tsp instances. In *International Conference on Artificial Intelligence and Statistics*, pages 1219–1231. PMLR, 2023.

[11] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In Jian Su, Kevin Duh, and Xavier Carreras, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 551–561, Austin, Texas, November 2016. Association for Computational Linguistics.

[12] Meriem Cherabli, Megdouda Ourbih-Tari, and Meriem Boubalou. Refined descriptive sampling simulated annealing algorithm for solving the traveling salesman problem. *Monte Carlo methods and applications*, 28(2):175–188, 2022.

[13] William Cook, Stephan Held, and Keld Helsgaun. Constrained local search for last-mile routing. *Transportation Science*, 58(1):12–26, January 2024.

[14] Paulo R. de O. da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In *Proceedings of The 12th Asian Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 465–480. PMLR, 2020.

[15] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*, pages 170–181. Springer, 2018.

[16] Peter Dieter, Matthew Caron, and Guido Schryen. Integrating driver behavior into last-mile delivery routing: Combining machine learning and optimization in a hybrid decision support framework. *European Journal of Operational Research*, 311(1):283–300, November 2023.

[17] Raafat Elshaer and Hadeer Awad. A taxonomic review of metaheuristic algorithms for solving the vehicle routing problem and its variants. *Computers and Industrial Engineering*, 140:106242, 02 2020.

[18] Han Fang, Zhihao Song, Paul Weng, and Yutong Ban. Invit: A generalizable routing problem solver with invariant nested view transformer. *arXiv preprint arXiv:2402.02317*, 2024.

[19] Getu Fellek, Ahmed Farid, Shigeru Fujimura, Osamu Yoshie, and Goytom Gebreyesus. G-dganet: Gated deep graph attention network with reinforcement learning for solving traveling salesman problem. *Neurocomputing*, 579:127392, 2024.

[20] Getu Fellek, Ahmed Farid, Goytom Gebreyesus, Shigeru Fujimura, and Osamu Yoshie. Deep graph representation learning to solve vehicle routing problem. In *2023 International Conference on Machine Learning and Cybernetics (ICMLC)*, pages 172–180. IEEE, 2023.

[21] Getu Fellek, Ahmed Farid, Goytom Gebreyesus, Shigeru Fujimura, and Osamu Yoshie. Graph transformer with reinforcement learning for vehicle routing problem. *IEEJ Transactions on Electrical and Electronic Engineering*, 18(5):701–713, 2023.

[22] Keld Helsgaun Filip Kilibarda. elkai. https://github.com/fikisipi/elkai, 2023. Accessed: 2023-06-01.

[23] Lei Gao, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu. Learn to design the heuristics for vehicle routing problem. *arXiv preprint arXiv:2002.08539*, 2020.

[24] Yong Liang Goh, Wee Sun Lee, Xavier Bresson, Thomas Laurent, and Nicholas Lim. Combining reinforcement learning and optimal transport for the traveling salesman problem. *arXiv preprint arXiv:2203.00903*, 2022.

[25] Nathan Grinsztajn, Daniel Furelos-Blanco, Shikha Surana, Clément Bonnet, and Tom Barrett. Winner takes it all: Training performant rl populations for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36, 2024.

[26] Gurobi Optimization, LLC. Gurobi optimizer. [Software]. Available from https://www.gurobi.com, 2023. Accessed: 2023-06-01.

[27] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[28] Tapan Kumar Hazra and Ankan Hore. A comparative study of travelling salesman problem and solution using different algorithm design techniques. In *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 1–7. IEEE, 2016.

[29] André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. *arXiv preprint arXiv:2106.05126*, 2021.

[30] IBM Corporation. Ibm ilog cplex optimization studio. [Software]. Available from https://www.ibm.com/products/ilog-cplex-optimization-studio, 2023. Accessed: 2023-06-24.

[31] Xinyu Ke, Rui Ding, and Shuangyuan Yang. Reinforcement learning for routing problems with hybrid edge-embedded networks. In *International Conference on Intelligent Computing*, pages 641–652. Springer, 2023.

[32] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems!, 2018.

[33] Gilbert Laporte. A concise guide to the traveling salesman problem. *Journal of the Operational Research Society*, 61(1):35–40, 01 2010.

[34] Haojian Liang, Shaohua Wang, Huilai Li, Liang Zhou, Xueyan Zhang, and Shaowen Wang. Bignn: Bipartite graph neural network with attention mechanism for solving multiple traveling salesman problems in urban logistics. *International Journal of Applied Earth Observation and Geoinformation*, 129:103863, 2024.

[35] Yang Liu, Fanyou Wu, Zhiyuan Liu, Kai Wang, Feiyue Wang, and Xiaobo Qu. Can language models be used for real-world urban-delivery route optimization? *The Innovation*, 4(6):100520, November 2023.

[36] Yang Liu, Fanyou Wu, Zhiyuan Liu, Kai Wang, Feiyue Wang, and Xiaobo Qu. Can language models be used for real-world urban-delivery route optimization?, November 2023.

[37] Fu Luo, Xi Lin, Zhenkun Wang, Tong Xialiang, Mingxuan Yuan, and Qingfu Zhang. Self-improved learning for scalable neural combinatorial optimization. *arXiv preprint arXiv:2403.19561*, 2024.

[38] Minh-Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation, 2015.

[39] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning, 2019.

[40] Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. *Advances in Neural Information Processing Systems*, 36, 2024.

[41] Daniel Merchán, Jatin Arora, Julian Pachon, Karthik Konduri, Matthias Winkenbach, Steven Parks, and Joseph Noszek. 2021 amazon last mile routing research challenge: Data set. *Transportation Science*, 58(1):8–11, 2024.

[42] Yimeng Min, Yiwei Bai, and Carla P Gomes. Unsupervised learning for solving the travelling salesman problem. *Advances in Neural Information Processing Systems*, 36, 2024.

[43] Baichuan Mo, Qingyi Wang, Xiaotong Guo, Matthias Winkenbach, and Jinhua Zhao. Predicting drivers' route trajectories in last-mile delivery using a pair-wise attention-based pointer neural network. *Transportation Research Part E: Logistics and Transportation Review*, 175:103168, July 2023.

[44] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takáč. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

[45] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62, 2021.

[46] Wenbin Ouyang, Yisen Wang, Paul Weng, and Shaochen Han. Generalization in deep rl for tsp problems via equivariance and local search. *SN Computer Science*, 5(4):1–20, 2024.

[47] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[49] Pedro Zattoni Scroccaro, Piet van Beek, Peyman Mohajerin Esfahani, and Bilge Atasoy. Inverse optimization for routing problems, 2023.

[50] Sami Serkan Özarık, Paulo da Costa, and Alexandre M. Florio. Machine learning for data-driven last-mile delivery optimization. *SSRN Electronic Journal*, 2022.

[51] Shichao Sun, Zhengyu Duan, and Qi Xu. School bus routing problem in the stochastic and time-dependent transportation network. *PLOS ONE*, 13(8):e0202618, August 2018.

[52] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[53] Jan Van Leeuwen. *Handbook of theoretical computer science (vol. A) algorithms and complexity.* Mit Press, 1991.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[55] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

[56] 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.

[57] Thi Quynh Trang Vo, Mourad Baiou, Viet Hung Nguyen, and Paul Weng. Improving subtour elimination constraint generation in branch-and-cut algorithms for the tsp with machine learning. In *International Conference on Learning and Intelligent Optimization*, pages 537–551. Springer, 2023.

[58] Zhenwei Wang, Ruibin Bai, Fazlullah Khan, Ender Ozcan, and Tiehua Zhang. Gase: Graph attention sampling with edges fusion for solving vehicle routing problems. *arXiv preprint arXiv:2405.12475*, 2024.

[59] Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems. *IEEE transactions on neural networks and learning systems*, 33(9):5057–5069, 2021.

[60] Yunqiu Xu, Ling Chen, Meng Fang, Yang Wang, and Chengqi Zhang. Deep reinforcement learning with transformers for text adventure games. In *2020 IEEE Conference on Games (CoG)*, pages 65–72. IEEE, 2020.

[61] Yunqiu Xu, Meng Fang, Ling Chen, Gangyan Xu, Yali Du, and Chengqi Zhang. Reinforcement learning with multiple relational attention for solving vehicle routing problems. *IEEE Transactions on Cybernetics*, 52(10):11107–11120, October 2022.

[62] Xue Yang and Jie-sheng Wang. Application of improved ant colony optimization algorithm on traveling salesman problem. In *2016 Chinese Control and Decision Conference (CCDC)*, pages 2156–2160. IEEE, 2016.

[63] Yunchao Zhang, Kewen Liao, Zhibin Liao, and Longkun Guo. Enhancing policy gradient for traveling salesman problem with data augmented behavior cloning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 327–338. Springer, 2024.

[64] Jie Zhao, Biwei Xie, and Xingquan Li. Weight uncertainty in transformer network for the traveling salesman problem. In *2023 International Symposium of Electronics Design Automation (ISEDA)*, pages 219–224. IEEE, 2023.

[65] Jiuxia Zhao, Minjia Mao, Xi Zhao, and Jianhua Zou. A hybrid of deep reinforcement learning and local search for the vehicle routing problems. *IEEE Transactions on Intelligent Transportation Systems*, 22(11):7208–7218, November 2021.

[66] Shijie Zhao and Shenshen Gu. A deep reinforcement learning algorithm framework for solving multi-objective traveling salesman problem based on feature transformation. *Neural Networks*, page 106359, 2024.

Acronyms

A-C

Actor-Critic

AI

Artificial Intelligence

AM

Attention Mechanism

CNN

Convolutional Neural Network

DL

Deep Learning

DRL

Deep Reinforcement Learning

LSTM

Long Short-Term Memory

ML

Machine Learning

NLP

Natural Language Processing

NN

Neural Network

PN

Pointer Networks

RL	Reinforcement Learning
RNN	Recurrent Neural Network
TSP	Travelling Salesman Problem
VRP	Vehicle Routing Problem