Query Execution Plan Search Space Enumeration by Reinforcement Learning

by

Chang Liu

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the
MCS degree in
Computer Science and Concentration Applied Artificial Intelligence

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Join order selection, a critical subfield of query optimization, focuses on determining the optimal join sequence for SQL queries to minimize execution cost. This problem becomes increasingly challenging as the number of tables in a query expands, leading to an exponentially growing search space. Exhaustive enumeration of all possible join orders is computationally infeasible. Recent advances in join order selection have leveraged deep reinforcement learning (DRL) to address this challenge, offering dynamic and adaptive solutions that improve on traditional rule-based or cost-based approaches. This thesis proposes a DRL framework specifically designed for join order selection. Our approach reformulates join order selection as a sequential decision-making problem, where an agent learns to select joins by maximizing query performance. By integrating graph neural networks (GNN) to model relationships between tables and Tree-LSTM networks to capture the hierarchical nature of join sequences, we create informative representations. Additionally, we replace the vanilla DQN with a dueling-DQN to enhance training stability and convergence time. The DRL-based framework adaptively explores the search space and learns optimal join sequences through iterative interactions. Our experimental results demonstrate that our method consistently outperforms other baselines, including both traditional methods and other DRL-based techniques. This highlights the potential of reinforcement learning to significantly enhance query optimization, offering a more adaptive and efficient solution for join order selection.

Acknowledgements

I am extremely grateful to my supervisor, Prof. Verena Kantere for invaluable advices, continuous support and patience during my master study. Her expertise knowledge in database has provided insightful feedback and constructive advice to help me shaping this work, and her encouragement kept me motivated during the challenging times and inspired me. I am also indebted to Amin Kamali, who is a PhD candidate in uOttawa, for his strong background in deep learning and always help me solve the problems I have encountered. Special thanks to my family for their love, understanding and patience.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Query Optimization and Query Plan . . . 1

1.1.1 Concepts . . . 2

1.2 The Join Order Selection Task . . . 2

1.3 Motivation and Problem Statement . . . 3

1.4 Methodology . . . 4

1.5 Contribution . . . 4

1.6 Thesis Organization . . . 6

2 Background - Query Optimization 7

2.1 Database Management System . . . 7

2.1.1 Query Processing . . . 7

2.2 Query Optimization . . . 8

2.2.1 Cardinality Estimation . . . 8

2.2.2 Cost Model . . . 9

2.2.3 Join Order Enumeration . . . 10

2.3 Modern Query Optimization Techniques . . . 11

2.3.1 Cardinality Estimation using ML . . . 11

2.3.2 Learning-based Cost Model . . . 14

2.3.3 Join Order Enumeration using DRL . . . 19

2.3.4 A Comparative Analysis of the State-of-the-art in ML-based Query Optimization . . . 24

3 Background - Deep Reinforcement Learning **29**
3.1 Deep Learning and Neural Network . . . 29
3.2 Layers in Neural Networks . . . 31
3.2.1 Linear Layer . . . 31
3.2.2 Convolutional Layer . . . 32
3.2.3 Pooling Layer . . . 33
3.3 Activation Function in Neural Networks . . . 34
3.3.1 Sigmoid . . . 35
3.3.2 Tanh . . . 35
3.3.3 ReLU . . . 36
3.4 Types of Neural Networks . . . 37
3.4.1 Graph Neural Network . . . 37
3.4.2 Recurrent Neural Network . . . 39
3.5 Reinforcement Learning . . . 46
3.5.1 Markov Decision Process . . . 46
3.5.2 Q-learning . . . 50
3.6 Deep Reinforcement Learning . . . 52
3.6.1 DQN . . . 52

4 Introduction of GTDD **54**
4.1 Representation Learning . . . 55
4.1.1 Column Representation Learning . . . 56
4.1.2 Table Representation Learning . . . 57
4.1.3 State Representation Learning . . . 61
4.2 Deep Reinforcement Learning in Join Order Selection . . . 64
4.2.1 DQN and Dueling-DQN . . . 64
4.2.2 Reward . . . 65
4.2.3 Curriculum Learning . . . 66
4.2.4 Action Mask . . . 67
4.3 Implementation of GTDD . . . 68

5 Experimental Study **70**
5.1 Experiment Setup . . . 70
5.1.1 Dataset . . . 70
5.1.2 Baselines . . . 71
5.1.3 Metrics . . . 71
5.1.4 Data Partitioning . . . 72
5.1.5 Training Sample Collection . . . 72
5.1.6 Training Time . . . 73
5.2 Evaluation . . . 73
5.2.1 Cost Training . . . 73
5.2.2 Latency Tuning . . . 76

6 Conclusion and Future work **79**
6.1 Conclusion . . . 79
6.2 Future Work . . . 80

APPENDICES **82**

A Bao's hint sets **83**

References **85**

List of Tables

2.1 Comparison of cardinality estimation models 25
2.2 Comparison of cost models 26
2.3 Comparison of join order selection models 27

5.1 MRC to DP 75
5.2 Learning efficiency in seconds 75
5.3 GMRL to DP 78

List of Figures

1.1 Two join orders for a query with different cost 3

2.1 One-hot encoding of T_q, J_q and P_q [1] 12
2.2 Framework of multi-set convolutional network [1] 13
2.3 Neo system model [31] 15
2.4 Neo query-level encoding [31] 15
2.5 Neo plan-level encoding [31] 16
2.6 Neo value network architecture [31] 16
2.7 Balsa architecture [54] 17
2.8 Balsa safe exploration [54] 17
2.9 Bao system model [30] 18
2.10 Bao vectorized query plan tree [30] 19
2.11 One episode of possible join order [22] 20
2.12 Rejoin join order selection [32] 20
2.13 A query and its corresponding featurization [21] 21
2.14 Selection and physical operators 21
2.15 Different joins trees with the same feature vector[53] 22
2.16 Representations in RTOS [53] 23
2.17 Representations in Jogger [7] 24

3.1 Deep neural network [15] 30
3.2 Linear layer 31
3.3 Image to matrix 32
3.4 Example of computation of a convolution calculation [25] 33
3.5 Example of max pooling layer [43] 34
3.6 Example of average pooling layer [43] 34

3.7 Sigmoid function . . . 35
3.8 Tanh function . . . 36
3.9 ReLU function . . . 37
3.10 The left graph is undirected, the right graph is directed . . . 38
3.11 traditional RNN [16] . . . 40
3.12 LSTM network [49] . . . 41
3.13 LSTM network - forget gate [49] . . . 42
3.14 LSTM network - input gate [49] . . . 43
3.15 LSTM network - output gate [49] . . . 44
3.16 A pictorial representation of reinforcement learning model . . . 47
3.17 Grid world example [55] . . . 48
3.18 Value iteration example . . . 49
3.19 Left figure is the arbitrary initialled value function, right figure is converged value function . . . 50
3.20 How does a model learn by trial and error [2] . . . 51

4.1 Framework of GTDD . . . 56
4.2 Column representation and table representation . . . 58
4.3 BFS vs DFS [10] . . . 58
4.4 Schema graph construction . . . 60
4.5 State representation . . . 61
4.6 DQN vs Dueling-DQN . . . 65
4.7 Feedback as reward vs ratio as reward . . . 66
4.8 Invalid action and valid action . . . 68

5.1 Training curve (MRC) on JOB . . . 74
5.2 Per-query result using MRC, evaluated on the test set . . . 74
5.3 Training curve (GMRL) on JOB . . . 76
5.4 GMRL on different templates of JOB . . . 77
5.5 GMRL on different templates of JOB . . . 77

Chapter 1

Introduction

In the digital era, data plays an important role in our daily lives. Data refers to the organized collection of information that can be accessed, managed, and updated. The invention of databases makes these functionalities easy to manipulate. Databases typically store data in a structured format using tables, which allows for efficient querying and retrieval of specific information. Data stored in tables that consist of rows and columns. Each column represents a specific field, such as name, date, or address, while each row contains a record that holds data for these attributes. This structure enables databases to handle large volumes of data and perform complex operations efficiently through query languages, such as structured query language (SQL). The request for information or action using SQL is known as a query. Queries are essential for interacting with databases and manipulating the data stored within them.

1.1 Query Optimization and Query Plan

A query is defined as a request that interacts with the database management system (DBMS), and when a query is sent to the DBMS, a series of processes are initiated before the query is executed. These processes typically require a certain amount of time to complete. Nevertheless, as the quantity of data in a database increases, the time required to execute queries also increases. Consequently, reducing the time to execute queries becomes crucial to improving the user experience and efficiency. Typically, the DBMS transforms the query into a set of query plans. These plans are usually represented as a tree structure. Different query plans for the same SQL return the same output, but the time and resource (e.g. CPU, memory, I/O) required to execute the query vary widely. Therefore, selecting the optimal plan can reduce response time, minimize resource consumption, and effectively handle larger data sets, greatly improving the user experience.

1.1.1 Concepts

The DBMS transforms the query into a set of query plans, and several factors contribute to the variability of each plan. In this section, we introduce these factors and essential fundamental concepts in the field of query optimization. This will facilitate a more comprehensive understanding of the question.

- **Join Operator:** The join operator is responsible for determining the methodology of combining rows from two tables based on a related column between them. This is a crucial aspect of retrieving data from multiple tables in a database. Different join operators have distinct characteristics in data retrieval, such that the merge join is a join algorithm that requires both input tables to be sorted on the join keys, whereas the hash join is a join algorithm that uses hash tables to find matching rows.
- **Scan Operator:** The scan operator is responsible for accessing and retrieving data from a table during query execution. There are different types of scans, each with its characteristics, such that the table scan involves scanning the entire table to retrieve the required data, whereas the index scan involves scanning the index structure to locate and retrieve the rows that match the search criteria.
- **Join Order:** The join order is the sequence in which tables are joined together in a database query. We explain this concept in greater detail in the next section.
- **Table Cardinality:** Table cardinality refers to the number of distinct rows in a database table. It represents the uniqueness of data within a table and it is a crucial factor in query optimization.
- **Column Selectivity:** Column selectivity refers to the uniqueness of values in a column. A highly selective column has many unique values, while a low-selectivity column has few unique values.
- **Join Cardinality:** Join cardinality refers to the number of rows in the result set of a join operator between two tables. It represents the size of the output dataset generated by the join operation.
- **Schema:** A schema is a structured framework that defines how data is organized and managed within the database. It outlines the logical configuration of the database, including the definition of tables, columns, data types, relationships, indexes, and so on.

1.2 The Join Order Selection Task

The objective of this research is to investigate the process of join order selection, or join order enumeration, which is a technique employed in query optimization for relational databases. It is a crucial aspect of query optimization in relational DBMSs, when a query

involves multiple tables, determining the order in which tables are joined significantly impacts the overall performance of the query execution plan. The goal is to find the join order that minimizes the cost of processing the query. For instance, shown in Figure 1.1, a query contains tables A, B, C and D. There are multiple join orders that could be applied, such as ((A JOIN B) JOIN C) JOIN D, ((A JOIN C) JOIN B) JOIN D, and so on. Each join operation has an associated cost, which depends on factors such as the number of rows being joined, and in a join tree, the cost accumulates from the leaf nodes to the root, and the previous context is important for future decisions. Even if the results were identical, the cost would differ. The problem is computationally intensive due to

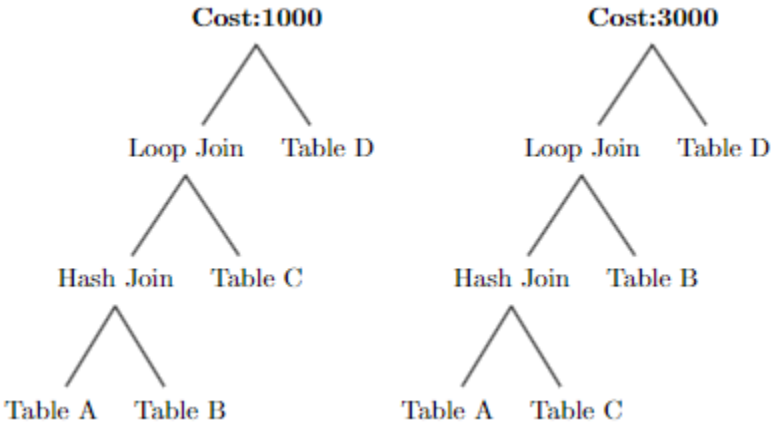


Figure 1.1: Two join orders for a query with different cost

the factorial growth of possible join orders as the number of tables increases, resulting in higher resource consumption that is not feasible. The join order selection aims to determine the most efficient sequence in which to execute join operations, minimizing resource consumption (e.g. CPU, memory, I/O) and reducing query response time using different approaches. Further details are provided in the subsequent section.

1.3 Motivation and Problem Statement

Efficient query processing is crucial to the performance of database systems, particularly in large datasets that are often subjected to complex queries. The selection of join order plays an important role in determining the execution time of queries and is a critical aspect of query optimization. Optimal join order should reduce the intermediate table cardinality to save resource consumption ultimately reduce response times, and improve overall efficiency. In real life, low response times on queries could positively impact users' experience. Although traditional plan enumerators are widely used and generate satisfactory plans through the use of heuristics, such as dynamic programming (DP) [9], there are two significant limitations. First, the search space for potential query execution plans increases exponentially with the number of tables included in the query, rendering exhaustive evaluation of all possible join orders computationally infeasible. Second, traditional plan

enumerators rely on heuristics or predefined patterns to generate query plans. Since these heuristics or predefined patterns are static, the enumerators lack the capacity to adapt to changes in the heuristics or pattern. Consequently, they continue to select the same join order and fail to utilize the response time for improvement. Many researchers leverage deep reinforcement learning (DRL) to address the limitations of traditional plan enumerators. The model utilizes the feedback to search for the optimal join orders and avoid suboptimal ones. The learned plan enumerator could incorporate the feedback as a guide to enable the DRL agent to learn through trial and error, thereby improving the quality of the generated join orders for queries. Our review of the most advanced methodologies reveals that RTOS [53] represents an optimal starting point. However, it is evident that DRL-based approaches continue to present certain limitations:

- Firstly, the process of training reinforcement learning is typically unstable due to the random selection of training samples from the training set, which lacks the preprocessing of the training data.
- Secondly, RTOS employs a simplistic one-hot encoding approach for query representation, which only captures the link information but ignores the table information.
- Thirdly, the majority of the work only captures the query information while failing to consider the schema information used to construct the database.

1.4 Methodology

Our study comprises several phases, each designed to contribute to a comprehensive understanding of the problem and the development of solutions. The study begins with a thorough review of existing literature on query optimization algorithms, techniques, and related methodologies. This review serves to establish a foundational understanding of these state-of-the-art approaches, identify gaps in current research and inform the design of our methodology. Next, we perform an analysis of the join order selection problem in detail and the gathering of insights through the research process. This analysis helps us to delineate the scope of the problem and identify relevant constraints and objectives. Based on the insights gained from the literature review and problem analysis, we devise novel algorithms and heuristics for the problem of representation learning and DRL. We implement our proposed algorithms within PostgreSQL, thereby enabling us to evaluate the performance of the model. Our work includes two phases, the cost training phase and the latency tuning phase, which we explain in Chapter 5. By adopting this methodology, we aim to contribute novel insights and practical solutions to the field of join order selection and to enhance the performance of the DBMS.

1.5 Contribution

In order to address the challenges present in traditional join order enumerators, numerous researchers have proposed the use of DRL as a potential solution. However, despite the

advantages of DRL-based approaches, there are still limitations that must be addressed. We proposed GTDD, a framework that is based on Graph neural networks (GNNs), Tree-structured long-short term memory (Tree-LSTM), and Dueling-DQN. GTDD aims to address these limitations and provide a more comprehensive solution:

- The first limitation is the unstable training process, which arises from the random selection of training samples from the training set, potentially leading to slow convergence. To address this issue, we implement curriculum learning, a technique that progressively presents training data in increasing order of difficulty. Initially, the model is exposed to easier examples, enabling it to learn basic patterns and build strong foundational knowledge. As training progresses, the difficulty of the examples gradually increases, allowing the model to tackle more complex data once it has acquired the necessary prior knowledge. This approach ensures smoother training dynamics and faster convergence by guiding the model through a structured learning path.

- The second limitation is that most existing methods represent queries using simplistic one-hot encoding, which primarily captures the structural connection between tables but fails to capture the rich, contextual information within the tables themselves. To address this, we leverage one-hot encoding as an adjacency matrix to construct a join graph, where nodes represent tables and edges represent the relationship between them. This graph is then fed into graph neural networks (GNNs), which are capable of learning complex, relational patterns from graph-structured data. Additionally, we incorporate table embeddings, which encode the semantic features of the tables into the model. This allows the GNN to generate more informative and context-aware representations of the query, enhancing the model's ability to make accurate predictions and optimize the join order.

- The third limitation is that most existing methods focus on extracting information from incoming queries but overlook valuable information contained in the schema file, which defines the relationships between tables. This schema information plays a crucial role in understanding the structure and dependencies of the database. To address this, we incorporate Node2Vec [10], a technique that learns low-dimensional embeddings of nodes in a graph. By applying Node2Vec to the schema's table relationship graph, we capture the structural and semantic relationships between tables, enriching the model's understanding of how the schema influences query execution. This additional information enhances the ability to optimize join orders by taking into account not only the query itself but also the underlying database schema.

- We have observed that previous work in this area generally relies on a standard deep Q-network (DQN) architecture, without exploring alternative reinforcement learning (RL) architectures that could potentially improve performance. To address this gap, we introduce the dueling-DQN architecture, which separates the q-value computation into two streams, value stream and advantage stream. This modification allows the model to more effectively evaluate the quality of actions by learning separate representations of the value of a state and the relative advantage of each action. By

using dueling-DQN, we aim to improve the stability and convergence of the learning process, especially in environments with large or complex state spaces, like join order selection.

We further explain these limitations and solutions in detail in Chapter 5.

1.6 Thesis Organization

Our studies encompassed a diverse range of research areas, and we split the background into two chapters. In Chapter 2, we introduce the background of query optimization and related research. We also analyze the state-of-the-art research on query optimization using ML and DL. In Chapter 3, we introduce the background of DL and DRL. In Chapter 4, we provide a detailed explanation of the proposed DRL model, GTDD. In Chapter 5, we describe the experimental setup and implementation details of the proposed DRL model, as well as the evaluation results of the model. Finally, in Chapter 6, we present a conclusion and future directions.

Chapter 2

Background - Query Optimization

2.1 Database Management System

A database management system (DBMS) is software that is designed to manage, store and retrieve data in a structured and efficient manner. It provides the connection between the users/application and the database. The most common database is relational database which organizes data into tables with rows and columns and establishes relationships between them. Popular relational databases are DB2 [14], PostgreSQL [9] and MYSQL [38].

2.1.1 Query Processing

When a query is submitted by the end user, the query goes through a series of processing steps before the results are returned. The process of the DBMS, which involves the interpretation and execution of queries, is referred to as query processing. This process can be divided into three parts: parsing and translation, optimization and evaluation.

1. **Parsing and translation:** The first step is parsing and translation, where the DBMS checks the syntax and structure of the query to ensure it satisfies the rules of the database and ensures that the tables and columns referenced in the query actually exist. It then translates the query into a query tree or a similar internal form.

2. **Optimization:** Query optimization is the most crucial aspect of query processing, wherein the parsed query is examined in order to determine the most efficient method of execution. A query can be translated into multiple expressions with the same meaning. The ultimate goal of query optimization is to determine the most efficient plan for executing the given query, thereby improving overall system performance and resource utilization. By leveraging various techniques and strategies, the query optimizer transforms a potentially slow query into a fast and efficient one. Three key factors have been identified as significantly impacting the optimizer, and have been studied for decades, **cardinality estimation**, **cost model** and **join order selection**. We explain these three factors in detail in the following sections.

3. **Evaluation:** Following the optimization process, the optimizer determines an optimal execution plan among other candidate plans that specify the actual steps to be taken to execute the query. These steps include such as scan operators, join orders, and index lookups. The system then follows the steps outlined in the physical execution plan, accessing tables, performing joins, and applying any necessary filtering or sorting operations.

2.2 Query Optimization

Query optimization refers to a series of techniques and methods to optimize the execution of a query to improve query performance and efficiency. The DBMS must decide how to execute incoming queries from different aspects; the index usage, join order selection, usage of join operators, etc. The main purpose is to find an optimal execution plan for a given query in a given database. The need for timely and efficient data retrieval becomes more critical as the volume and complexity of data continue to grow. There are three major fields that has been studied in query optimization: **Cardinality Estimation**, **Cost Model**, and **Join order selection**

2.2.1 Cardinality Estimation

Cardinality estimation involves the prediction of the number of rows or tuples that will be returned by a specific operation in the query execution plan. It provides information about the size of the intermediate result that guides the model to make decisions on join order, index selection, and join operator selection. Cardinality Estimation plays a fundamental role in query optimization, as inaccurate estimation errors can propagate further to the cost model and mislead the cost model to decrease the overall performance. As Lohman states about cardinality estimation: "*The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities* [27]". During decades of learning, numerous methods have been developed to improve the accuracy of cardinality estimation. The majority of traditional DBMSs employ two distinct methodologies: histogram and sampling [11].

Histogram-based estimation

A significant proportion of DBMSs employ the histogram-based estimation method. Two DBMSs that are worthy of particular mention are PostgreSQL [9] and SQL Server [42]. Histogram is a statistical result to estimate the number of distinct values in a column without having to scan the entire dataset. Each column has corresponding histogram represent the distinct value, when a query involving cardinality estimation is issued, the system looks at the histogram to make informed decisions on cardinality estimation by analyzing the distribution. This approach usually performs well since it produces based on statistical data, but when the distribution is skewed, the accuracy may deviate significantly from the actual cardinality.

Sampling-based estimation

In contrast, many other DBMSs employ a sampling-based approach, MySQL [38] and MariaDB [33] are two well-known DBMSs that utilize this methodology. The idea is to use a randomly selected subset of data from a table or column to approximate the characteristics of a dataset without processing the entire dataset. This is usually adopted in large datasets which provides a cost-effective way to gain insights into database characteristics. However, since the samples are selected randomly, the correctness of the technique depends on the representativeness of the sample, the cardinality estimation would be inaccurate once the samples are not representative.

Machine learning in cardinality estimation

As previously stated, both of the classical cardinality estimation methods yield unstable and inaccurate results. An increasing number of researchers are turning to alternative methods, employing machine learning techniques in cardinality estimation. Malik et al. [28] propose a method that treats the DBMS as a black box, instead of directly inspecting the data distribution, the estimator sends a set of queries to the database and observes the responses to infer cardinality information. Kipf et al. [1] propose a deep learning method based on sampling to extract informative representations from the query. Liu et al. [26] propose a risk-aware model that embeds uncertainty into cardinality estimation. We review the state-of-the-art techniques that employ ML for cardinality estimation in the following section.

2.2.2 Cost Model

Cost model aims to predict the resource usage associated with executing a particular query plan. It predicts the overall consumption of resources, such as CPU, disk (number of block transfers), memory usage (amount of RAM used), and number of tuples read/write during query execution. These factors are important when estimating the cost of executing the given query. Traditional optimizers employ a cost model to predict the cost, such as assigning a weight to each factor and identifying the optimal combination of factors that minimizes the cost. This enables the selection of plans that minimize the overall resource usage and the response time of executing selected plans. In most DBMSs, cardinality estimation and cost model are closely related, with the cardinality estimator providing information on intermediate results and guiding the cost model. Consequently, more accurate cardinality estimation leads to more accurate cost estimation. Nevertheless, the estimated cost is not always accurate. Many researchers employ modern techniques, such as machine learning, which is a technique that enables machines to learn in a manner similar to humans. This technique is often used to solve complex tasks, we explain machine learning in detail in the following section. Train a cost model using machine learning become a popular research line. Neo [31] is the first end-to-end query optimizer that replaces the cardinality estimation and cost model into their trained value model. Bao [30] takes

advantage of hint sets to generate query plans. Balsa [54] learns in a simple, simulated environment to learn prior knowledge before train with actual datasets.

2.2.3 Join Order Enumeration

Join order enumeration is a process within database query optimization in a relational DBMS where the optimizer explores and evaluates different orders of table joins based on cost. The goal is to find the most efficient join order that minimizes the overall cost of processing the query which is a critical step in generating an optimal query execution plan and can have a significant impact on the performance of the query. The enumeration of all possible join orders would be costly because the possible join orders grow exponentially with the number of tables involved. For a query with n tables involved, there would be (n!) possible join orders. PostgreSQL employs a dynamic programming approach, which is a search method that selects the two tables with the lowest join cost at each step and saves the solution to subproblems for using it later.

Dynamic Programming

Dynamic programming (DP) [41] is a problem-solving technique used to efficiently solve problems that can be decomposed into overlapping subproblems. DP solves and stores the solution to subproblems in a table, allowing the solution to a large problem to be constructed by combining the solutions to its subproblems. It is particularly useful in optimization problems where the goal is to find the best solution among a set of feasible solutions. PostgreSQL [9] adapts DP to avoid redundant computations. The ultimate goal is to find the join order for the input query, and this problem could be decomposed into problems for each possible combination of tables. Ideally, given an optimal join tree $\mathcal{T}$ with n relations $\{R_1,R_2,...,R_n\}$, any subtree t of $\mathcal{T}$ would be an optimal join tree [3]. Nevertheless, DP is a greedy algorithm that seeks to minimize the join cost at each step. However, it should be noted that local optimality does not guarantee global optimality. In order to address this issue, an increasing number of researchers leverage reinforcement learning (RL) to resolve the join order problem, as the join order selection can be considered as a Markov decision process, with the next step being contingent on the current step. RL is a technique that encourages the agent to explore and learn the action (in this case, the order of joining) by interacting with the environment. A significant number of research studies have demonstrated remarkable achievements, ReJOIN [32] is the first work that implements deep reinforcement learning (DRL) to solve join order problem, it encodes the join tree with height using row vectors, and using the estimated cost as feedback to the generated plan. ReJOIN [32] treats the join order as a join graph, and capture the selectivity information and physical operators, and uses both cost and latency as a feedback. RTOS [53] adapts tree-LSTM to capture the sequence of the join order and allows for adaptation when the schema changes. JOGGER [7] adapts attention tree and graph neural networks to capture an informative representation of the state. We present a thorough review of the above DRL approaches of join order selection in the following section.

2.3 Modern Query Optimization Techniques

The traditional query optimization techniques rely on static and predefined rules, which usually based on statistics and assumption about data distribution and query patterns. These predefined rules are tuned with expert knowledge and often lack adaptability to real-time changes. As a result, traditional DBMSs may not effectively handle data and schema changing, leading to degraded performance. Moreover, traditional query optimizers may struggle to scale efficiently with large and complex datasets, as the search space grows exponentially with the number of tables, joins, and selection conditions involved. This can result in long optimization times and limited scalability. Thus, query optimization still remains one of the hardest problems in DBMSs. In recent years, an increasing number of studies have demonstrated the potential of ML as a promising approach to address the challenges of query optimization, particularly in the domains of cardinality estimation, cost model, and join order selection. The advent of ML has opened up a new avenue of research in query optimization. Unlike traditional optimizers, ML models can handle more complex queries and, crucially, the optimizers learn from mistakes and avoid executing suboptimal plans. The field of query optimization has been studied and investigated for decades, with much of the research demonstrating significant improvements. Malik et al. [28] propose a method that treats the DBMS as a black box, and categorizes different query into patterns to infer the estimated cardinality. Kipf et al. [1] use DL to capture query features to predict cardinality. Rather than investigate cardinality estimation, the cost model has also attracted researchers' attention. Bao [30] leverages the idea of hint sets, Neo [31] provides the first end-to-end optimizer that handles the entire process of query optimation, which including join order, index selection, and operator selection. Balsa [54] proves that the model could learn without experts' demonstration. Many researchers have also looked at how to choose the join order with the least cost. DQ [21] takes the height of the join tree as a feature, ReJOIN [32] encodes the operators into consideration, RTOS [53] takes advantage of tree-LSTM to handle the sequence of join order and capture an informative representation and JOGGER [7] embeds attention mechanism to capture state representation. The researchers employ RL to select the join order. It appears that RL is a well-known solution for join order selection. Further details regarding the aforementioned works provide in the following sections.

2.3.1 Cardinality Estimation using ML

As explained above, traditional cost models are usually based on histogram or sampling which can be inaccurate and may result in an the execution plan that is suboptimal. Histogram-based methods are often relatively simple designs that partition the data space into bins and count the number of elements falling into each bin. This can provide fast and efficient estimates of data distribution and selectivity. However, if the data distribution is skewed or contains outliers, the bins may not accurately represent the underlying data distribution. The sampling method is typically employed in data spaces with large-scale datasets. Sampling provides fast estimates of cardinality with minimal computational overhead. However, sampling bias and sample size selection can result in inaccurate estimates.

The accuracy of the estimates is highly related to the size and quality of the sampled data. If the sample size is large, it could cause computational overhead and a smaller size may lead to inaccuracy. Additionally, the sampled data usually cannot represent the whole underlying dataset. Furthermore, biased samples can also lead to inaccurate cardinality estimates. Despite decades of research on cardinality estimation, traditional cardinality estimation methods still produce inaccurate results. Recent studies have employed ML methods in cardinality estimation to enhance the accuracy of cardinality estimation. Malik et al. [28] propose a novel method for estimating query cardinality without relying on detailed knowledge of the underlying data distribution or complex statistical models. Instead, it introduces a black-box approach that learns from executing a set of queries and observes the feedback that DBMS generats to infer cardinality. Their approach analyzes the queries with similar syntactic information into patterns and deploys ML techniques to extract useful information from the observed data. This approach constructs the feature representation of a query from attributes, operators, constants, and aggregates. It employs several machine-learning techniques, including classification and regression, model trees and locally-weighted regression. The objective of this study is to assess the effectiveness of ML in improving accuracy of cardinality estimation. The result indicate that the model not only learns from the feedback but also improves the accuracy and limits space overhead. However, the performance of ML models often depends on the availability and quality of historical query execution data used to train the estimation model. If the training data is sparse or not representative enough, the accuracy of the estimates may be compromised.

Rather than employing a pure ML approach, an alternative method combines DL, utilizing a multi-set convolutional network to capture the information of the query [1]. This approach is built based on sampling-based estimation, which is used to capture the query features and accuracy cardinalities. The authors primarily concentrate on representation learning, which aims at extracting critical information from queries, including tables, predicates, and joins. As illustrated in Figure 2.1, the query is represented as a collection (T_q, J_q, P_q) where T_q represents a set of tables participating in the query, J_q represents a set of joins in the query, and P_q represents a set of predicates in the query. The sets are encoded using one-hot encoding, with each set passing separately into a two-layer neural network. Subsequently, the outputs are then concatenated and fed into the final output network, which generates the predicted cardinality as shown in Figure 2.2.

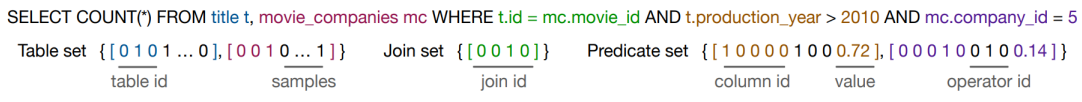


Figure 2.1: One-hot encoding of T_q, J_q and P_q [1]

The results demonstrate an improvement in cardinality estimation. However, the model was trained using queries with 0 to 2 joins and tested on queries with 1 to 4 joins. Additionally, the queries lack any predicates on strings. The simplicity of the training and testing datasets may not provide sufficient evidence of the model's performance when increasing

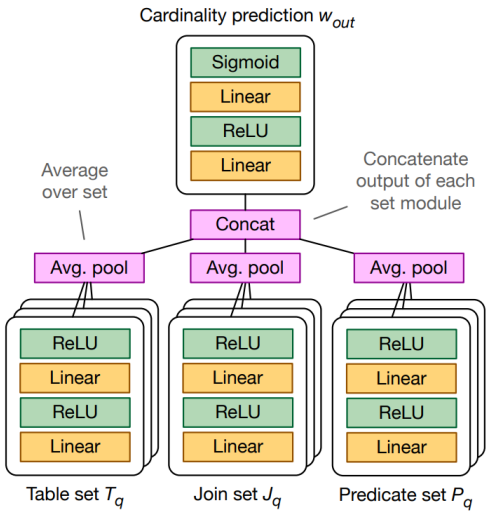


Figure 2.2: Framework of multi-set convolutional network [1]

the number of joins.

To address the need for more complex queries with multiple predicates involved, Fauce[26] has been proposed and it is the first work to incorporate uncertainty or confidence level in cardinality estimation. It introduces a novel feature extraction method called dependency graphs, which not only captures the information provided in the query, but also captures the real correlation across all the table columns. For a database $\mathcal{D}$, which contains n tables, $\mathcal{D} = \{T_1, T_2, ..., T_n\}$, and each table contains m columns, $T_i = \{C_{i1}, C_{i2}, ..., C_{im}\}$. The total number of columns is denoted as $Total_C = \sum_{i=1}^{n} \sum_{j=1}^{m} C_{ij}$. Each query is represented as a set, which contains four component: $\langle Tables \rangle$,$\langle Joins \rangle$,$\langle Columns \rangle$,$\langle Values \rangle$, where $\langle Tables \rangle$ is a set contains the target tables in query q, $\langle Joins \rangle$ contains join relation, $\langle Columns \rangle$ contains columns in the query q, and $\langle Values \rangle$ contains predicates values in query q. Besides, Act(q) denotes the number of row that satisfy all the predicates in query q and $Card$(q) denotes the estimated cardinality for query q. To estimate the cardinality, Fauce developed a regression model $\mathcal{M}$ since the cardinality of a query is definitely a real number. The model $\mathcal{M}$ aims to generate an estimated cardinality $Card$(q) that is as close as possible to the true cardinality Act(q). It first requires to transfer the four sets which are $\langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, and $\langle Values \rangle$ into four vectors repectively, and the combination of the vectors would be a representation for a query q. Rather than using one-hot encoding, Fauce treats the database as a undirected graph where tables are nodes and edges connect to tables that participate in a join, and finally get feature of set $\langle Tables \rangle$ as f_T with length of $m\lceil log(m+1) \rceil$. To be able capture the join features f_J from $\langle Joins \rangle$, it gets information from the neighbours and uses the current connected graph to predict the connected graph in its context, and then uses the result of the hidden layer as embedding. For columns features f_C from $\langle Columns \rangle$, Fauce uses randomized dependence coefficient (RDC) values for each pair of column to capture dependency on columns. If the RDC value exceed a constant threshold τ, it means the two columns are dependent to each other, and vice versa, and then it uses the dependency as edges to connect columns

to build a graph and use similar embedding method. To be able capture the last feature f_V from $\langle Values \rangle$, Fauce represents that for each column c_i in table T, the column can be represented as $lb_{ci} \leq c_i \leq ub_{ci}$, where lb_{ci} is the lower-bound in predicates in the query, ub_{ci} is for upper-bound, and c_i is the i-th column in table. The combination of these four features is the encoding of the incoming query that use to feed into the regression model $\mathcal{M}$. Fauce quantifies the uncertainty into data uncertainty and model uncertainty, with its variance $Var(y) = Var(E[y|x]) + E[Var(y|x)]$. The model uncertainty quantified as $U_m(y|x) = Var(E[y|x])$ and the data uncertainty is $U_d(y|x) = E[Var(y|x)]$, both of the uncertainty explains the variance. Fauce uses ϕ_m to represent the threshold of model uncertainty $U_m(q)$ and ϕ_d for data uncertainty $U_d(q)$. If the uncertainty exceed the threshold, it means the estimation is not confident, and vice versa. Depending on the corresponding threshold, and there are four situation:

- If both data and model are confident, the model accepts the predicted cardinality.
- If the data is not confident, but the model is confident, the model also accepts the redicted cardinality.
- If the data is confident, but the model is not confident, the model saves the query into a buffer and return the true cardinality as a label, while undergoing do incremental learning.
- If the data is not confident, and the model is not consident, the model saves the query into a buffer, and enlarge the number of queries in buffer by sampling additional training data for incremental learning.

Fauce is the first work that quantify uncertainty and incorporate it into the analysis. In this regard, it significant reduces the estimation errors for complex queries and provides accurate estimated cardinality.

2.3.2 Learning-based Cost Model

The employment of DL and ML in cardinality has not eliminated the possibility of inaccurate results, which can lead to suboptimal query execution plans. As a result, an increasing number of researchers have opted to combine DL and ML methods in the design of another component, known as the cost model.
Most of the studies focus on replacing one component of the optimizer with learned models. In contrast, Neo [31] implements a brand new idea that replaces the cardinality estimator and cost model with their own value network. The value network takes a partial query plan and predicts the best-expected value by completing the query plan. Neo is the first end-to-end learning approach that includes join order, index, and physical operator selection. As illustrated in Figure 2.3, the model comprises two phases: the expertise collection phase and the runtime phase. In the expertise collection phase, Neo leverages the existing query optimizer as a starting point to generate a set of plan/latency pairs as expertise experience. The second phase involves query processing, which learns from both new arriving queries

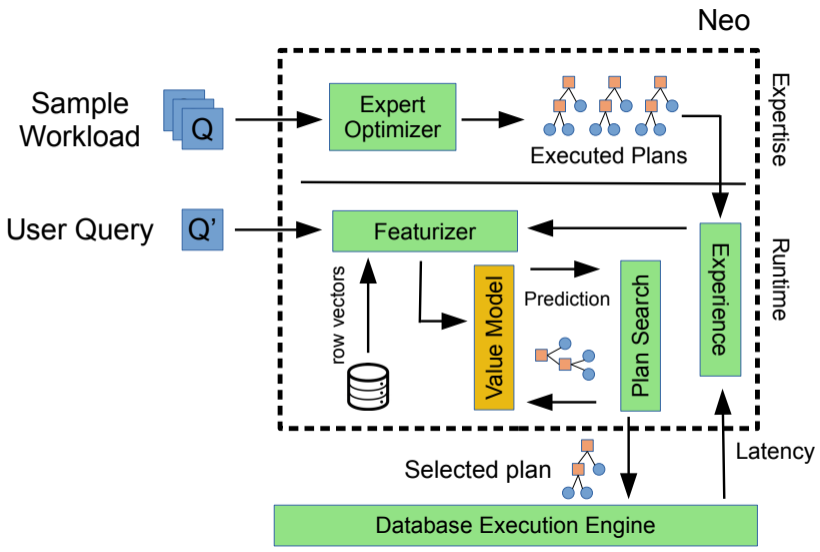


Figure 2.3: Neo system model [31]

and the expert experience.

Another important component of the Neo system model is the featurizer. Neo extracts both query-level and plan-level encodings. As Figure 2.4 shows, query-level encoding are represented by an adjacency matrix, which specifies the tables participate in the join of a given query. The value $m_{i,j} = 1$ refers that table i and j are joined together. To reduce the representation, it is sufficient to consider only the upper triangle, as it is a symmetric matrix. Figure 2.5 declares plan-level encoding. Each node contains a vector of size $2 + 2n$ where n is the number of tables in the database. The first two digits represent the join type (e.g. merge join or loop join), and each table can be scanned using either an index scan or a table scan. Rather than employing a simple convolutional neural network

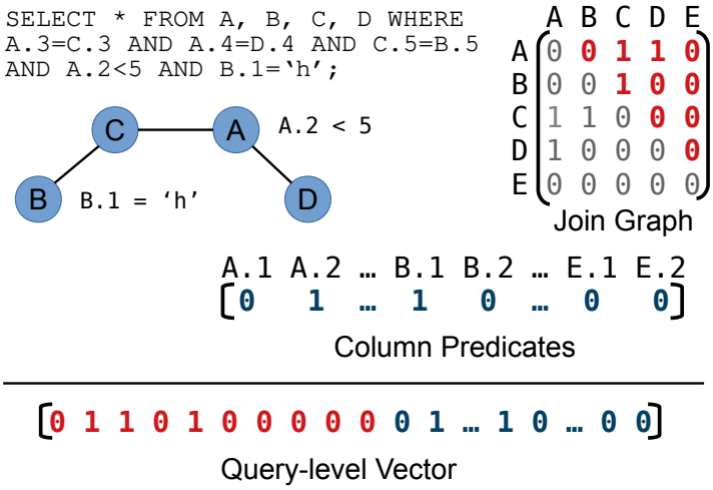


Figure 2.4: Neo query-level encoding [31]

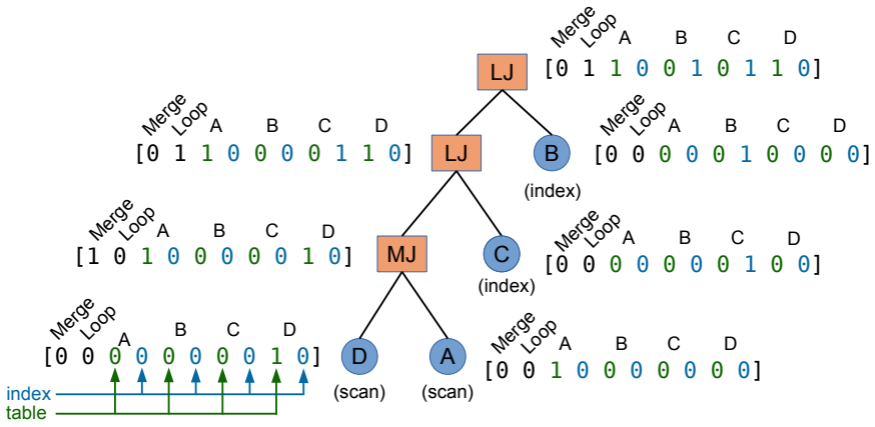


Figure 2.5: Neo plan-level encoding [31]

and fully connected layer, Neo has devised a novel approach, the use of a tree-structured convolutional neural network (TCNN) [37], as illustrated in Figure 2.6. This innovative technique aligns perfectly with the underlying structure of the execution plan. TCNN could capture information from parent-child relations, thereby enabling the network could learn from the relationship between the tables. In contrast to Neo, Balsa [54] introduces a new

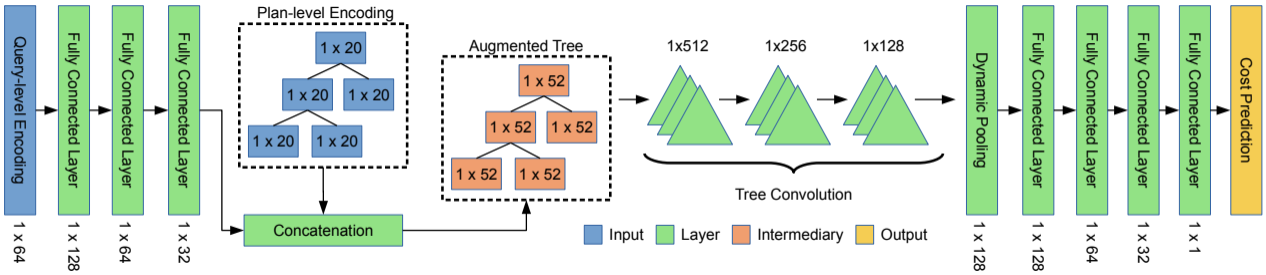


Figure 2.6: Neo value network architecture [31]

learning model that is capable of outperforming the DBMS without relying on expertise or experience. The unique challenge in such an approach is that most of the execution plans are expensive, which drastically downgrades the training time at the beginning since the model has no prior knowledge. To avoid disastrous plans, Balsa has deployed a simulator to generate cost feedback using a basic cost model with a cardinality estimator. Once it has acquired prior knowledge, it proceeds to the second phase, during which it learns by executing the query. However, even with prior knowledge, it is not guaranteed to avoid disastrous execution plans; therefore they employ a timeout mechanism to limit the search time. Balsa employs the same encoding at the plan and query levels as Neo, but Balsa takes different search strategies to accelerate and ensure the robustness of the model, safe execution and safe exploration as figure 2.7 shows. Safe execution is a mechanism that is used to avoid long-running plans with unacceptably high latency. Balsa has implemented a dynamic timeout to avoid such issues. Following the completion of the simulation learn-

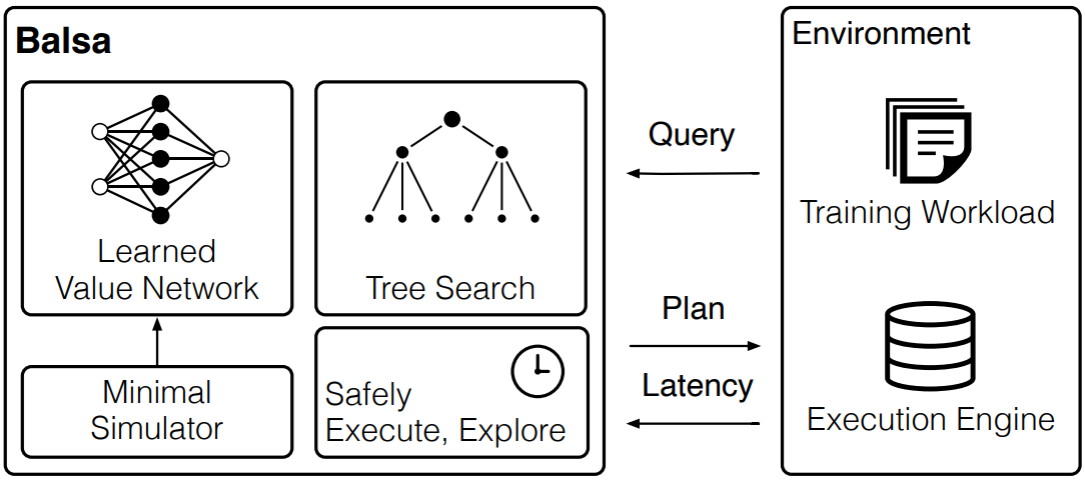


Figure 2.7: Balsa architecture [54]

ing phase, the model initiates the execution of queries with objective of measuring the latency. Given the model's prior knowledge during simulation learning, the model should able to identify the most disastrous execution plan. Consequently, the model executes all queries and observes the maximum runtime T across all queries. After the first iteration, the timeout has been set to $S \times T$ where S is a constant factor that need to be tuned, the parameter provides additional flexibility to accommodate variability in latency. If any plan exceeds the timeout, it should be terminated immediately, as it is slower than the current known plan. The timeout would be set to $S \times T'$ once the new maximum $T' < T$ has been observed. Instead of seeking for a single optimal query execution plan, Balsa aims

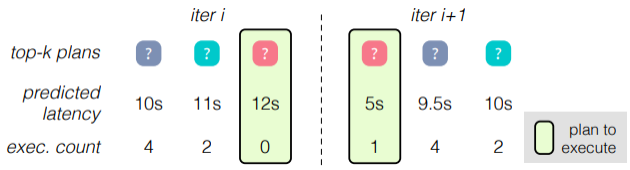


Figure 2.8: Balsa safe exploration [54]

to identify the top k plans through best-first beam search [34]. Beam search is a heuristic search algorithm that aims to identify approximate solutions when an exhaustive search is not feasible. The core idea is to explore multiple paths simultaneously and maintain a fixed number of candidates referred as 'beam size'. At each iteration, the algorithm expands the current set of candidates by considering all possible next steps. In the case of selecting plans, the candidate is a sub-plan, and the next step is to add a table into the sub-plan. The algorithm terminates when k query execution plan are have been found. In addition to the beam search, Balsa also considers the number of execution plans, it prioritizes the unseen plan to execute like Figure 2.8 shows.
While both Balsa and Neo have demonstrated remarkable performance, they suffer from a

significant drawback: they are unable to adapt to the changes in the schema due to their fixed-size feature representation, either on plan-level and query-level encoding. Whenever there is a modification in the database schema, retraining the model becomes necessary, which increases the overhead of training. Besides that, the model usually encounters a 'cold-start' problem since the model require a huge amount of data to train before it could have a positive impact on the performance. Bao [30] has introduced a novel approach to address such issues, opening up a new gate for tackling this challenge. Bao works as an extension on top of the optimizer and takes advantage of decades research on query optimization that directly captures global information from the underlying optimizer (e.g. PostgreSQL). As Figure 2.9 shows, when a user submits a query, Bao first uses predefined

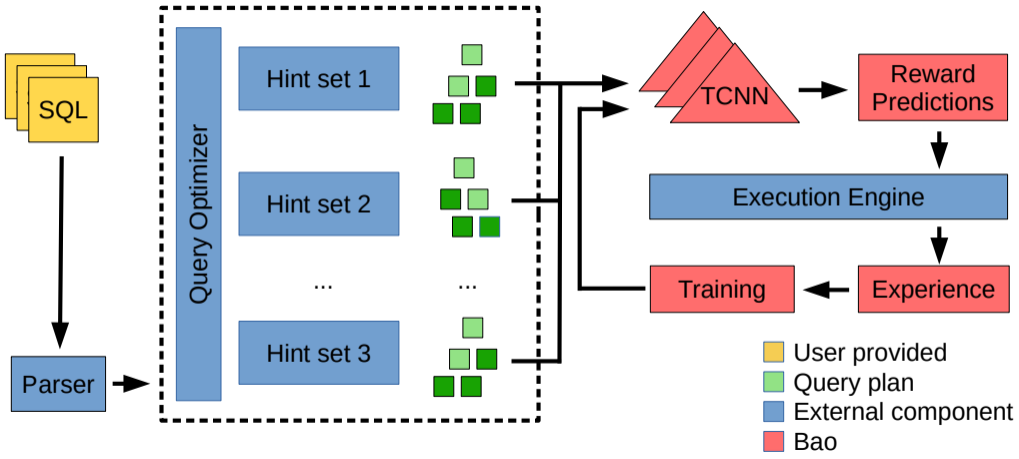


Figure 2.9: Bao system model [30]

hint sets to generate a plan for each hint set. In query optimization, a hint represents a switch that enable or disable operators within the underlying optimizer, such as enabling or disabling an index scan or a merge join. A hint set is a combination of these hints that declares which operators are enabled and which are disabled. Bao employs 48 distinct hint sets for each incoming query[1]. Each plan will transfer into query plan trees like Figure 2.10 shows, each node including information about the scan type, join type, cardinality and cost. The plan tree is then passed through a multiple-layer TCNN, which predicts the latency of each query plan. The results demonstrate that Bao can outperform traditional query optimizers for both open-source (e.g. PostgreSQL) and commercial (e.g. Oracle) databases within an hour of training. Moreover, as Bao takes advantage of the underlying query optimizer, Bao has cardinality and cost provided by the optimizer, which allows the model encode the query plan instead of encoding from the query-level and plan-level. This approach provides flexibility to enable Bao to adapt to changes in schema and data without the need for retraining the model, which greatly reduces the overhead. Although Bao demonstrates remarkable achievement, it still has limitations. The hint is global, which means it cannot be modified during the query optimization process. This characteristic

[1]see appendix

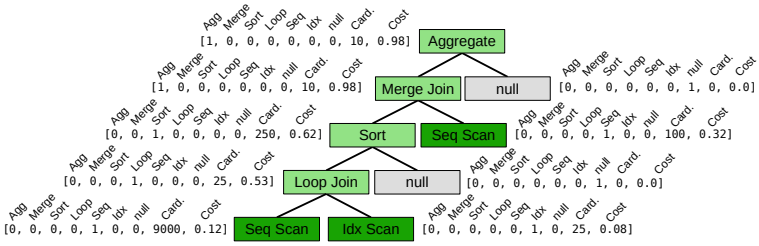


Figure 2.10: Bao vectorized query plan tree [30]

makes Bao likely to discover sub-optimal query plans, but relatively good plans overall. Another limitation is that Bao still requires the experts' domain knowledge to select the hint sets.

2.3.3 Join Order Enumeration using DRL

All join order enumeration works have adopted RL to select the join order, as the problem can be constructed to a Markov decision process and perfectly fits the Markov property. The Markov property means that the next state and reward only depend on the given current state. To convert the join order enumeration problem into RL, there are many denotations that require clarification:

- **Environment:** The environment is the database in which the agent interacts.
- **Agent:** The model that predicts action based on the current state.
- **State:** The state would be a combination of two parts 1) the tables that have already been joined together, and 2) the tables that are ready to be joined.
- **Action:** The action space is defined as the set of potential actions corresponding to different pairs of tables that are joined together.
- **Reward:** The reward is usually not obtained until the plan is complete. This could be based on the feedback (e.g. cost) provided by the optimizer.
- **Episode:** One episode means the model completes the join order permutation for a given query and gets feedback.

Join Order Enumeration Example

Figure 2.11 describes a complete episode from initial step s_0 to the terminate state s_3. At s_0, the episode begins with all tables awaiting joins. We can extract the possible action from the join predicates in the query, and the action space would be $\{(A\ join\ B), (B\ join\ C), (C\ join\ D)\}$. The agent then selects the action a_0 which joins table A and table B together,

the environment gets the action and transit to a new state s_1. This process iterates until the terminate state s_3 where all the tables are joined together. In other words, a query plan tree is generated. Once the plan tree is ready, it passes through the underlying optimizer to obtain the reward which may be based on latency or estimated cost. This reward is then used to guide the learning process of the RL framework. ReJOIN [32]

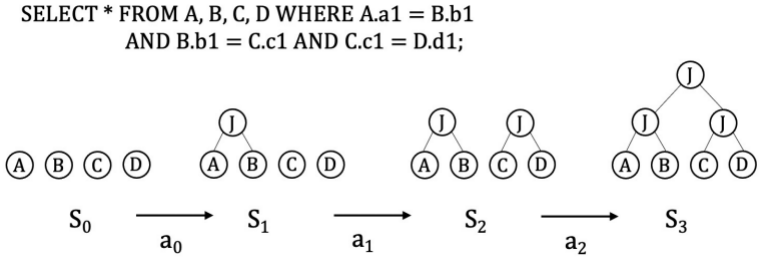


Figure 2.11: One episode of possible join order [22]

highlighted that current query optimizers utilize static join enumeration algorithms, which lack feedback on plan quality. Consequently, such optimizers may persistently select the same suboptimal plans due to the missing mechanism making the optimizer "learn from their mistakes". To make the model learn from their mistakes, they propose a model called *ReJOIN* which takes advantage of DRL. ReJOIN uses a vector representation to capture the state information as a tree structure. Furthermore, it also captures the join predicate and column predicate as Figure 2.12 shows. It first constructs a matrix with

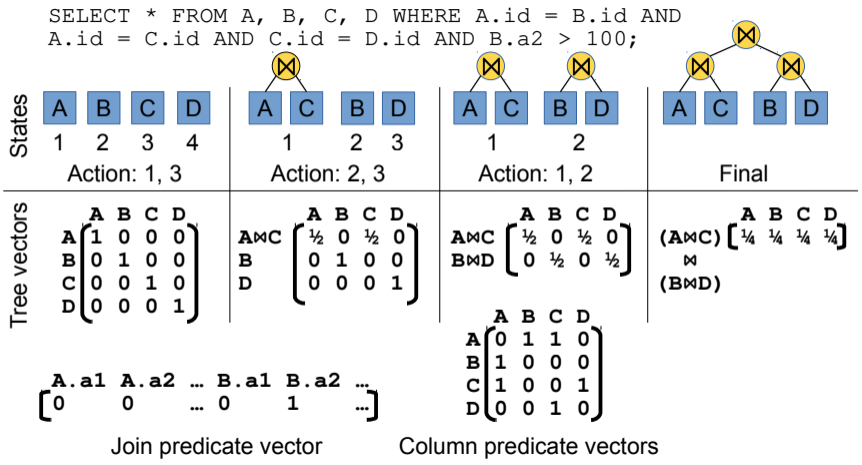


Figure 2.12: Rejoin join order selection [32]

size $N \times N$ where N is the number of tables in the database. The value would set to 0 if the table doesn't participate in the query and set to $\frac{1}{h(i,x)}$ otherwise, where h is the height of relation r_i in the subtree x. Join predicates and column predicates are the critical information that needs to be captured. Similarly to the tree representation, ReJOIN has

another matrix with $N \times N$ to represent the join predicates. for each value $v_{i,j}$, if table i and table j does not participate in join conditions, $v_{i,j}$ sets to 0 and set 1 otherwise. For instance, $A.id = B.id$ means Table A and B joins together and $v_{A,B}$ and $v_{B,A}$ would set to 1. Besides join predicates, Rejoin uses a vector of size k as a column predicate where k is number of columns across the entire database. The column which participate in the local predicates will set to 1 and set to 0 otherwise. For instance, $B.a2 > 100$ is the local predicate for table B, column 2, and the value would be set to 1.
DQ [21] has further improves the encoding of a query, in order to capture more information from the query. DQ incorporates selectivity and a physical join operator. Nonetheless, DQ ignores the height of the tree and considers the join plan solely as a graph, the join of two tables represents an edge connecting two corresponding nodes. As Figure 2.13 shows,

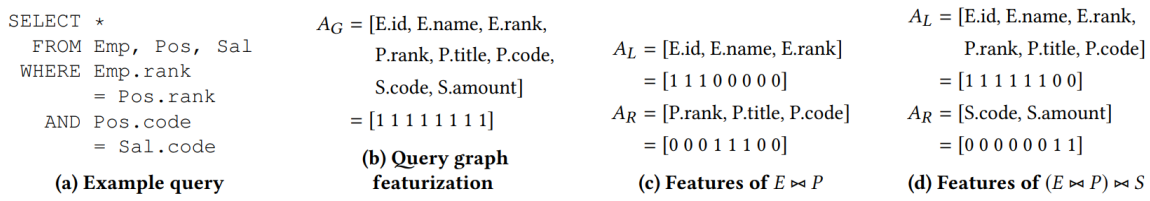
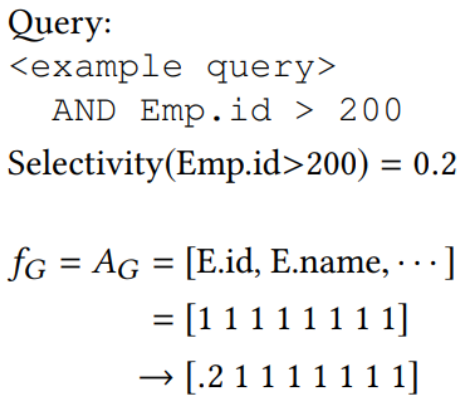
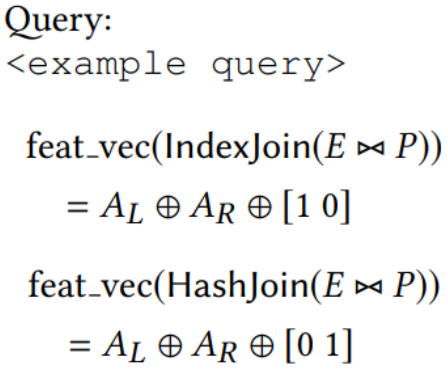


Figure 2.13: A query and its corresponding featurization [21]

the query graph featurization denotes the tables and their corresponding columns and simply sums up the table representations to obtain the intermediate representation of a join. Subsequently, if a column participates in a local predicate, it replaces the number to its selectivity. The join operators are then concatenated with the table representations in order to obtain the final state representation as Figure 2.14 shows. This encoding provides more detailed information on the selectivity on the columns and the type of join operator comparing to ReJOIN [32].



(a) Selectivity scaling in query graph feature[21]



(b) Concatenation of physical operators in join features[21]

Figure 2.14: Selection and physical operators

Even though DQ and ReJOIN demonstrate an improvement in the join order selection

problem, they also exhibit a significant drawback in their feature representations. As illustrated in Figure 2.15, different join orders may result in the same feature representation.

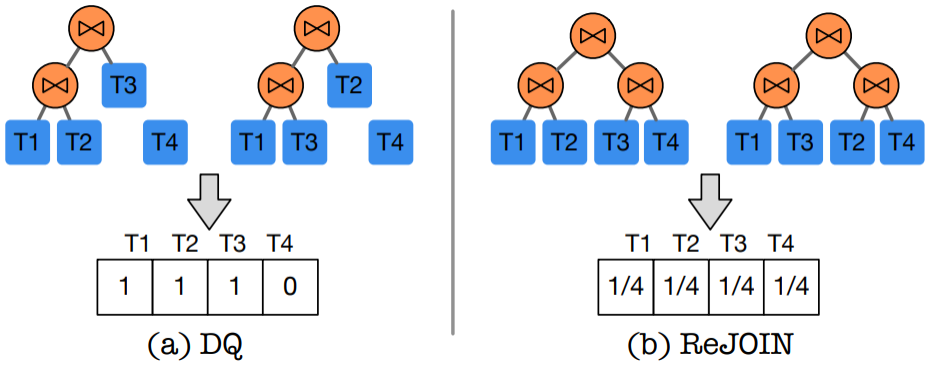


Figure 2.15: Different joins trees with the same feature vector[53]

The join order of DQ in the left tree is $(T1 \bowtie T2) \bowtie T3$, while for the right tree, it is $(T1 \bowtie T3) \bowtie T2$. Similar to ReJOIN, the left tree's join order is $(T1 \bowtie T2) \bowtie (T3 \bowtie T4)$, and for the right tree, it is $(T1 \bowtie T3) \bowtie (T2 \bowtie T4)$. Although the join orders vary, their representations are the same. This similarity misleads the model into treating different join orders as identical, causing ambiguity and leading to sub-optimal results. RTOS [53] has proposed tree-LSTM to effectively address this challenge, ensuring that different join orders yield different representations. Instead of using single vector representation and fully connected layers, RTOS takes the advantage on tree-LSTM which reads a tree structure as input and outputs a representation of the tree, which perfectly fits the structure of the join plan. Moreover, RTOS employs fine-grained representations for columns, tables and queries.

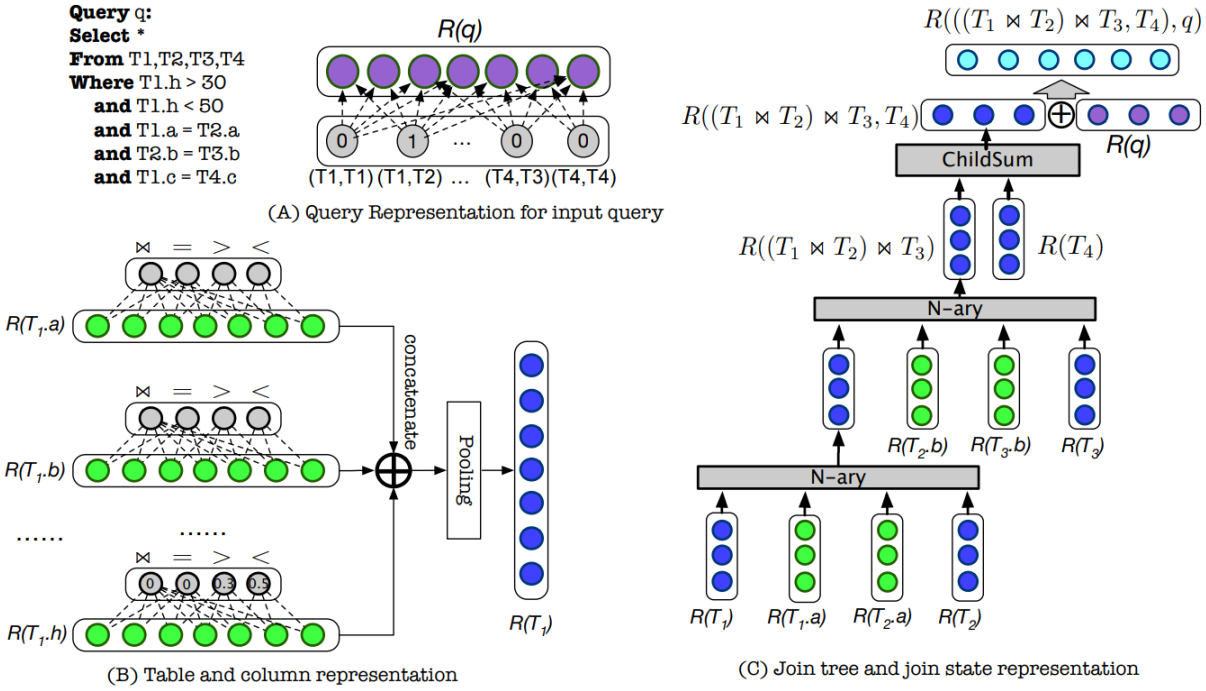


Figure 2.16: Representations in RTOS [53]

As Figure 2.16 shows, the column representation is a vector of size 1×4. the first digit represent whether the column participate in a join, the second,third and fourth digits represent if the column participates in predicates with operator $=,>,<$ respectively. The table representation is the concatenation of all columns representation in the tables and is fed into a pooling layer. The query representation is a matrix m with size $N \times N$ where N is the number of total tables across the database. $m_{i,j}$ is 1 means there is a join between table i and table j. It is worth noting that all the representations will be fed into the tree-LSTM to obtain the final representation of the state that will be used in the RL environment. There are two different types of tree-LSTM, **N-ary tree-LSTM** and **childsum tree-LSTM**. N-ary tree-LSTM accepts inputs from both the left and right table representations, as well as the column representation when two tables are joined together. In contrast, childsum tree-LSTM uses all join trees in the forest as input when generating the state representation. Furthermore, RTOS employs dynamic pooling to enable the model adapts changes in schema (i.e. Adding a new column, adding a new table) without retraining. The tree-LSTM approach enables RTOS to capture a more accurate representation comparing to DQ and ReJOIN. However, it captures only the table information but ignoring the correlation between tables. Jogger [7] proposes use of random walk to capture the correlation between tables and feed into GCN to get the final table representation. Similar to RTOS, Jogger has column representation, table representation, and query representation as Figure 2.17 shows.

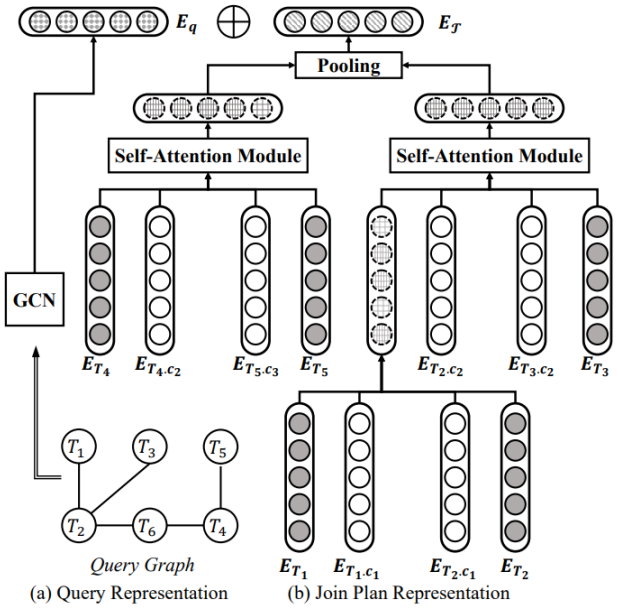


Figure 2.17: Representations in Jogger [7]

For column representation, Jogger employs a vector with size 1×4, the first element of the vector represent whether the column participate in a join, the rest 3 entities represent the selectivity partition, if the estimated selectivity is $< \frac{1}{3}$, the column is assigned to the the first partition, and so on. For table representation, it adopts similar approach as RTOS, using pooling layer to get table representation from the concatenation of column representation. However, in order to enhance the quality of the table representation, Jogger uses a random walk to capture additional information such as the connection between tables. The query representation is identical to that of RTOS, although it differs in that it feeds into GCN rather than fully connected layers. Jogger proposes a self-attention tree to reduce the complexity of the join plan representation. This approach has fewer parameters to learn than tree-LSTM, which facilitates the efficiency.

2.3.4 A Comparative Analysis of the State-of-the-art in ML-based Query Optimization

This subsection presents a summary of the various research areas in query optimization, including tables that compare different approaches to cardinality estimation, cost modeling, and join order selection.

Category	Producer	Pros	Cons
Cardinality Estimation	Blackbox (Malik et al.)[28]	• Groups query with similar syntax as templates. • Feature Extraction from attributes, operators, constants and aggregates.	• Performance depends on the availability and quality of historical query execution data.
	MSCN (Kipf et al.)[1]	• Provides more accurate carinality estimation with simple encoding.	• Only testing on simple query with maximum 4 joins.
	Fauce (Liu et al.)[26]	• First take uncertainty into consideration and propose a robust model	• Numerous parameter need to learn

Table 2.1: Comparison of cardinality estimation models

Category	Producer	Pros	Cons
Cost Model	Neo (Marcus et al.) [31]	• First end-to-end model including cardinality estimation,cost model and plan search algorithm.	• Does not adapt schema changing.
	Bao (Marcus et al.) [30]	• Embedding on top of query optimizer • lightweight model that take advantage on decades of wisdom on DBMS. • adapt schema changing	• The use of hint sets has a impact on the entire process, which precludes Bao from limiting hints to a specific segment of a query plan. • Needs experts to carefully craft hint sets.
	Balsa (Yang et al.) [54]	• Aims finding top-k plans instead of one. • Safe execution with dynamic timeout strategy. • Safe exploration with beam search and prioritize unseen plans.	• Could not adapt schema changing.

Table 2.2: Comparison of cost models

Category	Producer	Pros	Cons
Join Order Enumeration	ReJOIN (Marcus et al.) [32]	• Encoding height of the tree in the state representation	• Could not adapt schema changing. • Different join order could have the same representation.
	DQ (Krishnan et al.) [21]	• Use true execution time to train the model which would be more accurate.	• Could not adapt schema changing. • Different join order could have the same representation.
	RTOS (Yu et al.)[53]	• Adapts to schema changing. • Directly captures the plan tree as input to generate the state representation.	• Missing relations between tables.
	Jogger (Chen et al.)[7]	• Using tree-structured attention mechanism	• Does not adapt schema changing.

Table 2.3: Comparison of join order selection models

All of the models mentioned in the previous section rely to some extent on feedback from traditional DBMSs to train. In order to enhance the reliability of the cost model and its ability to select optimal plans, a learned model is adopted. The use of ML to train a model to replace the cost model demonstrates remarkable achievement. However, it is crucial to ensure that the cardinality estimator is performing well, as inaccurate estimation will propagate further into the cost model and potentially lead to suboptimal results. All of the studies discussed above typically include data engineering to extract the features of the queries. The more information is extracted, the more accurate the representation is proposed. With regard to join order selection, a significant improvement over dynamic programming is that it avoids making the same mistakes repeatedly. In other words, it learns from historical experience. Nevertheless, the different levels of embedding, or the feature extraction from different levels plays an important role. We select RTOS as the base model is motivated by two key considerations. Firstly, the structure of the tree-LSTM architecture aligns perfectly with the tree-structured execution plan, enabling the capture of sequence information, such as join orders. Secondly, the training process is divided into two phases, the estimated cost training phase and the true latency tuning phase. Estimating a cost of a query is much faster than obtaining the true latency. The cost training phase enables the model to gain prior knowledge of the data distribution, which allows it to avoid the join orders that result in long execution times. Consequently, it reduces the overall training time.

Chapter 3

Background - Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a topic that combines deep learning (DL) and reinforcement learning (RL). DL and neural networks are closely related concepts in the field of machine learning (ML). A neural network is a computational model that simulates the structure and function of a biological neural network, which consists of numerous neurons. Each neuron receives signals from other neurons and generates output signals. The neural network is responsible for enabling the model to learn complex data representation and map the data into higher dimensions, thereby capturing the relationship or pattern of the data. Deep learning is typically defined as a neural network comprising multiple layers, in contrast to traditional neural networks. This allows the network to learn more abstract and complex representations, which are essential for many applications. The core principle of deep learning is the iterative process of learning through the use of layers. The network's capacity for deep understanding and analysis of input data is enhanced as each layer learns increasingly complex representations from the previous layer. The advent of deep learning has been a pivotal factor in the evolution of neural networks. The construction of deep neural network models enables the resolution of numerous complex tasks across various domains, including natural language processing and recommendation systems. However, the decision-making abilities of DL are limited. Conversely, RL is capable of decision-making but is unable to learn complex data representations. Consequently, the integration of these two approaches, which complement each other's strengths, represents a potential solution to the perceptual decision-making challenge faced by complex systems.

3.1 Deep Learning and Neural Network

A deep learning model is typically composed of multiple different neural networks, and a neural network is typically comprised of several key components, arranged in a hierarchical manner. These include the **input layer**, **hidden layer**, and **output layer**:

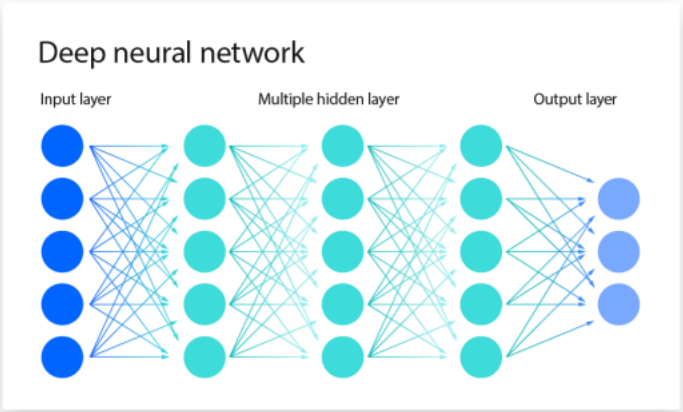


Figure 3.1: Deep neural network [15]

Input layer: The input layer is the initial point of interaction between the raw input data and the neural network. Its primary purpose is to prepare the raw input data for further processing by the subsequent layers (hidden layers). The input layer typically does not contain any neurons and is only used for data transfer.

Hidden layer: The hidden layer is an intermediate layer between the input and output layers. Its function is to extract or transfer input data features. The hidden layer may vary depending on the problem and the architecture of the network. It could be a linear layer, a convolutional layer, or a pooling layer.

Output layer: The output layer is the final layer of the network and is responsible for transferring the learned features to the final prediction or classification result, which depends on the task. For example, in a binary classification problem, the output is of size two.

Activation functions: In addition to these layers, another crucial component is the activation function. This enables the model to learn complex mappings between the inputs and outputs, as it introduces non-linearity into the neural network. The most common activation functions are relu, sigmoid, tanh, and softmax.

Loss functions: The training of the neural network is to identify the optimal function that represents the difference between the predicted value (output of the neural network) and the actual value (label). This is achieved by minimizing the loss function through back propagation using policy gradient.
These are the fundamental components of neural networks. Over decades of research, numerous neural networks with varying perspectives have been proposed, including graph

neural networks (GNNs), convolutional neural networks (CNNs), and others. In addition to the aforementioned networks, there are numerous other variants.

3.2 Layers in Neural Networks

The most important components of a neural network are its layers, particularly the hidden layers, which typically including different types of layers. This section will provide an overview of the diverse types of layers that comprise a neural network.

3.2.1 Linear Layer

The linear layer, also known as the fully connected layer or dense layer, is responsible for performing a linear transformation that maps the input to the output. Each node in the layer is connected to each node in the previous layer, and each connection is associated with a weight that controls the influence of the input data. To enhance flexibility, the layer also incorporates a bias. To be more precise, the layer follows the linear function:

$$y = Wx + b \tag{3.1}$$

In Equation 3.1, y represents the output vector, x denotes the input vector, W is the weight matrix, which are learnable parameters that are updated using back propagation. b is the bias, which is also a learnable parameter. The linear layer performs a linear combination of the input data x and obtains the output y by adding the bias b. The weight matrix W represents the linear relationship between the input and output.

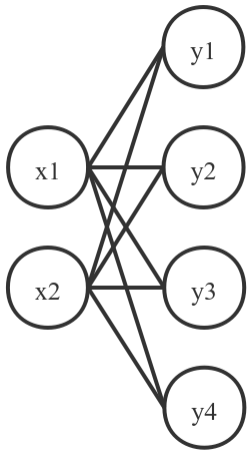


Figure 3.2: Linear layer

Figure 3.2 depicts a fully connected layer, where x represents the input of the linear layer and y represents the output. The calculation is presented below:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$Y = WX + B = \begin{bmatrix} x_1 w_{1,1} + x_2 w_{2,1} + b_1 \\ x_1 w_{1,2} + x_2 w_{2,2} + b_2 \\ x_1 w_{1,3} + x_2 w_{2,3} + b_3 \\ x_1 w_{1,4} + x_2 w_{2,4} + b_4 \end{bmatrix}$$

However, the transformation is linear and therefore unable to capture complex non-linear relationships. The addition of non-linear activation functions could enable the neural networks to learn more complex non-linear relationships. The linear layer is typically used at the end of a model.

3.2.2 Convolutional Layer

The convolutional layer [39] is a fundamental component of image processing algorithms. Its primary function is to extract local features from input data through convolutional operations. Each convolutional kernel slides over the input data and computes the result, generating a corresponding feature map.

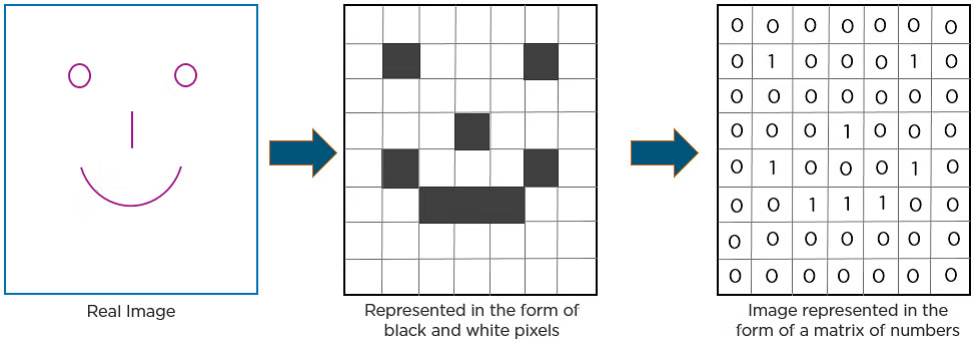


Figure 3.3: Image to matrix

As illustrated in Figure 3.3, the input image is represented by a matrix, which can be utilized for input into the subsequent layer. Prior to commencing the calculation process, two crucial components must be considered: the kernel and the stride number. The kernel is a matrix that slides over the input matrix, extracting localized features, thereby enabling the neural network to learn complex representations. The stride number controls the distance the kernel moves each time it slides over the input data. The following illustrative example provides a further explanation:

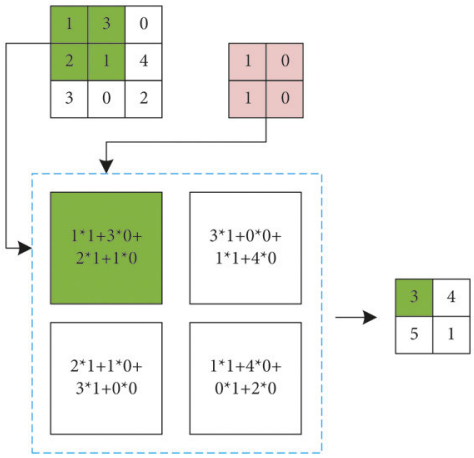


Figure 3.4: Example of computation of a convolution calculation [25]

In Figure 3.4, the 3×3 matrix represents the input matrix, while the 2×2 matrix, painted in pink, represents the kernel. The green portion of the matrix represents the position of the kernel. In this case, the stride number is set to 1, which indicates that the kernel slides to the right by one position each time.

3.2.3 Pooling Layer

The pooling layer [8] is commonly used in convolutional neural networks. It typically employs after the convolutional layer and has the function of reducing the spatial size of the data, thereby reducing the computational complexity of the model and extracting important features. There are two major types of pooling strategies: max pooling and average pooling.

Max pooling

Max pooling is a feature extraction and dimension reduction method that selects the maximum value within each region. This approach preserves the most representative features within the region as Figure 3.5 shows.

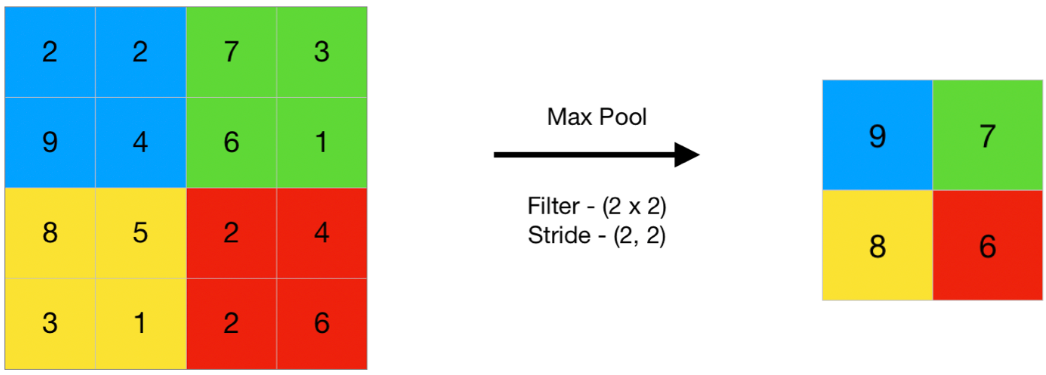


Figure 3.5: Example of max pooling layer [43]

Average pooling

The purpose of average pooling is identical to that of max pooling. However, former differs in that the average pooling calculates the average value within each region as Figure 3.6 shows.

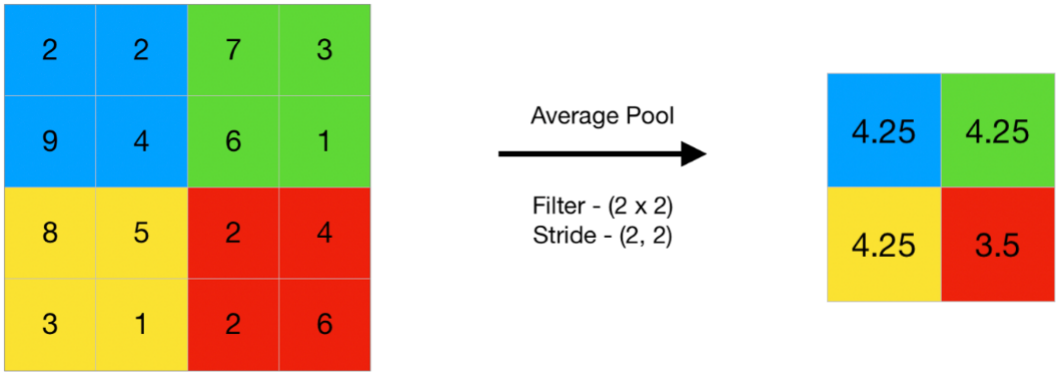


Figure 3.6: Example of average pooling layer [43]

3.3 Activation Function in Neural Networks

The activation function is also a crucial component in neural networks, serving to map inputs to outputs and introduce non-linearities that enable the neural network to learn and approximate more complex functions. In the absence of an activation function, the output of a neural network is a linear combination of inputs, regardless of the number of layers it possesses. Consequently, such a network is unable to learn and approximate complex functions. The selection of an appropriate activation function is dependent upon the specific purpose for which they are employed. The following subsections present a number of different activation functions.

3.3.1 Sigmoid

The sigmoid function is a non-linear activation function that compresses the input between 0 and 1. This approach is widely used in early neural networks for converting outputs to probabilities. However, since its gradient is close to 0, it can lead to the problem of gradient vanishing, which can affect the training of deep neural networks. The calculation is described below:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

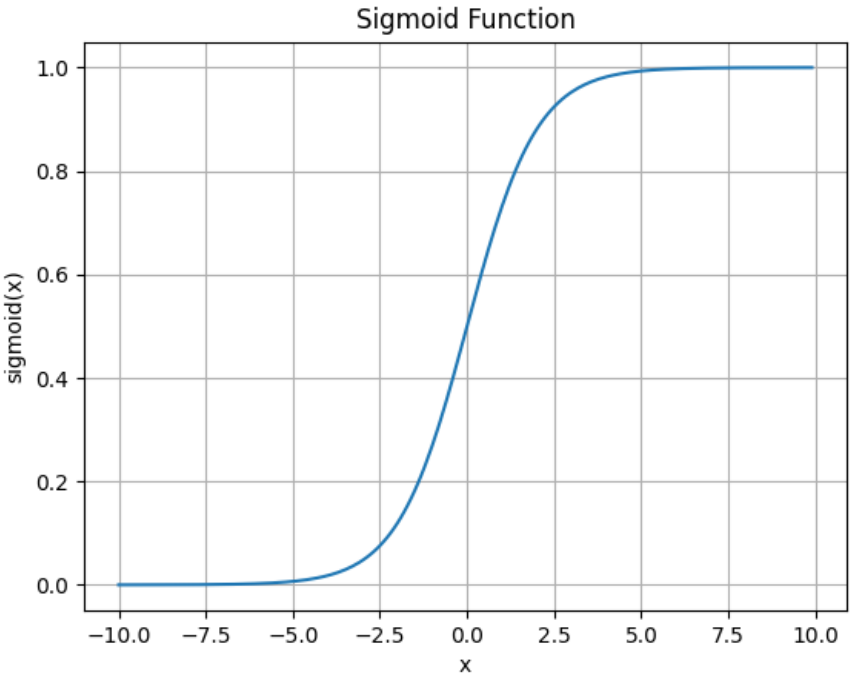


Figure 3.7: Sigmoid function

3.3.2 Tanh

Tanh is a variant of sigmoid where the input is compressed between -1 and 1. In comparison to sigmoid, the output of Tanh is centered around 0, which makes the training process more stable. However, it suffers from the same problem of vanishing gradients. The calculation is described below:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

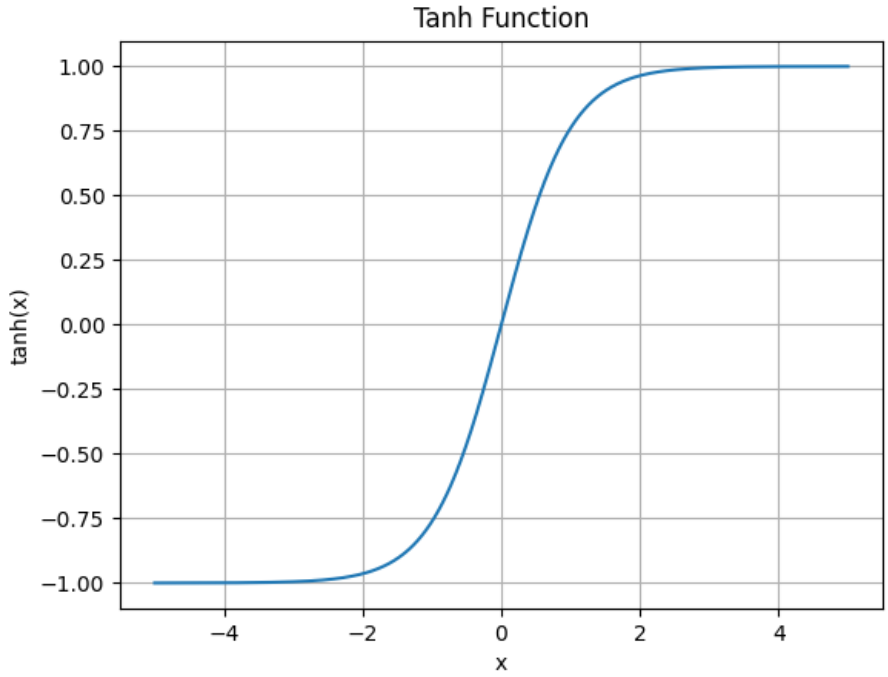


Figure 3.8: Tanh function

3.3.3 ReLU

ReLU is one of the most widely used activation functions. Its simplicity and computationally efficiency during the training process are due to its characteristics. When the input is positive, the output of ReLU is equal to the input, and the output is 0 when the input is negative. This effectively alleviates the gradient vanishing problem, but ReLU causes another problem that 'kills' the neuron, since the gradient is always 0 and can never be activated again. The calculation is described below:

$$\text{ReLU(x)} = \begin{cases} 0 \; if \; x \; \leq 0 \\ x \; if \; x \; > \; 0 \end{cases}$$

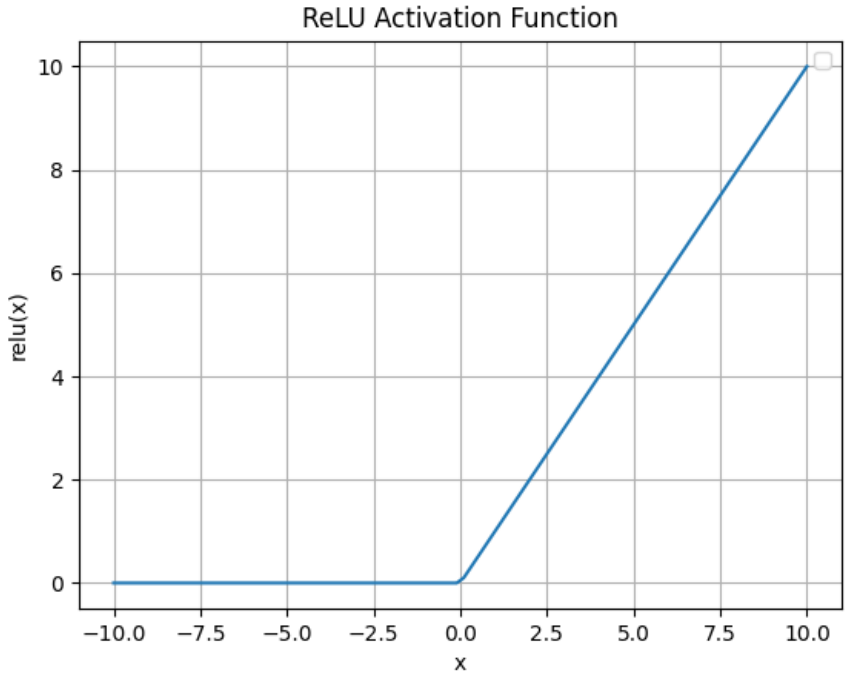


Figure 3.9: ReLU function

3.4 Types of Neural Networks

As previously stated, there are numerous different types of layers. We explain the construction of different types of neural networks in this section. Neural networks are usually combined with multiple layers and assembled to address specific tasks and challenges. The most well-known neural networks are GNN [44] and RNN [45].

3.4.1 Graph Neural Network

GNN is a deep learning model that employs neural networks to learn graph-structured data and extract and discover the features and patterns in graph-structured data. These features and patterns can be utilized to address various graph learning tasks, including clustering, classification, prediction, segmentation and generation. The core operation of a graph neural network is graph convolution, which updates the representation vector of a node by aggregating information from its neighbors and itself. There are numerous classical models of graph neural networks, including graph convolutional network (GCN) [20], which is one of the earliest GNN models. In addition to GCN, modern GNN algorithms typically integrate the graph convolutional layer with the graph attention layer to create a novel layer known as the transformer convolutional layer [47]. GNNs have a diverse range of applications in numerous fields, including social network analysis and medicinal chemistry.

In the context of social network analysis, GNNs can be employed for a range of tasks, including user classification, link prediction, and recommendation systems.

Graph

A graph G is a special data structure comprising nodes V and edges E. A node represents an object, while an edge represents a connection between objects. There are two types of graphs: directed and undirected. In an undirected graph, the nodes connected by edges are bidirectional, and a directed graph means that the nodes connected by edges have direction that follows a from-to relationship. To illustrate the concepts in greater detail, we use an illustrative example.

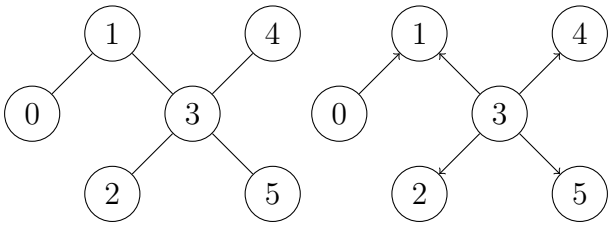


Figure 3.10: The left graph is undirected, the right graph is directed

Figure 3.10 illustrates two graphs, the left-hand graph G_l representing an undirected graph and the right-had graph G_r representing a directed graph. The distinct characteristics of these two graphs, despite their similar connections, result in different representations. Both G_l and G_r have the same node set where $V_l = V_r = \{0, 1, 2, 3, 4, 5\}$. However, the vertex set E_l and E_r are distinct, where $E_l = \{(0, 1), (1, 0), (1, 3), (3, 1), (4, 3), (3, 4), (2, 3), (3, 2), (5, 3), (3, 5)\}$ and $E_r = \{(0, 1), (3, 1), (3, 4), (3, 2), (3, 5)\}$. Consequently, the adjacency matrix would be different. In this scenario, the neighbours of a node would be disparate. For example, in G_l, the neighbours of node 1 are 0 and 3, whereas in G_r, node 1 has no neighbours because node 1 does not have any edges connected to other nodes.
In GNN, each node typically contains its own representation, which varies depending on the task. For instance, in a social network, a node could represent a user, and the node representation could include information about the user's height, width, habits, and activity level. Additionally, the edges could represent link information to different users. During training, each node collects information from itself and its neighbours, which result in the undirected graph collecting information in a different manner than the directed graph.

Transformerconv

The advent of NLP models such as the *Transformer* [50] has led to a surge in the popularity of attention mechanisms. These mechanisms offer a flexible and powerful means of selectively focusing on relevant aspects of input data, thereby enhancing models' ability to effectively capture informative representations. Nevertheless, some researchers have incorporated the attention mechanism into GNNs and proposed a novel graph convolutional

layer, namely *Transformerconv* [47]. The self-attention mechanisms assign different weights to different parts of the input data, thereby improving the model's focus on critical information. The output is calculated based on the weighted input. In other words, attention mechanisms act as an additional weight to the input data. In the *Transformer* [50] model, self-attention plays an important role that enables the model to compare each element in the input sequence with other elements when processing the sequence, thereby ensuring the correct processing of each element in a diverse range of contexts. The self-attention mechanism comprises three key input matrices: the query matrix Q, the key matrix K, and the value matrix V (value). These matrices are derived from the input sequence through a series of linear transformations, and the parameter d denotes the dimension of the hidden layer, and $\sqrt{d}$ is a normalization technique to stabilize and control the scale of dot products. The three matrices allow the computation of attention for each element via the following formulas:

$$a = softmax(\frac{Q \times K^T}{\sqrt{d}}) \times V \tag{3.2}$$

Transformerconv embeds the self-attention mechanism in GCN with the calculation below:

$$x'_i = W_1 x_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} W_2 x_j$$

where the attention factor $\alpha_{i,j}$ is computed with formula below:

$$\alpha_{i,j} = softmax(\frac{(W_3 x_i)^{\mathsf{T}} (W_4 x_j)}{\sqrt{d}})$$

x represents a node that contains node features. It updates itself by gathering information from its neighbours. For each node, it computes the attention for each of the neighbours and finally gets the updated representation of itself.
In conclusion, GNNs, as a powerful deep learning model, are continuously developing and improving. They provide an effective tool for processing graph-structured data and has the potential to be applied in a number of fields.

3.4.2 Recurrent Neural Network

Feed-forward neural networks are unable to process input data that exhibits dependencies and sequential patterns. This is due to the lack of correlation between the previous input and the current input, which results in all the outputs being independent. The recurrent

neural network (RNN) [45] is a special network that addresses this issue. In contrast to feed-forward neural networks, RNNs are capable of processing temporal information in sequential data through recurrent connections, which confers an advantage in processing sequential data such as text, temperature prediction, stock market quotes, and so on. Such tasks typically require historical data to train the model in order to predict certain amounts of data in the future. The fundamental structure of an RNN is a recurrent unit, which comprises of an input and an output, in addition to a hidden state. The input of an RNN is the input data of the current moment and the hidden state of the previous moment, while the output is the output data of the current moment and the hidden state of the current moment. This cyclic connection method enables RNN to retain the historical information when processing sequence data, and to utilise this information to influence the output and hidden state at the current moment. One of the most well-known RNN models is Long Short-Term Memory (LSTM) [12], and its variant, Tree-structured Long Short-Term Memory (tree-LSTM) [48].

Traditional RNN

RNN [45] is a type of neural network utilized for processing sequential data. IN contrast to CNN, RNN is capable of handling data that varies in sequence. For example, the meaning of a certain word may differ depending on the preceding and subsequent words. RNN is able to solve such issues. The traditional RNN structure can be conceptualized as a "loop" comprising of multiple neurons. Each neuron receives input information and produces an output, which is then used as input for the subsequent neuron.

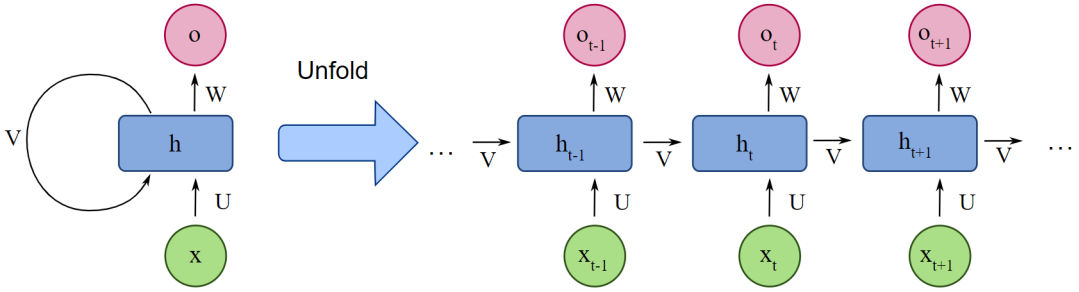


Figure 3.11: traditional RNN [16]

Figure 3.11 illustrates RNN process, where x represents the input, o represents the output and h represents the hidden state. The hidden state is a key component of RNN, influencing the output in a manner that is highly dependent on the hidden state. In this case, the state of the current time step t is determined by the inputs of this time step and the hidden state of the previous time step. The computation of this process as follows:

$$h_t = \sigma(W_{t-1}^h + W^i x)$$
$$o = \sigma(W^o h_t)$$

The output is influenced not only by the input, but also by the hidden states h, which demonstrate dynamic time-series behavior.

Long Short-Term Memory

LSTM [12] is a special RNN that is more suitable for processing and predicting tasks with long intervals in a time series than traditional RNNs. Traditional RNNs are capable of processing short sequences of data, but they have difficulty when the sequence is long. LSTM effectively solves the long sequence problem by introducing the concepts of memory cells, input gates, output gates and forgetting gates as shown in Figure 3.12.

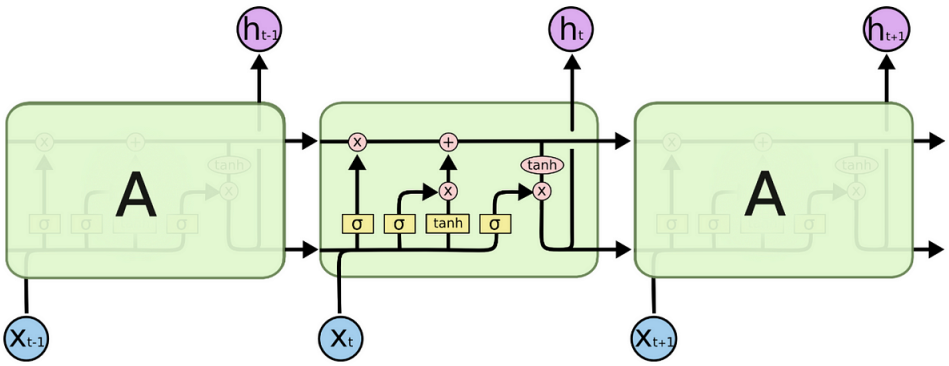


Figure 3.12: LSTM network [49]

LSTM architecture has a comparable structure to the traditional RNN, with the addition of the memory cell c. The memory cell is responsible for carrying important information across different time steps. In order to understand how this information flow is controlled, we must first consider the gates. These gates regulate the input and output of information to the cell state, thereby enabling the model to flexibly control the process of remembering or forgetting over time.

forget gate: The forget gate controls the flow of information from the previous cell state to the current cell state, and determines whether to discard the information in the memory cell or not. The forget gate takes input x_t from the current state and the hidden state h_{t-1} from the previous unit as follows:

$$f_t = \sigma(W_f[x_t, h_{t-1}] + b_f)$$

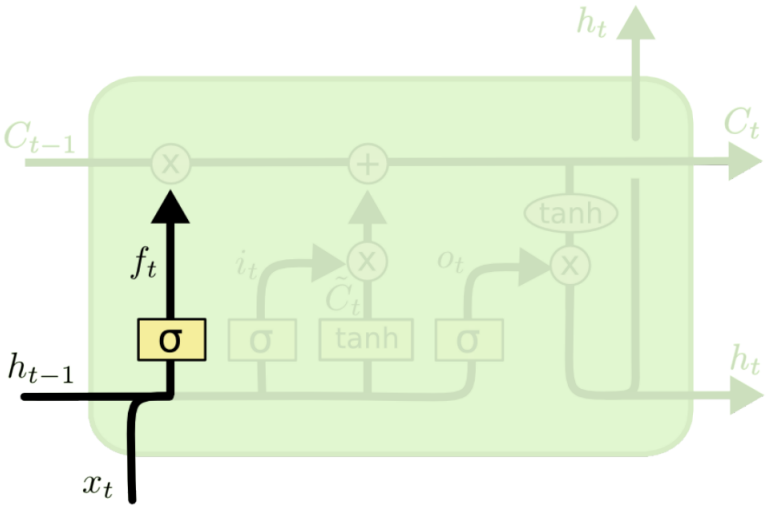


Figure 3.13: LSTM network - forget gate [49]

The sigmoid activation function compresses the input between 0 and 1 and acts as a gate that determines which information from the previous cell state c_{t-1} should be retained or forgotten, where 0 corresponds to the part of the previous memory to be forgotten, and 1 corresponds to the part of the previous memory to be retained. W_f is the weight matrix and b_f is the bias vector specific to the forget gate.

input gate: The input gate controls which information from the current input and the previous hidden state is incorporated into the cell state. The input gate receives the input x_t from the current state and the hidden state h_{t-1}. Since it is necessary to discard the previous memory and receive a new memory, it also requires the output from the forget gate. This results in the generation of a new state of memory C_t, using the following formula:

$$
\begin{aligned}
i_t &= \sigma(W_i[x_t, h_{t-1}] + b_i) \\
\widetilde{C}_t &= tanh(W_c[x_t, h_{t-1}] + b_f) \\
C_t &= f_t * C_{t-1} + i_t * \widetilde{C}_t
\end{aligned}
$$

It is important to note that we have a candidate cell state $\widetilde{C}_t$, which can be incorporated into the cell state once it has been obtained. Subsequent to this, the aggregation of the candidate cell state into the cell state for the current time step can then be performed. The sigmoid function, analogous to the forget gate, compresses the input between 0 and 1, thereby determining which information is allowed to pass through and which is discarded. W_i is the weight matrix and b_i is the bias vector specific to the input gate, W_c is the weight matrix used to compute the candidate cell state.

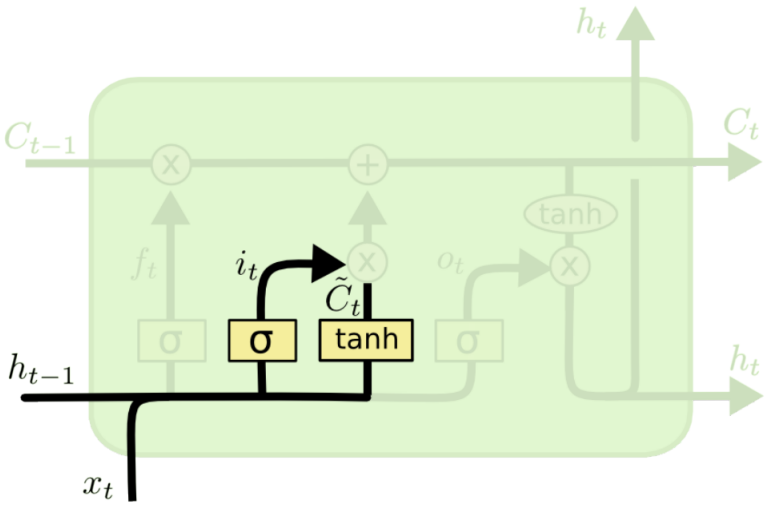


Figure 3.14: LSTM network - input gate [49]

output gate: The output gate is the final gate in LSTM, which ensures that the model produces the appropriate output h_t based on the current input and the updated cell state.

$$o_t = \sigma(W_o[x_t, h_{t-1}] + b_o)$$
$$h_t = o_t * tanh(C_t)$$

As with the other two gates, the sigmoid function compresses the input between 0 and 1, thereby determining the extent to which the updated cell state should be revealed as the output. W_o represents the weight matrix, while b_o denotes the bias vector specific to the output gate.

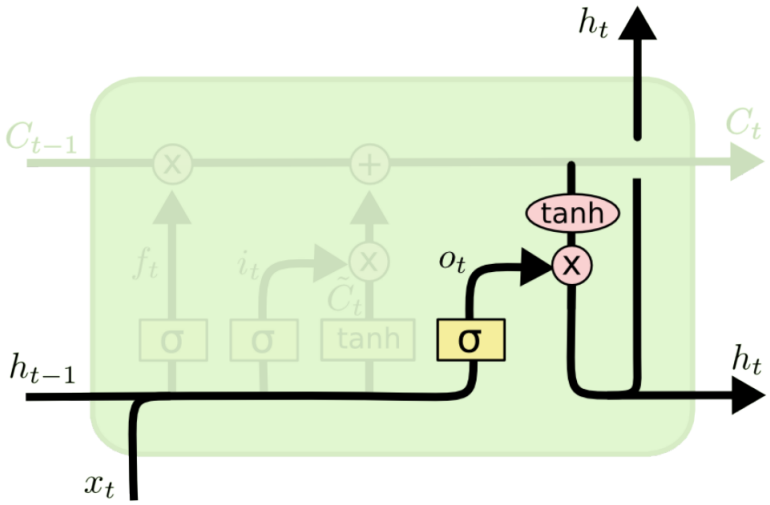


Figure 3.15: LSTM network - output gate [49]

The combination of these gates enables the control of the flow of information over long sequences, thereby rendering LSTM a highly effective characteristic for modeling sequential data with complex dependencies.

Tree-structured long short-term memory

Tree-LSTM [48] is a type of recurrent neural network that has been specifically designed to handle hierarchical structures such as trees (e.g. parse trees) in natural language processing. In contrast to traditional LSTMs, the gates and cell state updates in tree-LSTMs depend on all the cell states associated with the child nodes. Moreover, tree-LSTM has multiple forget gates, corresponding to each child node of the current unit. This architectural approach enables the tree-LSTM to selectively preserve information flow from disparate children. Two distinct types of tree-LSTM have been proposed, n-ary tree-LSTM and child-sum tree-LSTM.

N-ary LSTM: N-ary tree-LSTM is a variant of tree-LSTM in which the internal node has N children. It is employed when the size of the input is fixed. It has the same gates as the LSTM, but each child has its own forget gate. The calculation is a bit complex, for simplicity, we assume $N = 2$ and decompose the computation step by step.

Denotations

- t : current time step
- h : hidden states, h_t is the hidden states under current time step while h_{t-1} denotes the hidden states from previous time step.

- j : current internal node
- k : children of internal node, k=2 in binary tree-LSTM

Forget gate

$$
\begin{aligned}
f_{jk} &= \sigma(W^f x_j + \sum_{n=1}^{N} U^f_{kn} h_{jn} + b^f) \\
f_{j1} &= \sigma(W^f x_j + U^f_{11} h_{j1} + U^f_{12} h_{j2} + b^f) \\
f_{j2} &= \sigma(W^f x_j + U^f_{21} h_{j1} + U^f_{22} h_{j2} + b^f)
\end{aligned}
$$

The weight matrix W_f and the bias vector b_f, are specific to the input of the forget gate. U^f_{kn} is the weight matrix for the hidden states for the children. The assignment of a forget gate to each of the children allows for more flexible control over the flow of information from the children.

Input gate

$$
\begin{aligned}
i_j &= \sigma(W_i x_j + \sum_{n=1}^{N} U^i_n h_{jn} + b^{(i)}) \\
i_j &= \sigma(W_i x_j + U^i_1 h_{j1} + U^i_2 h_{j2} + b^{(i)}) \\
\widetilde{c}_j &= tanh(W_u x_j + \sum_{n=1}^{N} U^u_l h_{jn} + b^u) \\
\widetilde{c}_j &= tanh(W_u x_j + U^u_l h_{j1} + U^u_l h_{j2} + b^u)
\end{aligned}
$$

The weight matrix W_i and the bias vector b_i, are specific to the input of the input gate. U^i_n is the weight matrix for the hidden states of the children.

Output gate

$$
\begin{aligned}
o_j &= \sigma(W^o x_j + \sum_{n=1}^{N} U^o_n h_{jn} + b^{(o)}) \\
o_j &= \sigma(W^o x_j + U^o_1 h_{j1} + U^o_2 h_{j2} + b^{(o)}) \\
c_j &= i_j \odot \widetilde{c}_j + \sum_{n=1}^{N} f_{jn} + c_{jn} \\
c_j &= i_j \odot \widetilde{c}_j + (f_{j1} \odot c_{j1}) + (f_{j2} \odot c_{j2}) \\
h_j &= o_j \odot tanh(c_j)
\end{aligned}
$$

The weight matrix W_o and the bias vector b_o, are specific to the input of the output gate, U_n^o is the weight matrix for the hidden states of the children.

Child-sum tree-LSTM

In contrast to the n-ary tree-LSTM, child-sum is designed for an unknown number of children in the tree. Child-sum tree-LSTM employs a significantly simpler calculation than the n-ary tree-LSTM:

$$\begin{aligned}
\widetilde{h}_j &= \sum_{k \in C(j)} h_k \\
i_j &= \sigma(W^i x_j + U^i \widetilde{h}_j + b^i) \\
f_{jk} &= \sigma(W^f x_j + U^f h_k + b^f) \\
o_j &= \sigma(W^o x_j + U^o \widetilde{h}_j + b^o) \\
\widetilde{c}_j &= tanh(W^u x_j + U^u \widetilde{h}_j + b^u) \\
c_j &= i_j \odot \widetilde{c}_j + \sum_{k \in C(j)} f_{jk} \odot c_k \\
h_j &= o_j \odot tanh(c_j)
\end{aligned}$$

C_j denotes the set of children of node j, child-sum tree-LSTM sums up all the hidden states in its children, denoted $\widetilde{h}_j$. It then updates through a similar process, to that of the n-ary tree-LSTM. Each node has its own forget gate, which controls the information flow.
In conclusion, n-ary tree-LSTM is capable of capturing sequential information, but only permits a fixed number of children for each internal node. In contrast, the child-sum tree-LSTM could handle trees with different numbers of children in the internal nodes, but it loses the sequential information of its children.

3.5 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning that involves a model, or agent, learning to make decisions by interacting with an environment, also known as learning by trial and error [46]. A RL model is constructed from five essential components: **agent**, **environment**, **state**, **action**, and **reward**.

3.5.1 Markov Decision Process

The Markov Decision Process (MDP) [4] is a mathematical framework utilized in decision-making problems under uncertainty. It includes the state information and transition functionality between different states. It is worth noting that the state transitions of MDP must satisfy the Markov property, whereby the next state and reward depend solely on the given current state. The primary components of an MDP are:

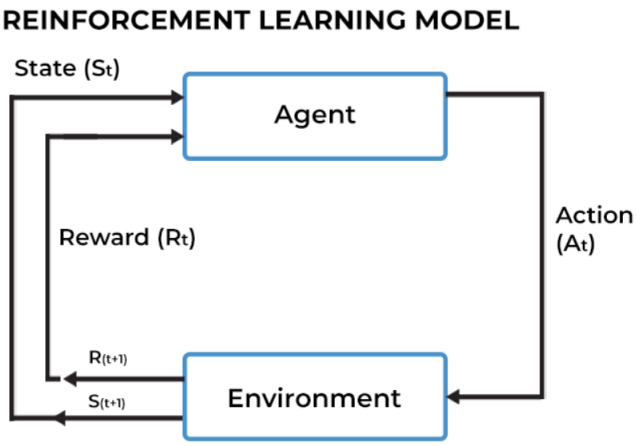


Figure 3.16: A pictorial representation of reinforcement learning model

State ($\mathcal{S}$): $\mathcal{S}$ is a set of states that describes the environment in which the agent operates, where $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. The state is transferred to another state after the agent takes a given action.

Action ($\mathcal{A}$): $\mathcal{A}$ is a set of actions that describes the actions that the agent could take under given states, where $\mathcal{A} = \{a_1, a_2, ..., a_m\}$. Action spaces can be either discrete or continuous. Discrete action spaces denote a certain movement such as "move up", or "move down". Continuous action spaces are a number in a given range such as "robot arm move 40 degrees to the left".

Transition Probability (p): The probability of transitioning from a given state s and action a to a new state s', is given by the following equations: $p(s, a, s') = p(S_{t+1} = s'|s_t = s, a)$

Reward (r): The reward is the feedback provided by the environment after the agent takes a certain action under a given state to transition to another state, where $r = r(s_t, a_t, s_{t+1})$.
The agent and the environment interact at discrete time steps such that $t = 0, 1, 2, 3, ..., T$. At each time step, the environment provides the agent with a representation of the state of the environment, $s_t \in S$. The agent then chooses an action a based on the state, $a_t \in A(S_t)$. Once the agent takes an action and transitions to a state, it receives a reward $r(s_t, a_t, s_{t+1})$, and observes a new state s_{t+1}. The core problem in RL is to find a solution to a problem, and more importantly, to find the optimal solution to the problem efficiently. This can be illustrated by the example of finding an exit in a maze with the least steps. In order to achieve the desired outcome, MDP must identify a policy for the agent, which is denoted as π. This specifies the action that the agent is likely to take given a particular state. The goal of MDP is to identify the optimal policy π, which maximizes the accumulated reward,

and the expected return serves as a measure of how valuable it is for the agent to be in a particular state or to perform specific action within that state, under a certain policy:

$$G_t = \sum_{t=0}^{T} \gamma^t r(s_t, a_t, s_{t+1}) \tag{3.3}$$

In 3.3, t represents the time step, while γ is a discount factor between 0 and 1 that controls the relative importance of future rewards. When $\gamma = 1$, the agent considers future rewards with no discount, which means that the agent values future rewards just as much as immediate rewards. Conversely, when $\gamma = 0$, the agent ignores future rewards entirely and only focuses on maximizing the immediate reward for the next action. To solve the core problem, the optimal value function can be found using value iteration. Value iteration is a dynamic programming algorithm used to find the optimal policy for a MDP. It iteratively refines the value estimates for each state until they converge. It directly computes the optimal state values, by iteratively applying the Bellman optimality equation as equation 3.4 demonstrates.

$$v(s) = max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')] \tag{3.4}$$

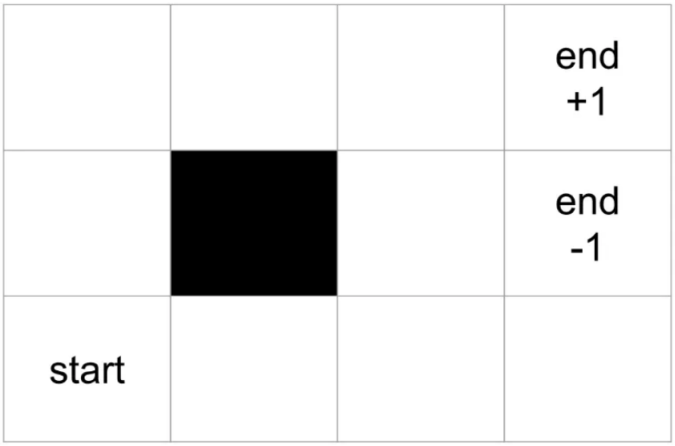


Figure 3.17: Grid world example [55]

Grid World Example: The environment is constructed with 12 grids, each representing a state s where $s \in S = \{s_1, s_2, ..., s_{16}\}$. The action $a \in A = \{UP, DOWN, LEFT, RIGHT\}$ is applied to the agent in each state. The initial state s_{init} is represented by the **start** grid, and the agent receives a reward of either +1 or -1 on the 'end' grid, which represents the termination states. The **episode** starts on the "start" grid, and there is a wall grid that is painted black. The goal in the grid world is for the agent to navigate to the correct goal location (+1) and avoid the incorrect goal location (-1), which is penalized with a negative reward.

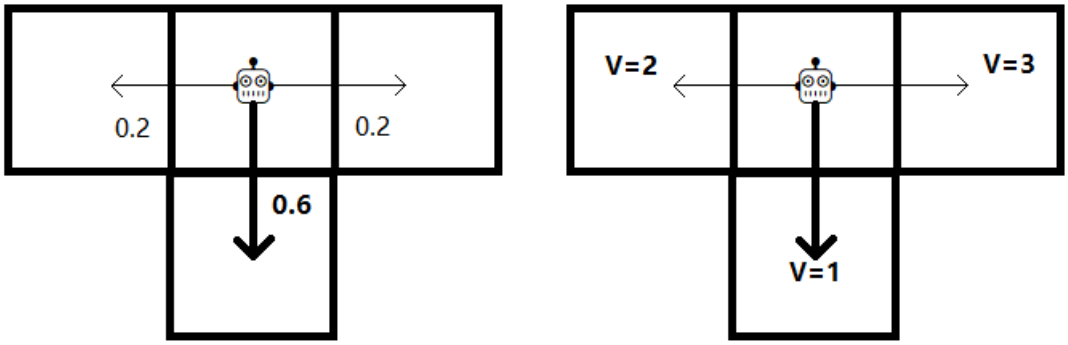


Figure 3.18: Value iteration example

Value Iteration Example: The agent is located at the center of the grids and can obtain V(s) and R(s) for all its neighbouring grids. The agent is unable to move up. The probability of moving 'LEFT' and 'RIGHT' is 0.2, while the probability of moving 'DOWN' is 0.6. Assuming a discount factor $\gamma = 0.9$, the grid world example has no immediate reward unless it reaches the goal. The value function of the grid can be computed as follows:

$$v(s) = r(s) + \gamma \sum_{s'} p(s'|s, a = a_i) v(s')$$

$$\begin{aligned}
V(s) = (& \\
& (0 + 0.2 * (0.9 * 2))(left) \\
& + (0 + 0.2 * (0.9 * 3))(right) \\
& + (0 + 0.6 * (0.9 * 1))(down) \\
& + (0 + 0 * (0.9 * 0))(up)) \\
V(s) = & (0.36 + 0.54 + 0.54) \\
V(s) = & 1.44
\end{aligned}$$

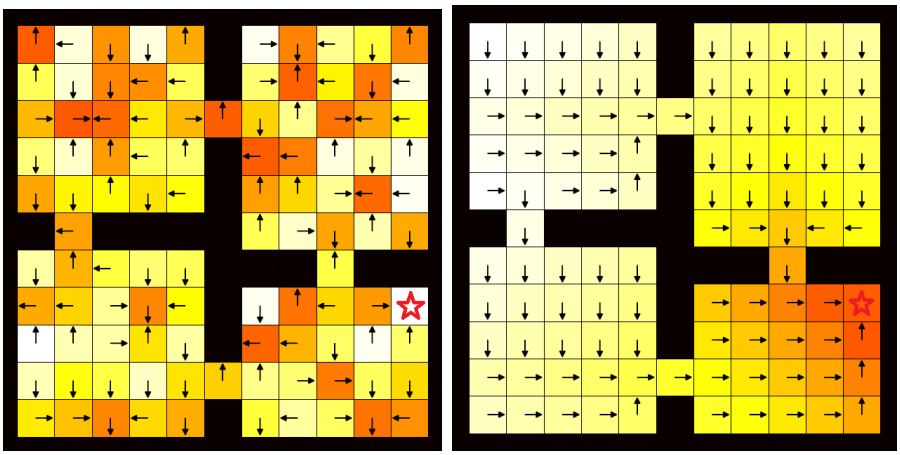


Figure 3.19: Left figure is the arbitrary initialled value function, right figure is converged value function

The state value for the grid on which the agent is located is 1.44. To facilitate a more comprehensive understanding of the algorithm, an additional illustrative example is provided to see the difference between the arbitrary value function at the initialization stage and the value function following a number of iterations until convergence, as shown in Figure 3.19.

3.5.2 Q-learning

In fact, value iteration only provides a feasible idea in a real-life scenario. However, the agent does not know the state transition probabilities P. Instead, the agent explores the environment and discovers which action gives the highest reward. To enhance the efficacy of value iteration, q-learning [52] has been proposed as an alternative. In q-learning, the agent typically stores the action-value pair $Q(s, a)$ in a q-table of size of $N \times M$, where N is the number of states and M is the number of actions. In the case of the grid world, which has 11 states and 4 actions, the size of the q-table would be $11 \times 4 = 44$. The q-value [52] represents the cumulative expected return starting from the current state s, by taking action a, and following the policy in effect at that time. The q-value accounts for both the immediate reward of taking the action a and the future reward that will be obtained by taking action a. Q-learning is an algorithm that is widely used in RL, particularly in scenarios where the agent lacks sufficient knowledge about the environment. It learns the optimal policy by interacting with the environment. Figure 3.20 illustrate an example of how a model learns through trial and error. Q-learning aims to identify the policy that maximizes the q-value for each state-action pair. A common strategy for exploring the environment is to use the epsilon-greedy policy, where epsilon ϵ is a fraction between 0 and 1 that controls the agent's behavior. It determines whether the agent should explore a new state that may yield a better reward or exploit current knowledge to select the current known-good action that yields appropriate reward. This is illustrated by Equation 3.5.

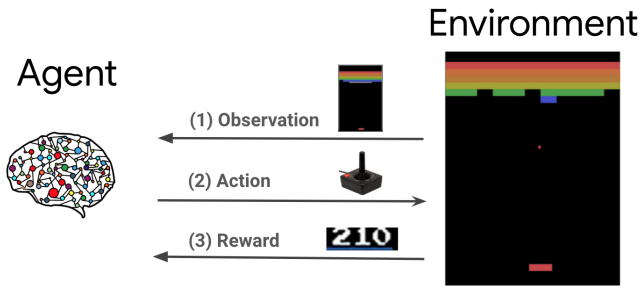


Figure 3.20: How does a model learn by trial and error [2]

$$a = \begin{cases} Random\ action & with\ probability\ \epsilon \\ max_{a \in \mathcal{A}}\ Q(s,a) & with\ probability\ 1-\epsilon \end{cases} \tag{3.5}$$

The q-learning update rule is very similar to value iteration, in which transition probabilities are not considered:

$$Q(s,a) = Q(s,a) + \alpha[r(s,a,s') + \gamma max_{a'} Q(s',a') - Q(s,a)] \tag{3.6}$$

$Q(s,a)$ is the current estimated q-value for the state-action pair (s,a). The learning rate, denoted by α, is a real number between 0 and 1 that is used to control the weight given to new information compared to existing knowledge. This formula iterates over the states within time steps, updating the value of each state based on the values of its neighbouring states until it converges. The final converged values for each state represent the optimal value function, which can be used to determine the optimal policy.

Algorithm 1 Q-Learning with ϵ

1: Initialize:
2: Arbitrary q-values $Q(s,a)$ for all state-action pairs
3: Loop until termination or reaches number of episodes:
4: Choose an action based on epsilon greedy equation and observe reward
5: $a = \begin{cases} Random\ Action & with\ probability\ \epsilon \\ max_a Q(s, a \in \mathcal{A}) & with\ probability\ 1-\epsilon \end{cases}$
6: Update $Q(s,a)$ using equation:
7: $Q(s,a) = Q(s,a) + \alpha[R(s,a,s') + \gamma max_{a'} Q(s',a') - Q(s,a)]$

The core idea of q-learning is to approximate the optimal action-value function through the Bellman optimality equation. Q-learning can be employed to learn in a small, naive with limited state and action spaces. It relies heavily on the hand-crafted feature representation.

However, when the state space and action space become large, it becomes impractical to maintain a large q-table. However, to address this challenge, there has been a growing interest in combining deep learning models to extract features and learn approximations using q-learning. This approach, known as deep reinforcement learning (DRL), has been extensively studied [36].

3.6 Deep Reinforcement Learning

In 2016, deep learning (DL) [23] was invented, focusing on artificial neural networks (ANNs) [17]. In the following year, a growing number of researchers began combining DL with RL to learn a function that approaches the optimal solution, which is called deep reinforcement learning (DRL).

3.6.1 DQN

As previously stated, RL is only capable of handling the environments with simple state and action spaces when utilizing a q-table. However, as the state space $\mathcal{S}$ and action space $\mathcal{A}$ become increasingly complex, it becomes infeasible to create a q-table that contains all state-action pairs. Instead of having a value function, an approximate value function $Q(s, a; W)$ may be trained by neural networks to approximate the optimal value function $Q^*(s, a)$ where $Q(s, a; \theta) \approx Q^*(s, a)$. This approach is known as a deep q-network, and θ represents the parameter to be learned in the neural networks. The input of the deep neural network is the state, and the output is the score for all the possible actions, which enables the agent to select an appropriate action. In order to train the neural network in an RL manner, we need to formalize DQN update formula:

$$L(\theta) = ((r + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^{target})) - Q(s, a; \theta^{pred}))^2 \tag{3.7}$$

Similar to q-learning, r represents the immediate reward received after taking action a under state s, γ is the discount factor that determines the importance of the future reward, s' denotes the next state after taking action a. Finally, a' is the action taken under the next state s'. Instead of having one q-network, deep q-learning employs two q-networks: the prediction q-network, and the target q-network. Having two networks can enhance the stability of the training process. Both of the networks are identical at the initialization phase, and the target q-network copies all the parameters from the prediction q-network periodically. The target q-network can be conceptualized as a fixed target, with a relatively lower frequency of updates compared to the prediction q-network. In addition, the target network incorporates another important component, the replay buffer. The replay buffer serves as a memory that stores the experiences (state, action, reward, next state, done) observed by the agent during interactions with the environment. By storing and replaying experiences, the replay buffer enables the agent to learn from past experiences and break

Algorithm 2 DQN

Initialize:
  Prediction Q network Q^{pred};
  Target Q network Q^{target};
  Copy Parameter from Q^{pred} to Q^{target};
  (make sure the two networks are identical at initialization)
for episode = 1,M **do**
  initialize the environment and observe s_0
  for time step t=1,T **do**
    select action a_t based on ϵ-greedy
    Agent executes the action and observes reward r_t and s_{t+1}
    if done **then**
      break
    end if
  end for
  Sample random transitions(s_t, a_t, r_t, s_{t+1}) from Replay Buffer
  $y_t = \begin{cases} r_t & \text{if episode terminates at step t+1} \\ r_t + \gamma max_{a'} Q^{target}(S_{t+1}, a') & \text{else} \end{cases}$
  Back Propagation with Loss function $(y_t - Q(s_t, a_t))^2$
  Every N episodes, $Q^{target} = Q^{pred}$
end for

temporal correlations in the data, thereby improving the sampling efficiency and stability of the training process. The complete DQN learning process is described in Algorithm 2.

Overall, deep q-learning and q-learning are very similar in terms of the target value and the manner in which the value is updated. The primary distinctions between the two are that deep q-learning integrates q-learning with deep learning to approximate the value function through the use of neural networks, whereas q-learning maintains a q-table to store the value.

Chapter 4

Introduction of GTDD

The previous chapter presents a summary of both traditional and modern solutions to the join order selection problem. Traditional approaches employ dynamic programming which breaks complex problems into simpler subproblems and stores the solutions to those subproblems to avoid redundant computations. Modern methods adapt RL to train an agent for join order selection and to avoid the selection of disastrous join orders for incoming queries. Our approach is based on RTOS, which leverages two distinct types of tree-LSTMs that perfectly align with the structure of the join plan. The tree-LSTMs are capable of capturing the sequence of the join orders. Furthermore, RTOS has two phases: an estimated cost training phase and a latency tuning phase. This structure allows for more efficient training. Existing research primarily focuses on capturing join order sequences and their representations at various levels, with limited comparative analysis of reinforcement learning methods. In this paper, we propose GTDD, a novel framework that integrates Graph Neural Networks (GNN), Tree-structured Long Short-Term Memory (Tree-LSTM), and Dueling-DQN. We construct a schema graph using the foreign-primary key pairs and utilize an algorithm named *Node2Vec* [10] to create embeddings of the relations between tables. Instead of using an adjacency matrix to represent the tables participating in the join, we use the adjacency matrix to construct a join graph, wherein each node contains the table embedding and the vertices represent two tables that are joined together. In order to gain a more comprehensive understanding of the intermediate state of the environment, we capture both link information and table information from the join graph. Furthermore, we replace the vanilla DQN with a dueling-DQN to improve convergence time and enhance the stability of the training process.

Figure 4.1 illustrates the framework of GTDD for an episode. The training query set is split into three partitions, and the training data is sampled from the partitions. The sampled query passes into the feature extraction component, where it is extracted into three levels of representation: column representation, table representation, and query representation. These representations are then fed into different neural networks to acquire the initial state. The agent selects an action based on the state information, and the state after taking the selected action passes into the Tree-LSTM network to obtain a new intermediate state. This process repeats until the terminal state is reached, where all the tables are joined together. The join order is then passed to the underlying DBMS to obtain the reward

(e.g. cost), and the result with the join order passes into the memory pool to update the model. GTDD is based on DRL, which aims to identify the optimal join order that produces the minimum cost by interacting with the environment. The model takes a query as an input and outputs the desired join order to execute in order to obtain the reward. However, the model is not only based on RL, but also based on representation learning [5]. The utilization of capturing informative representation has the potential to enhance the training process. Similar to RTOS [53], GTDD has two phases, the cost training phase and the latency tuning phase. The cost is estimated by the underlying optimizer, while the latency is the actual execution cost. The objective is to accelerate the entire training process, as obtaining true latency typically takes a considerable amount of time when the plans are disastrous. The initial phase of the process involves training GTDD using the estimated cost to gain prior knowledge of the data distribution. This is followed by a second phase in which the true latency is used to fine-tune GTDD in order to achieve the ultimate objective, finding the join orders with the lowest actual cost. In this section, we first introduce the methods we use for representation learning at different levels of representation, and then proceed to introduce the DRL method.

4.1 Representation Learning

Representation learning [5], also known as feature learning, aims to learn meaningful features directly from the data, allowing models to better understand and utilize complex structures within it. Essentially, representation learning enables computers to learn how to extract relevant features. This process transforms raw data into vectors, which are then fed into neural networks to extract information. Before delving into a more detailed discussion of representation learning, it is necessary to describe the information contained in a query. Below is an example of a query following a Select-Project-Join pattern, which includes target table information, table information, and column information:

$$\begin{aligned}&\textbf{SELECT } T_1.c\\&\textbf{FROM } T_1, T_2\\&\textbf{WHERE } T_1.a = T_2.a \textbf{ AND } T_1.b < 10;\end{aligned}$$

The target table, T_1, is the repository of information from which the user wishes to retrieve data. The column, c, specifies the column of interest to the user. T_1 and T_2 are the tables involved in the query. The notation $T_1.a = T_2.a$ indicates that the two tables participate in a join condition on column a. $T_1.b$ is a selection predicate that acts as a filter to filter out the tuples that do not satisfy the condition. We deploy various neural networks to learn representations at different levels: column level, table level, query level, and state level, each providing unique insights and contributing to a more comprehensive understanding. In order to facilitate a more comprehensive understanding of the concept of representation learning, we present a query as an illustrative example and explain in detail in the following

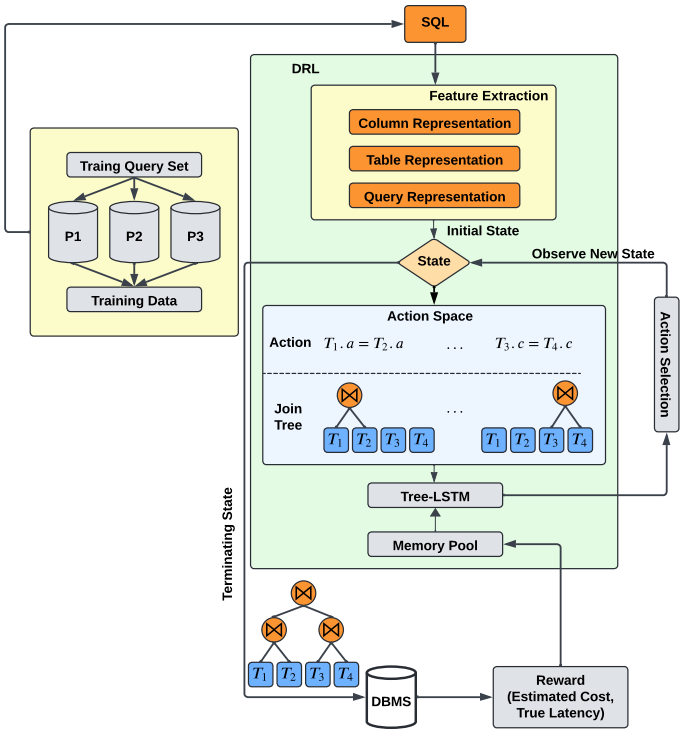


Figure 4.1: Framework of GTDD

subsections:

$$
\begin{aligned}
&\textbf{SELECT } T_1.c \\
&\textbf{FROM } T_1, T_2, T_3, T_4 \\
&\textbf{WHERE } T_1.a < 40 \\
&\textbf{AND } T_1.a > 60 \\
&\textbf{AND } T_1.d \textbf{ BETWEEN } 10 \textbf{ AND } 20 \\
&\textbf{AND } T_2.b = T_3.b \\
&\textbf{AND } T_1.c = T_4.c;
\end{aligned}
\tag{4.1}
$$

4.1.1 Column Representation Learning

For each query Q, there are two types of predicates. The first type is the selection predicates, which refer to one table and serve as a 'filter', eliminating columns that do not satisfy the condition. The second type is the join predicate, which refers to two tables that join together on the given column. Figure 4.2 illustrates that each column c in a table is represented by a vector of size 6. The elements of this vector are defined as follows:

$$F(c) = [c_{\bowtie}, c_{=}, c_{<}, c_{>}, c_{\leq}, c_{\geq}]$$

where $c_{\bowtie}$=1 indicates that the column participates in a join, and the other operators refer to the selectivity of the column with respect to different operators. We treat the operator

"BETWEEN" with both $c_{<=}$ and $c_{>=}$ for the upper and lower bounds, respectively. Since data in different columns has different scales, we normalize the selectivity into a ratio between 0 and 1 based on the uniqueness. Additionally, a matrix $M(c)$ is defined with shape $[6, hs]$ for each column, where hs is the size of the hidden layer and the matrix contains learnable parameters. The final column representation is $R(C) = F(C) * M(C)$.

Example:

Consider Query 4.1, $F(T_1.a) = (0, 0, 0.4, 0.6, 0, 0)$ since column $T_1.a$ has no join condition, "=", "≤", nor "≥" predicates. Consequently, we set $c_{\bowtie}$, $c_{=}$, $c_{\leq}$ and $c_{\geq}$ to 0. Assume column $T_1.a$ contains 100 unique data, ranging from 1 to 100. In this case, we set $c_{<}$ to $\frac{40}{100} = 0.4$ for $T_1.a < 40$ and $c_{>}$ to $\frac{60}{100} = 0.6$ for $T_1.a > 60$. The final column representation for $T_1.a$ is $F(T_1.a) * M(T_1.a)$.

4.1.2 Table Representation Learning

For each table t with n columns, the combination of column representations is used to represent the table. While extracting the information from the columns of the table is important, the correlation between the tables also provides semantic information particularly for join predicates. We leverage the schema of the database to extract correlation between tables and combine the information from both column representations and information of tables from schema to obtain the final representation of tables.

Capture Schema Information in Table Representation

We treat the database schema as an undirected graph $G = (V^T, E^T)$ where V^T is a set of vertices (tables), and S^T is a set of edges (primary-foreign key relations). By capturing the schema information, one can gain more information on the over, which in turn makes the join order selection more straightforward. Once the schema graph has been constructed, we employ the *Node2Vec* [10] algorithm, which is a method for converting nodes on a graph into a vector embeddings without any labels as shown in Algorithm 3. The most common graph sampling strategies are breadth-first search (BFS) and depth-first search (DFS), as shown in Figure 4.3. BFS starts from an unvisited node v in the graph and traverses the neighboring nodes first, and then traverses the neighboring node of each neighboring nodes in turn. BFS starts from an unvisited node v in the graph and traverses along until the end, then go back to the node v to visit another neighbour to the end. Both search strategies iterate until all the vertices are visited. These two methods are usually too extreme, to be able to balance both strategies, node2Vec provides a biased random walk which uses two hyper-parameters p and q to control the walk, it is likely to be a DFS or BFS. First, Node2Vec randomly selects a node T_i as the starting node and performs a walk in the graph. It selects the neighbour to move to on the next step based on probabilities that use p and q to control. This process repeats until reach maximum walk length. After many trails, the sequence of the walks is captured as elements for each table. Second, the algorithm

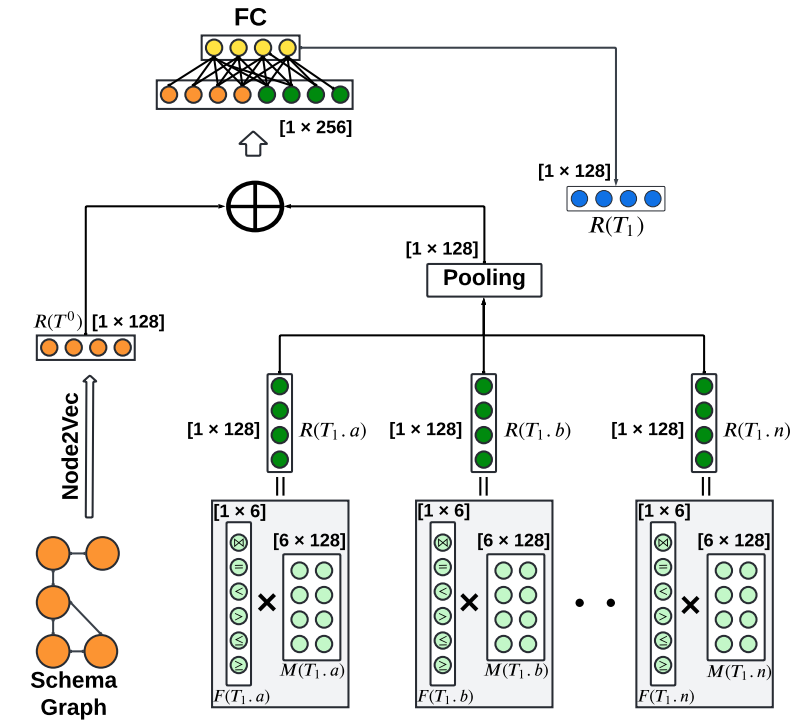


Figure 4.2: Column representation and table representation

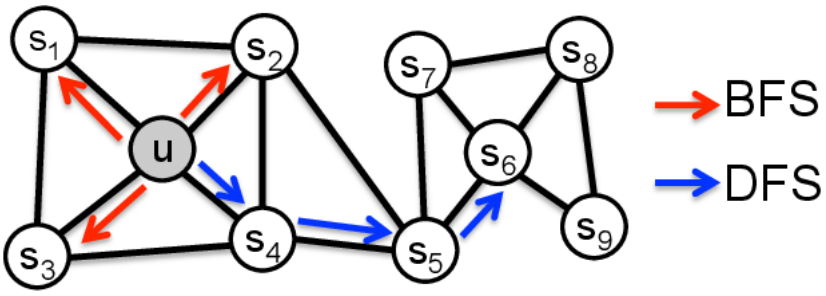


Figure 4.3: BFS vs DFS [10]

learns the representation from the sequence collects in the first step. We treat the node in the sequence as a word, while the sequence as a sentence. We can take the advantage of the well-learned language model *Skip-Gram* [35] to learn the node representation. By applying these two methods, we have the schema embedding $R(T^0)$ as shown in Figure 4.2

Algorithm 3 Schema Representation

Input: Database Schema ,maximum length of the walk W, Number of walks N, hyperparameter p and q

Output: Schema representations

1: Build a schema graph G = (V^T, E^T) based on the primary-foreign key relationships;
2: **while** not reach the number of walks N **do**
3: **for** node n in V^T **do**
4: **while** not reach the length of the walk W **do**
5: select next node $NextNode$ based on:
6: $NextNode = \begin{cases} \frac{1}{p} & \texttt{move to previous node} \\ \frac{1}{q} & \texttt{move to neighbour node} \end{cases}$
7: **end while**
8: **end for**
9: **end while**
10: Perform Skip-gram algorithm and use an average pooling layer to get the correlation embedding $R(T^0)$ for each table in the database

Example:

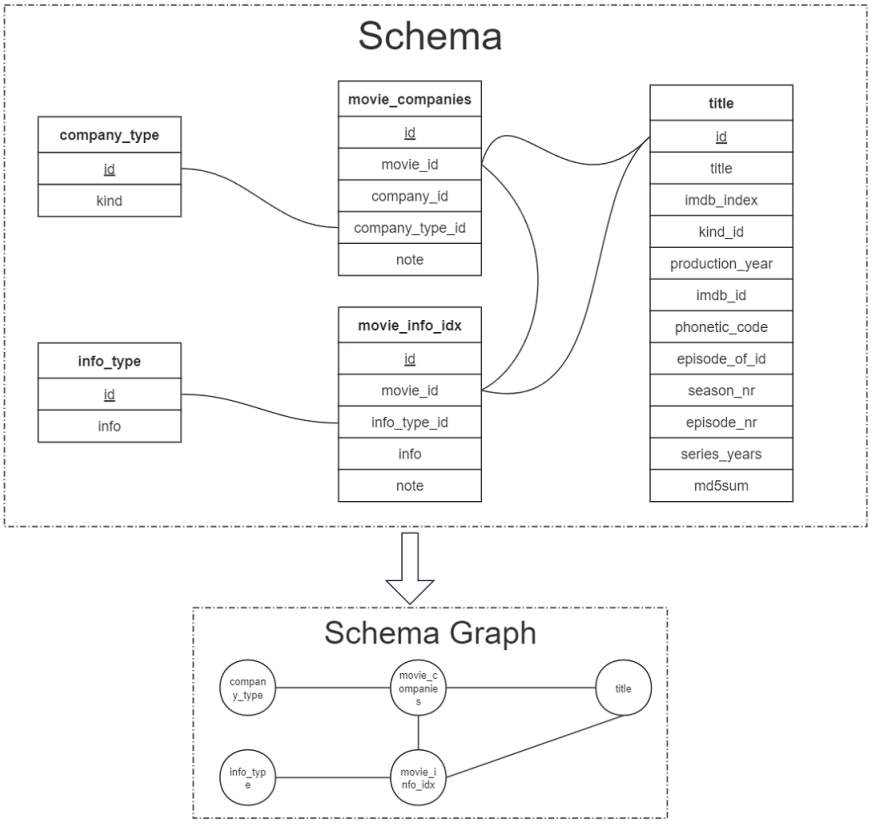


Figure 4.4: Schema graph construction

Figure 4.4 illustrates a database with five tables, and the tables are connected using primary-foreign keys. For instance, Table *company_type* connects to Table *movie_companies* by the column id of *company_type* and company type id of *movie_companies*.

Final Table Representation

As previously stated, the column representation continues to provide indispensable information, as it is the fundamental aspect of the table representation. The final table representation considers both the feature of table from schema and the composition of column embedding. To obtain the final representation, we employ a pooling layer that extracts information from column embedding, and concatenate schema to the result after pooling:

$$R(T) = Pooling(R(T_1.a), R(T_1.b), ..., R(T_1.n)) \bigoplus R(T^0)$$

where $R(T^0)$ is the feature of table from schema embedding and $R(T_1.i)$ represents i-th column embedding for table T_1.

4.1.3 State Representation Learning

The state is a crucial component in DRL which affects the learning performance. It is typically represented by multiple fully-connected layers to map state to dimension. An informative state representation could significantly enhance the performance of the model which is the estimation cost in this case. The join order selection problem usually contains two components of representation: the target query presentation $R(q)$ and the current join forest $R(F)$ which contains a set of join trees. The final state representation is the concatenation of these two components, where $R(s) = R(q) \bigoplus R(F)$.

Target Query Representation

RTOS [53] uses a adjacent matrix m with size $n \times n$ to represent the target query encoding, where n represents the number of tables in the database and $m_{i,j}$ denotes whether tables T_i and T_j participate in a join in the given query, with a value of 1 indicating join exists and 0 otherwise. However, this approach only captures the connection information between the tables, while the information about the tables is ignored. To address this limitation, we construct a join graph using the adjacent matrix and employ a graph neural network [44] to learn the representation of the entire join graph as Figure 4.5 shows.

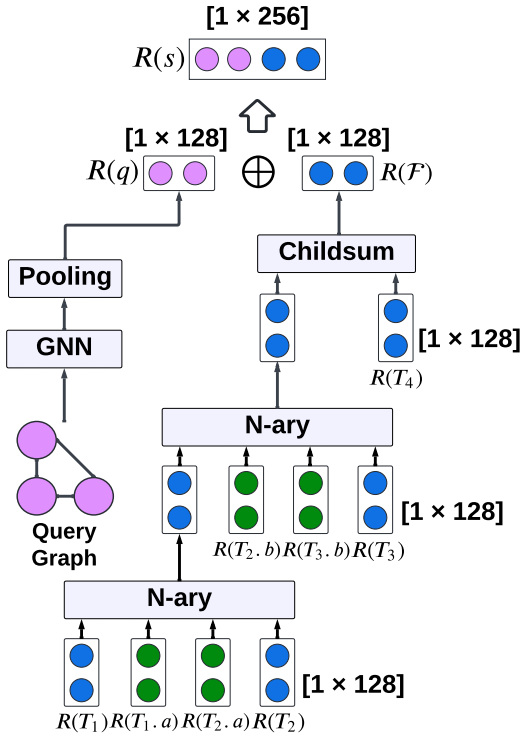


Figure 4.5: State representation

Join Tree, Join Forest and the State Representation

To capture the hierarchical structure inherent in tree-structured data and the data features from a long sequence, we implement the tree-LSTM [48] model. The model captures compositional semantics by recursively combining representations of child nodes to compute the representation of their parent node, thereby enabling the model to capture informative representations. We use two distinct types of tree-LSTM, the child-sum tree-LSTM and the n-ary tree-LSTM.

Child-sum tree-LSTM does not consider the sequence of its children. For a given node j with multiple children $\alpha_{j,k}$, the model computes the sum of representations from all child nodes to construct its own representation.

N-ary tree-LSTM has a fixed number of children and considers the order of its children. For each internal node j, there are N children $\alpha_{j,k}$. The representation of each child $\alpha_{j,k}$ is computed separately and each child node has its own weight matrix. In order to obtain the representation of the forest F which is composed of several join trees $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n$, we employ these two tree-LSTM methods.

1) **Join Tree Representation:** The join tree is comprised of two distinct types of nodes: leaf nodes and internal nodes. A leaf node represents a table or a column, while an internal node corresponds to a join that is composed of 4 nodes, namely, α_0,β_0,β_1,α_1. The nodes α_0 and α_1 represent tables or join trees, while the nodes β_0 and β_1 represent the corresponding columns in the join. As illustrated in Figure 4.5, we integrate both

Algorithm 4 Encodetree (Node n)

Input: Node
Output: Representation of Join tree

```
if n is a leaf Node then
    h = R(n)
    m = Zeros_init()
    return h,m
else
    h_{α0}, m_{α0} = Encodetree(α0)
    h_{α1}, m_{α1} = Encodetree(α1)
    h_{β0}, m_{β0} = Encodetree(β0)
    h_{β1}, m_{β1} = Encodetree(β1)
    return n-aryUnit(h_{α0}, m_{α0}, h_{α1}, m_{α1}, h_{β0}, h_{β1})
end if
```

child-sum tree-LSTM and n-ary tree-LSTM to construct a state representation. Algorithm 4 demonstrates that the tree is constructed recursively. Similar to the traditional LSTM, the tree-LSTM comprises two key components: h_j and c_j, which represent the hidden state and cell state, respectively, for each tree node j.

- If node j is a leaf node, $h_j = R(j)$, j could be either a column or table
- If node j is a internal node representing a join, it must have 4 child nodes α_0, β_0, β_1,

α_1.The representation of these 4 children will feed into n-ary unit in n-ary tree-LSTM to get the representation of node j

The following equations clarify the computation of n-ary tree-LSTM unit for node j:

$$
\begin{aligned}
i_j &= \sigma(W_0^i h_{\beta_{j,0}} + W_1^i h_{\beta_{j,1}} + U_0^i h_{\alpha j,0} + U_1^i h_{\alpha_{j,1}} + b^i) \\
f_{j,0} &= \sigma(W_0^f h_{\beta_{j,0}} + W_1^f h_{\beta_{j,1}} + U_{0,0}^f h_{\alpha j,0} + U_{0,1}^f h_{\alpha_{j,1}} + b^f) \\
f_{j,1} &= \sigma(W_0^f h_{\beta_{j,0}} + W_1^f h_{\beta_{j,1}} + U_{1,0}^f h_{\alpha j,0} + U_{1,1}^f h_{\alpha_{j,1}} + b^f) \\
o_j &= \sigma(W_0^o h_{\beta_{j,0}} + W_1^o h_{\beta_{j,1}} + U_0^o h_{\alpha j,0} + U_1^o h_{\alpha_{j,1}} + b^o) \\
u_j &= tanh(W_0^u h_{\beta_{j,0}} + W_1^u h_{\beta_{j,1}} + U_0^u h_{\alpha j,0} + U_1^u h_{\alpha_{j,1}} + b^u) \\
c_j &= i_j \odot u_j + f_{j,0} \odot m_{\alpha_{j,0}} + f_{j,1} \odot m_{\alpha_{j,1}} \\
h_j &= o_j \odot tanh(m_j)
\end{aligned}
$$

The hidden state h_j represents the intermediate output of the n-ary tree-LSTM, which also represents a join tree $\mathcal{T}$ such that $R(\mathcal{T}) = h_j$

2) **Join Forest Representation:** Once we obtain representations for each join table, we can compose them into a forest, denoted as $F = \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n$. The number of trees in the forest is dynamic and thus we apply child-sum tree-LSTM to handle the issue since it disregards the number of children. The child-sum tree-LSTM not only captures the partial join tree but also considers the tables that have not yet joined. It represents the aggregate information of the entire forest. Similarly to the n-ary tree-LSTM, the child-sum tree-LSTM also contains a child-sum unit, with the calculation being:

$$
\begin{aligned}
h &= \sum_k h_{\mathcal{T}_k} \\
i &= \sigma(U^i h + b^i) \\
f &= \sigma(U^f h_k + b^f) \\
o &= \sigma(U^o h + b^u) \\
u &= tanh(U^u h + b^u) \\
m^{root} &= i \odot u + \sum f_k \odot m_{\mathcal{T}_k} \\
h_{root} &= o \odot tanh(m^{root})
\end{aligned}
$$

As with the join tree representation, the hidden state h^{root} is the intermediate output of the child-sum tree-LSTM, and we use the output as the representation of a join forest $\mathcal{F}$ such that $R(\mathcal{F}) = h^{root}$.

3) **State Representation:** The state representation $R(s)$ would be the concatenation of $R(\mathcal{F}) \bigoplus R(q)$.

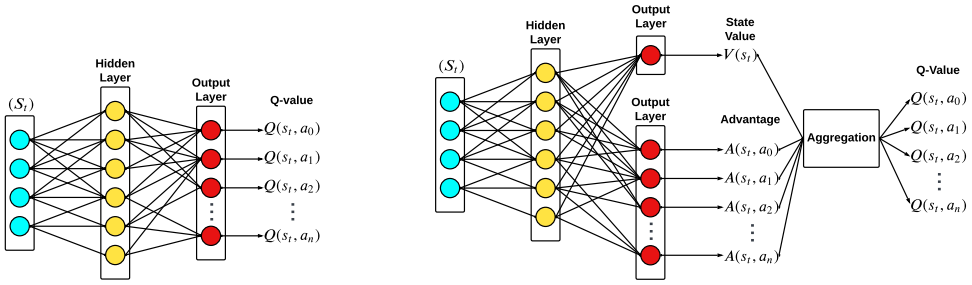
4.2 Deep Reinforcement Learning in Join Order Selection

The join order selection problem can be easily mapped into RL, with the following components:

- **Environment:** Database and the underlying DBMS.
- **State:** Tables are joined together and tables await for a join.
- **Action:** Pick two tables join together.
- **Reward:** 0 in the intermediate state, and a improvement ratio for the terminal state.
- **Agent:** The agent that learns to pick the best join order.

4.2.1 DQN and Dueling-DQN

Due to the discrete action space in the join order selection problem, we employ the value-based strategy RL method, DQN, as the join order selection agent. The simple q-learning approach typically maintains a q-table to store the optimal value for all state-action pairs. However, this exhaustive search approach requires significant memory and is impractical in complex environments due to the large training time. One potential method for reducing the training overheads is to leverage the capabilities of deep neural networks to estimate an unseen state from a previously observed state. This approach could facilitate generalization. The agent selects an action based on the deep q-network $Q(R(s), a; w)$, shorten as $Q(s, a; w)$, where w is the weight of the fully connected layer, s is the state, and a is the action. The training process typically involves numerous episodes to identify an approximate optimal q-value function. An episode represents a complete process in which the agent generates a single join tree, in which all tables in the query are joined together, and observes the feedback v. Although previous work demonstrates remarkable results, the training stability and convergence time remain potential drawbacks, and none of these works use different RL methods to stabilize the training process. We discover one potential solution to stabilize the training process, Dueling-DQN [51], which is an enhancement of the DQN architecture. As Figure 4.6a demonstrates, DQN is a single neural network that directly estimates the q-values Q(s,a) for each action in a given state, and the q-value is used to guide the agent to select the best action under the given state. Different than DQN, dueling-DQN separately estimates state value and the advantage of each action. Equation 4.2 demonstrates the process of aggregation of value stream and advantage stream, value stream, estimates the state value $V(s)$, which indicates how good it is to be in a given state, without considering any specific action, and action stream, estimates the advantage function $A(s, a)$ for each action, which measures the relative quality of each action in that state, and $\frac{1}{|\mathcal{A}|}\sum_{a' \in \mathcal{A}} A(s, a'; \theta)$ is the normalization term that computes the average



(a) Deep Q Network (b) Dueling Deep Q Network

Figure 4.6: DQN vs Dueling-DQN

advantage over all possible actions a' in state s.

$$Q(s, a; w) = V(s; \theta) + (A(s, a; \theta) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \theta)) \tag{4.2}$$

The network takes the state s as input and eventually outputs q-values for all possible actions, where the action space is defined as $\mathcal{A}_s = \{a_1, a_2, ..., a_n\}$. In order to achieve a balance between exploitation and exploration, the action is controlled by epsilon ϵ, with the agent randomly selecting an action with the probability ϵ and choosing the action with the maximum estimated q-value with probability $1 - \epsilon$. Both DQN and dueling-DQN have two networks, one is the predict network $Q^{predict}$ and the target network Q^{target}. Both networks are initialized with the same structure and the same parameters. The target network periodically copies the parameters of the predict network as an update. The loss function is the standard MSE, which aims to minimize the gap between $Q^{predict}$ and Q^{target}:

$$L(\theta) = (y - Q^{predict}(s, a; w))^2 \tag{4.3}$$

$$y = r + \gamma max_{a'} Q^{target}(s', a'; w) \tag{4.4}$$

$$r = log(\frac{DP_{feedback}}{GTDD_{feedback}}) \tag{4.5}$$

Equation 4.3 is the loss function that shows the difference between the predicted value and expected value, where Equation 4.4 shows the expected value that sums target value and reward. It is worth noting that if state s' is a terminal state, the environment ends the episode, and no further rewards or actions are available. Therefore, the target formula simplifies to $y = r$.

4.2.2 Reward

The reward function is a crucial component in DRL as it directly impacts the behaviour and performance of the agent. It defines the agent's objective by providing feedback on its

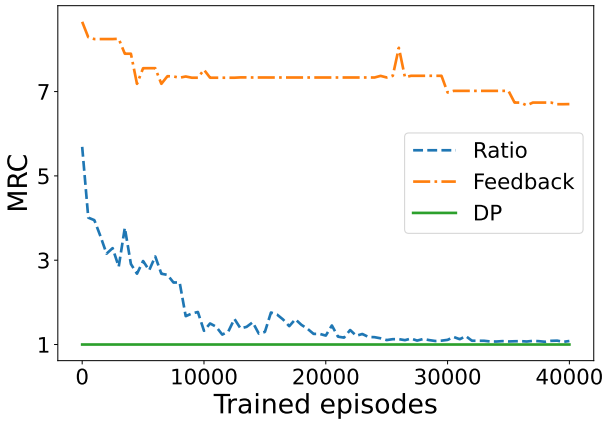


Figure 4.7: Feedback as reward vs ratio as reward

actions and directing it towards the desired outcomes. Recent research, including DQ [21], ReJoin [32], and RTOS [53], establishes a reward of 0 for all intermediate states. For the terminal state, the reward value is set to the feedback of the entire execution plan as provided by the DBMS. However, as Figure 4.7 shows, using the feedback as a reward leads to selecting suboptimal query plans. We infer that this may be due to the varying ranges of feedback for different queries. Simple queries tend to have significantly lower costs compared to complex queries. Thus, we express the feedback into a ratio as Equation 4.5 shows.

The feedback of the dynamic programming execution plan is denoted by $DP_{feedback}$, while $GTDD_{feedback}$ is the feedback of the GTDD execution plan. For the sake of illustration, let us assume that $GTDD_{feedback} = 100$ and $DP_{feedback} = 90$. In this case, the reward $r(a)$ is given by the expression $log(\frac{90}{100}) = -0.046$, which is a negative value. This is because the execution plan of DP has a superior better performance than GTDD. If we assume that $GTDD_{feedback} = 90$ and $DP_{feedback} = 100$, then the reward $r(a)$ is equal to $log(\frac{100}{90}) = 0.046$ which is a positive reward since GTDD obtain better performance. This reward function encourages the agent to find a better execution plan that results in positive rewards. In Equation 4.4, the parameter γ represents the discount factor, which indicates the relative importance of future rewards. A value close to 0 indicates that the agent prioritizes immediate rewards over long-term rewards.

4.2.3 Curriculum Learning

The networks are sensitive to parameter changes, which harm convergence and negatively impact the finding of optimal join orders. Instead of training the DQN agent on randomly sampled data from the training set, a curriculum-based approach could be employed, whereby the training queries are sampled from a sequence of increasingly complex or challenging queries. In particular, curriculum learning [6] was inspired by the field of human education, which is a learning approach that progresses from simpler to more challenging tasks. The critical aspect is the curriculum setting. In the join order selection problem, curriculum setting refers to the design of the data partition. We consider the number

of participating tables in a query as the learning difficulty, such that a larger number of tables in a query refers to a larger space that is more complex than the query with less tables. The training data is initially sorted based on the number of joins and equally split into three partitions, designated as $\{p1,\ p2,\ p3\}$, with increasing difficulty. Partition $p1$ contains query with 4 to 7 joins, $p2$ contains query with 8 to 9 joins, and $p3$ contains 10 to 17 joins. The input query samples from these partitions, initially, the training data samples from $p1$. After certain episodes, the training data samples from $p1 \cup p2$, and so on. This approach is based on the rationale that learning join orders of simple queries is easier than complex queries. Prior knowledge of the problem is therefore beneficial to the model, as it allows it to identify optimal join orders more efficiently. Nevertheless, the ultimate goal is to resolve the challenging queries, which necessitate a greater number of episodes to identify optimal join orders. Consequently, it is preferable to dedicate a greater proportion of episodes to more challenging queries, rather than taking too many episodes for the simple query. The process illustrates in Algorithm 5:

Algorithm 5 Curriculum Learning

Input: Training Set $\{q_1, q_2, ..., q_n\}$,number of partition k, updating interval I
Output: Updated Training Dataset $\mathcal{D}$

Sort the training dataset based on the number of tables participating joins
split the data into k partitions $\{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_k\}$
Initialize the empty training set $\mathcal{D} = \emptyset, i = 0$
for $t = 1, 2, ..., T$ **do**
    if t is divisible by I and i<k **then**
        $\mathcal{D} = \mathcal{D} \cup \mathcal{P}_i$
        $i \mathrel{+}= 1$
    end if
end for
return $\mathcal{D}$

4.2.4 Action Mask

The output of DQN is the q-value for all the actions, which usually not piratical in real-life tasks. However, some of the actions are invalid under certain scenarios. For instance, an agent located in the top-left corner of a maze would be able to take the valid action of "DOWN" and "RIGHT", as this would result in the agent moving in the right direction. Similarly, "UP" and "LEFT" are two invalid actions. The action mask mechanism [13], allows for the implementation of a mask for invalid actions, with a value of either 0 or 1. By incorporating this mechanism, it is possible to prevent the agent from taking invalid actions, thereby allowing it to focus on the actions that are valid.

Example

Consider an agent located in the top-left corner of a maze, and the q-value for each action is $Q = \{q(s_0, LEFT), q(s_0, RIGHT), q(s_0, UP), q(s_0, DOWN)\} = \{0.6, 0.1, 0.2, 0.5\}$. The agent selects the action with the maximum q-value which is 'LEFT', but it hits the wall by taking the action 'LEFT'. The action mask mechanism is employed to set invalid actions to 0 and valid actions to 1. The mask is defined as $\mathcal{M} = \{LEFT, RIGHT, UP, DOWN\} = \{0, 1, 0, 1\}$. We could formalize the masking mechanism as follows:

$$Q_{masked}(s, a_i) = \begin{cases} Q(s, a_i) & \text{if } M_i = 1 \\ -\infty & \text{if } M_i = 0 \end{cases} \tag{4.6}$$

By applying the masking mechanism, the result of $Q_{masked}(s_0, a_i)$ would be $\{-\infty, 0.1, -\infty, 0.5\}$. Consequently, the agent selects the valid action 'DOWN' instead of the invalid action 'LEFT' as Figure 4.8 shows. In the context of the join order selection problem, there are

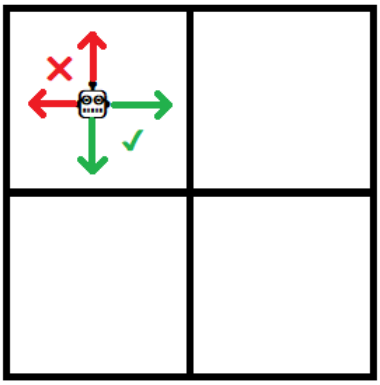


Figure 4.8: Invalid action and valid action

$N \times N$ possible actions, where N is the number of tables in the database. Since $A\ Join\ B$ is equivalent to $B\ Join\ A$, we can eliminate the action space to $\frac{(N \times N - N)}{2}$. Despite this reduction, the action space remains sufficiently large, and the agent is likely to select an invalid action. The implementation of actions masking ensures that all invalid actions are excluded from the action space, thereby enabling the agent to select only valid actions.

4.3 Implementation of GTDD

The dimensions of the hidden layers are 128, and the batch size of the replay buffer is 32. Furthermore, we use the Adam optimizer [19] to update the parameters with a learning rate of 0.003. In order to obtain an informative state representation, we implement both child-sum tree-LSTM and n-ary tree-LSTM for join forest representation learning. In addition, we replace the one-hot encoding with a GNN to capture both the link information and the table information. Furthermore, we have apply two layers of transformerconv [47]

to learn the node representation and use global average pooling to obtain the representation of the join graph.
In the context of curriculum learning, we split the training set into three partitions, designated as $p_1,\ p_2,\ p_3$, in order of increasing number of joins. Each partition is defined to have no overlap with the preceding partition. The updating interval sets to 2,000, with the model initially trained on the first 2,000 episodes based on p_1. At 4,000 episodes, the training set extends to include p_2. At 6,000 episodes, the training set extends to include p_3. The model is then trained on the entire training set until end of training.

Chapter 5

Experimental Study

The objective of our experimental study is to evaluate the performance of our model in comparison to both DRL-based and traditional join order selection approaches. In our setup, we leverage the feedback provided by the PostgreSQL optimizer, such as estimated cost, as a important guiding principle for our model to learn the optimal policy.

5.1 Experiment Setup

All of the experiments are run on an Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz, and NVIDIA GeForce RTX 3070 Ti.

5.1.1 Dataset

We conduct our experiment on two benchmark datasets to evaluate the performance of our proposed method. The first benchmark dataset is Join Order Benchmark (JOB) [24], and the second benchmark is TPC-H [40]. For each dataset, we select 90% of the queries for training and the rest for testing.

Join Order Benchmark (JOB) [24] JOB is a real-world dataset that derived from the Internet Movie Database (IMDB), which is a comprehensive database of information related to films, television series, actors, etc. JOB was specifically designed to evaluate the performance of different join order optimization strategies. The majority of the research related to the join order selection problem uses JOB as a benchmark for evaluating their work. JOB consists of a set of queries that involve joining multiple tables from the IMDB dataset. JOB contains 113 queries derived from 33 templates, with each query containing a varying number of tables, ranging from a minimum of 4 to a maximum of 17. The total size of the database is 3.6G. Moreover, the schema of JOB has 21 tables and 108 columns in total, and we have use the schema file to construct the schema graph to capture the relations between tables. The resulting graph comprises 21 nodes and 49 edges.

TPC-H [40] The TPC-H dataset is a standard benchmark dataset used to evaluate the performance of DBMS, specially designed to simulate a decision support system environment. The total size of the database is 4G. Moreover, the schema of TPC-H has 8 tables and 6 column in total, and we have use the schema file to construct the schema graph. The resulting graph comprises 8 nodes and 10 edges.

5.1.2 Baselines

We compare GTDD with both the traditional methods and DRL-based methods:
- GTD: To evaluate the effectiveness of dueling-DQN, we replaced the dueling-DQN component with a standard DQN in the GTDD framework, naming the resulting model GTD. The only difference between GTDD and GTD would be the reinforcement learning architecture.
- RTOS: RTOS [53] is a DRL-based approach which constructs join forest with tree-LSTMs to capture the sequential join information.
- JOGGER: JOGGER [7] introduces a lightweight, tailored-tree-based attention module designed to effectively capture join orders with fewer parameters to tune compared to RTOS.
- Dynamic Programming [9]**:** Dynamic Programming is a built-in technique in PostgreSQL, which is designed to find the optimal plan with the lowest cost. This method is controlled by a parameter 'geqo_threshold', which determines the extent to which the dynamic programming approach is activated. By setting this parameter to a value greater than the number of tables involved in a query, the dynamic programming approach is activated.

5.1.3 Metrics

As stated in Chapter 4, GTDD encompasses two distinct phases: the cost training phase and the latency tuning phase. The measurement of these two phases is conducted via the application of different metrics. In order to evaluate the performance of GTDD, we employ two metrics that were used by RTOS [53] and Jogger [7]. In the cost training phase, we use Mean Relative Cost (MRC) as the metric, while in latency tuning phase, we use Geometric Mean Relevant Latency (GMRL).

Mean Relative Cost (MRC)

MRC is a method used to evaluate the cost-training phase, It uses cost as feedback to guide the agent in its learning process. We have selected DP as the baseline because DP employs an exhaustive searching strategy and returns the optimal plan. The formula for MRC is presented below:

$$MRC = \frac{\sum_{q \in Q} \frac{cost(q)}{cost_{DP}(q)}}{|Q|} \tag{5.1}$$

If the plan generated by GTDD has the same cost as DP, then MRC would equal 1. A value of MRC closer to 1 indicates a closer approximation to the optimal join order. In Equation 5.1, Q represents the entire dataset, and q is a query within the set. The estimated cost for query q is from GTDD is denoted by $cost(q)$, while the estimated cost for query q from DP is represented by $cost_{DP}(q)$. $|Q|$ represents the size of the dataset, and Equation 5.1 evaluates the average performance on the cost training phase.

Geometric Mean Relevant Latency (GMRL)

GMRL is a method that used to evaluate the latency, which refers to the real-world performance of the model, since minimizing the true latency is the primary goal. We have also selected DP as the baseline. Once the model transitions to the true latency tuning phase, DP executes the plan based on the estimated cost, which may differ significantly from the actual latency. Thus, most of the models could generate better plans than DP after tuning with the true latency. From this perspective, MRC would not be suitable to capture the relative performance. GMRL tends to provide a more balanced measure that reduces the impact of extremely high or low latency values, which might skew the results when using an mean value. For instance, given two queries, if the model generates join plans with 2 and 0.5 compared to the baseline, MRC yields $\frac{2+0.5}{2} = 1.25$. However, the actual improved performance should be $\sqrt{2 \times 0.5} = 1$. The geometric mean is less affected by extremely high or low value compared to the mean.

$$GMRL = (\prod_{q \in Q} \frac{Latency(q)}{Latency_{DP}(q)})^{\frac{1}{|Q|}} \tag{5.2}$$

5.1.4 Data Partitioning

RTOS [53] randomly selects 10% of the queries in the JOB as the test set, while the remaining 90% are used as the training set. However, there is a possibility that the test set may contain patterns that are also present in the training set. To address this issue, we have selected the entire template #10 in JOB, which contains 3 queries, along with 8 other queries randomly chosen from the remaining 110 queries. This approach allows us to test the generalization capabilities of our model. In total, the training set contains 102 queries and the test set contains 11 queries.

5.1.5 Training Sample Collection

In order to evaluate the learning ability of the model, the networks within the model are randomly initiated without any pre-training process during the cost-training phase. This differs from previous work DQ [21] which act as dynamic programming when the number of tables involved in the query is small, and only explores using the model when the number of tables is large. However, in order to evaluate the model's learning ability, our model is only trains using the feedback provided by PostgreSQL.

5.1.6 Training Time

We would not evaluate the training time on the latency tuning phase, given that the latency tuning part typically requires a significant amount of time. In order to reduce the latency tuning time, we maintain a latency pool, which stores all the previous observed join order of each query with the corresponding latency as a key-value pair in a dictionary. In order to further reduce the tuning time, we utilize the maximum execution time of queries in the training set as a reference point to set a timeout. The timeout is set to $5 * max_{latency}(Q)$ which is 10 minutes.

5.2 Evaluation

Although the true latency is the ultimate goal, training with pure latency would be prohibitively expensive. It is probable that the agent explores the environment during the earlier training phase, which may result in the selection of an order that leads to a poor join order, taking a very long time to execute. RTOS [53] proposes a method that uses estimated cost to train the model as a bootstrap and uses true latency to tune the model. This method could significantly reduce the training time. We adopt this method and use cost as feedback to train for 40,000 episodes and use latency as feedback to tune for 20,000 episodes. In the cost training phase, we evaluate the performance using MRC and compare the training time. Furthermore, we extract one entire template from JOB to test the performance of the model on unseen queries. In the latency tuning phase, we evaluate the performance using GMRL, and we also extract one entire template to test the performance of the model on unseen queries, as denoted in previous section, we would not evaluate latency tuning time since we maintain a large latency pool to save the training time.

5.2.1 Cost Training

We first evaluate the **cost training** phase which uses estimated cost as feedback to train the model. We use DP as the baseline and evaluate the performance by comparing other implementations with mean relative cost (MRC).

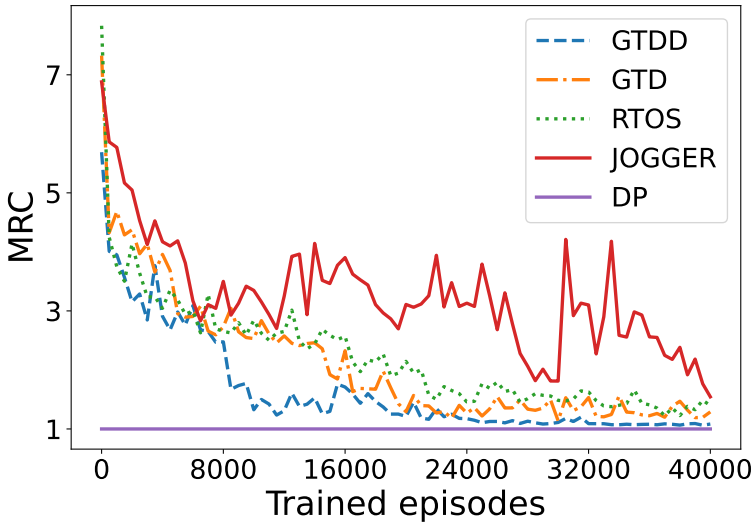


Figure 5.1: Training curve (MRC) on JOB

Figure 5.1 shows the training curve on JOB. GTDD outperforms the rest of the DRL-based methods at around 8,000 episodes and achieves performance comparable to DP at around 25,000 episodes, whereas RTOS still results in plans that are 1.6 times more costly than those produced by DP. JOGGER continues to oscillate throughout the training process, ultimately achieving the worst MRC among other baselines. Compared to the overall training curve, JOGGER demonstrates more frequent fluctuations. This may be due to the attention tree's inability to retain long-term memory, resulting in the loss of critical information.

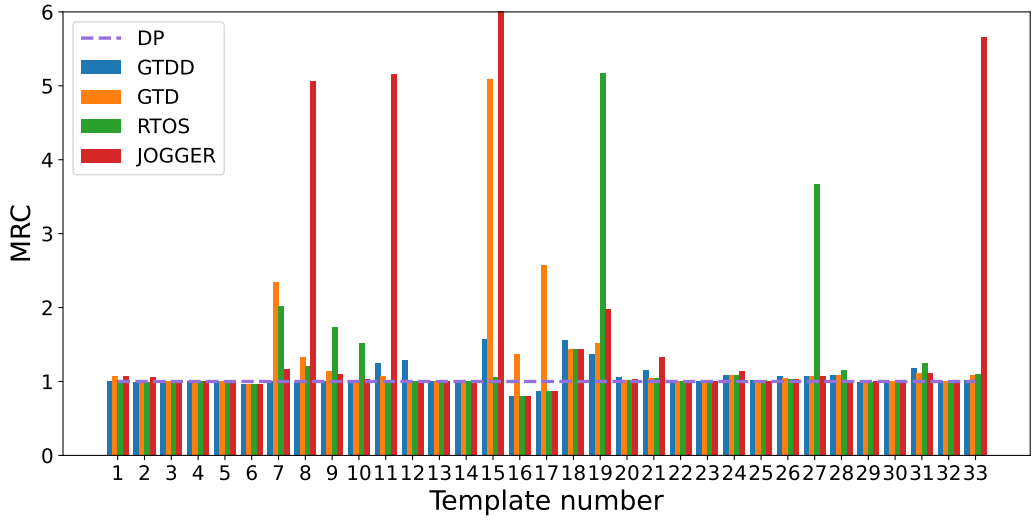


Figure 5.2: Per-query result using MRC, evaluated on the test set

We performed 11-fold cross-validation on the JOB dataset, as it contains 33 different

templates. In each fold, we included 3 templates in the test set and the remaining 30 templates in the training set. Figure 5.2 presents the test set result in 11-fold cross-validation. All methods achieve satisfactory results on most templates. However, RTOS exhibits suboptimal performance on templates #19 and #27, while JOGGER performs inadequately on templates #8, #11, #15, and #33, which are among the most complex templates. Notably, GTDD demonstrates superior performance in terms of generalization compared to other approaches, even on these challenging templates. The main purpose of using k-fold cross-validation is to help ensure that the model generalizes well to unseen data.

Algorithm	MRC on JOB	MRC on TPC-H
GTDD	**1.06313**	**1.0000**
GTD	1.13165	**1.0000**
RTOS	1.22442	**1.0000**
GTDD	1.54444	**1.0000**

Table 5.1: MRC to DP

Table 5.1 summarizes the overall performance on both benchmarks. The results indicate that GTDD outperforms other baselines and performs comparably to DP on JOB. On TPC-H, all the methods exhibit the same performance as DP. This disparity can be attributed to the complexity of the dataset: JOB includes 21 tables, with the largest query involving 17 joins, whereas TPC-H includes 8 tables, with the largest query involving 8 joins. We can conclude that join order is not the primary challenge in TPC-H.

Algorithm	optimal MRC	Cost training time	Episode of optimal
GTDD	**1.06313**	6323.59742	39,000
GTD	1.13165	6185.84266	**31,000**
RTOS	1.22442	5868.75733	38,000
JOGGER	1.54444	**5332.49978**	40,000

Table 5.2: Learning efficiency in seconds

From Table 5.2, we can conclude that GTDD observes the lowest MRC on JOB, indicating that dueling-DQN significantly improves performance compared to vanilla DQN. As expected, GTDD requires the longest training time due to the additional computation overhead associated with Dueling-DQN compared to vanilla DQN. Although GTDD achieves the best MRC at around 39,000 episodes, which is the latest among the other methods, Figure 5.1 shows that GTDD outperforms GTD, RTOS, and JOGGER after approximately 8,500 episodes and continues to do so.

5.2.2 Latency Tuning

The ultimate objective of join order selection is to find the optimal sequence of join orders that minimizes latency. Following the cost training phase, GTDD has prior knowledge of the data distribution, which enables it to generate plans with low cost. This knowledge can be leveraged to inform the tuning of the pre-trained model to generate plans with lower latency. Based on the model trained with cost, we move to the actual latency to further tune the model.

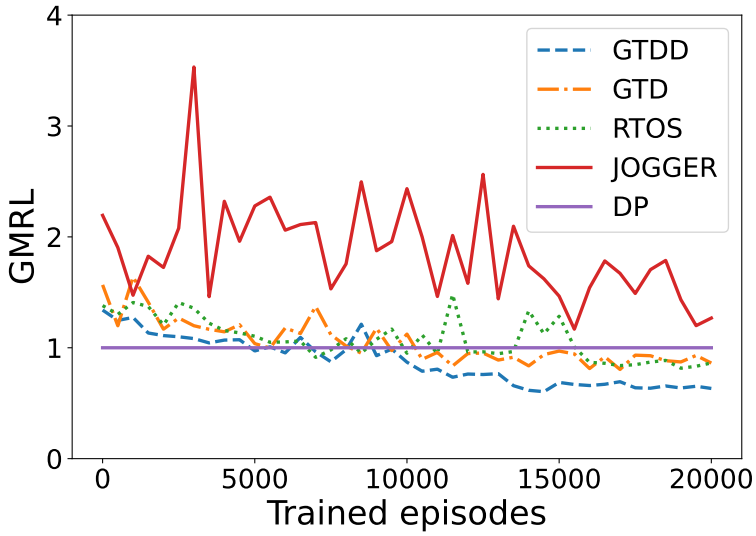


Figure 5.3: Training curve (GMRL) on JOB

The GMRL metric is used to measure the performance improvement in ratio. It should be noted that a GMRL value below 1 indicates the identification of a superior join order compared to DP on latency. Figure 5.3 illustrates the result of latency tuning, demonstrating that most DRL-based models achieve superior solutions compared to DP. In contrast, JOGGER fails to identify better solutions than DP, which is reasonable given that it does not fully leverage prior knowledge during the cost training phase. Notably, GTDD demonstrates a more stable learning process than other baselines. The latency tuning process of GTDD stabilizes after approximately 15,000 episodes, while other baselines continue to exhibit exploration. It is evident that GTDD leverages superior prior knowledge compared to other baselines, leading to a faster and more stable learning process throughout the latency tuning phase.

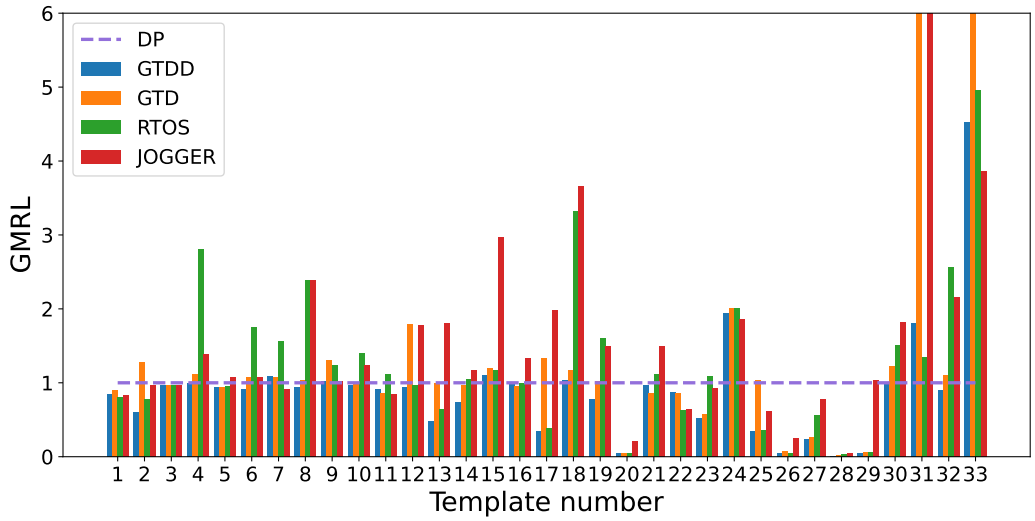


Figure 5.4: GMRL on different templates of JOB

Figure 5.4 demonstrates the GMRL for different templates in JOB. In most cases, GTDD finds better or comparable plans to DP and outperforms other DRL-based models. However, for template #33, all the DRL-based methods fail to find an execution plan that is optimal and performs drastically poorly. This is because template #33 is one of the most complex templates, containing 19 join predicates, resulting in a large search space. In templates #30 and #32, GTDD finds the join orders close to DP, while other baselines generate worse plans than DP. Although GTDD does not find the best join order for all templates, it still demonstrates superior performance compared to both the other DRL-based optimizers and the native PostgreSQL optimizer.

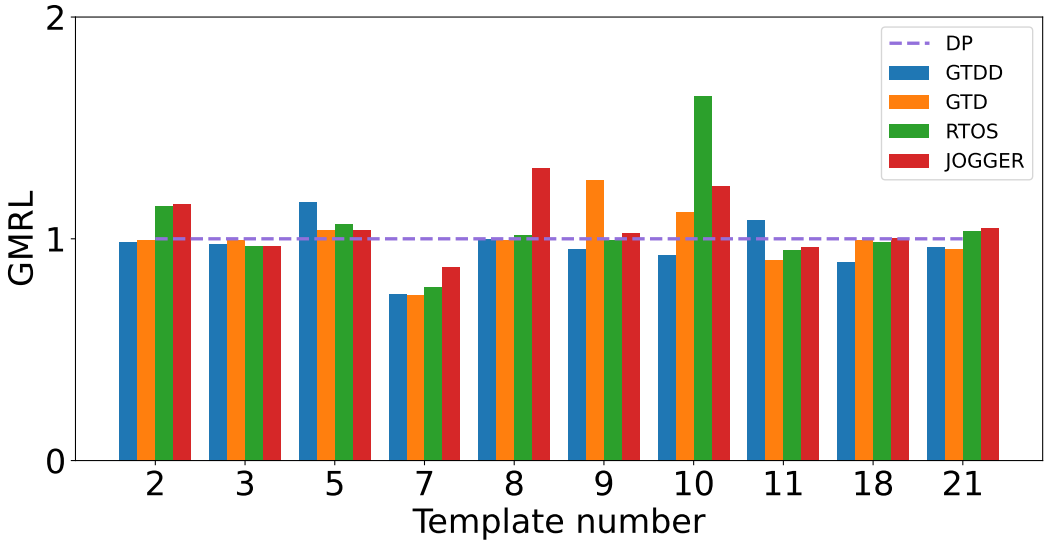


Figure 5.5: GMRL on different templates of JOB

Figure 5.5 demonstrates the performance on TPC-H. We conclude that there's no big difference on this dataset due to a much smaller search spaces. The queries are very simple compared to the queries in JOB, and we can see that GTDD achieves the best results compared to GTD and RTOS.

Algorithm	GMRL on JOB	GMRL on TPC-H
GTDD	**0.60381**	**0.96399**
GTD	0.80425	0.99225
RTOS	0.83313	1.04111
JOGGER	1.16758	1.05568

Table 5.3: GMRL to DP

Table 5.3 shows that GTDD achieves the best GMRL in the latency tuning phase for both JOB and TPC-H. Although GTD has a performance similar to RTOS, Figure 5.3 indicates that the GTD exhibits a more stable training process compared to RTOS, at around 12,000 episodes, GTD becomes relatively stable while RTOS continues to oscillate. GTDD achieves 0.60381 GMRL on JOB, representing a 20.044% improvement compared to GTD, a 22.932% improvement compared to RTOS, and a 56.377% improvement compared to JOGGER, demonstrating the effectiveness of adapting dueling-DQN over DQN.

Chapter 6

Conclusion and Future work

6.1 Conclusion

The topic of query optimization has become an important area of research in database and provides many research directions, especially in **Cardinality Estimation**, **Cost Model** and **Join Order Selection**. With decades of learning, many remarkable contributions have been made, demonstrating significant improvements in the field. However, despite these advances, query optimization remains an NP-hard problem. In this study, we conducted a comprehensive examination of the existing research literature on different components of query optimization, discussing a range of methods, including traditional approaches and ML approaches. We have also discussed the strengths and limitations of these methods. Our work is primarily focused on the application of RL to the problem of join order selection. In this approach, the existing optimizer's feedback serves as a guide for the agent to learn. Our experiments have demonstrated that DRL is capable of learning the join order and even discovering more optimal join plans. However, in certain cases, the traditional optimizer has been observed to outperform the DRL models. Moreover, the selection of join orders is not only a question on RL, but also on representation learning (e.g. column representation, table representation, state representation). The informative representations could significantly improve the training time and enhance the performance (e.g. MRC). Additional information could facilitate the model's comprehension of the query.

In the process of developing the DRL model, we have encountered difficulties in capturing more representative information and determining the most appropriate approach. The first decision we have made is to remove one-hot encoding and use graph neural network to capture more representative information. We have attempted different types of pooling layer and GCN and have determined that the most effective approach is to use global max pooling and transformer convolutional layer. The second challenge we have encountered is that the agent selects an invalid action, which picks two tables that are not even participants in the query to join. To address this, we initially attempted to use a 'fake join' for each valid action and score them which is computationally expensive. Consequently, we have changed to a more elegant way, using action mask to mask out invalid actions.

Finally, series of experiments was conducted to optimize the selection of join orders. The experimental results demonstrated that DRL could enhance the performance of the join order selection, and that the use of informative representations could significantly improve the model's performance.
Throughout this research, we have observed that existing benchmarks for evaluating join order selection in query optimization exhibit notable limitations. The most commonly used benchmarks, JOB and TPC-H, are based on static query workloads and fixed data distributions, which do not fully reflect the dynamic, real-world environments of modern database system. Additionally, they often lack diversity in query patterns, failing to represent the wide range of query complexities and join conditions encountered in practical applications. These gaps underscore the need for new benchmarks that better capture the demands and diversity of modern database workloads to enable a more thorough and meaningful evaluation of advanced optimization techniques.

6.2 Future Work

There are many interesting future works to be discussed. Firstly, RobOpt [18] and Fauce [26] have proposed an interesting concept on robustness that considers both data uncertainty and model uncertainty. It would be beneficial to extend the concept to RL, whereby actions are not only selected based on the highest q-value, but also with the consideration of uncertainty. It also gains some explanation on action selections and the queries with high uncertainties could be trained with incremental learning, or capture the pattern to generate more similar queries to add to the training set. More importantly, this could prevent the model from selecting the action that leads to a disastrous plan and making the worst-case scenario acceptable. If the worst-case scenario has been improved, it could be adopted in commercial databases since the overall performance would be satisfactory.
Secondly, all the join order selection solutions train different models for different databases and even different datasets. However, the ultimate goal is be the same, to find the optimal join order for the query. One potential line for future research is the use of transfer learning to train a model in a database with a dataset and then transfer it to other databases or datasets. This could enhance the practicality of the model and potentially make it suitable for industrial applications. Furthermore, if the model is capable of generalization and can adapt to changes in the schema without requiring retraining of the entire model, it could be a valuable asset in an industrial perspective. Moreover, the model learns variance queries crossing many databases and it could learns expert knowledge by discovering the underlying pre-defined rules across various DBMS, which could potentially enhance generalization and robustness.
Thirdly, another problem in join order selection problem is that the model does not receive a reward until the end of the episode. One potential solution is to utilize curiosity driven rewards, enabling the model to not only learn from the ultimate reward from the environment, but also from the intrinsic curiosity reward. This approach brings two benefits: it employs a model to mitigate the exploration-exploitation dilemma and could eliminates the naive $\epsilon-$greedy strategy. In addition, the model could observe some intermediate re-

wards during the training process, rather than solely focusing on the ultimate reward. This approach could encourage the agent to select more reliable actions, as the intermediate rewards could provide a motivating factor. An additional potential solution is to pre-train a basic cost estimator that utilizes a join between two tables to estimate the intermediate cost and provide intermediate feedback as a reward to the agent.

Embedding DRL-based join order selection techniques into existing database engines represents a promising yet challenging direction for future work. Unlike traditional optimizers, a DRL model lacks prior knowledge of incoming queries, which limits the feasibility of curriculum learning approaches for efficient adaptation to query workloads. Additionally, in industrial settings, databases undergo frequent updates, and DRL-based models are not well-suited for environments with rapidly changing data structures. DRL models also typically require extended decision-making times and substantial training data, which conflicts with the low-latency, real-time demands of modern database systems. Combined with the need for reliability and explainability in optimization decisions, these challenges highlight the complexity of directly embedding DRL-based methods into current database infrastructures. Embedding a DRL-based model into existing databases could bring significant improvements to join order optimization by enabling dynamic learning and adaptation to workload patterns, leading to ongoing query performance optimization. Unlike traditional optimizers, which often struggle with complex, multi-join queries in large-scale analytics, a DRL-based optimizer could continuously refine its strategy to handle diverse and evolving query patterns. This adaptability would enable DBMSs to maintain consistently high performance, reducing both latency and resource consumption in analytics-intensive applications. Additionally, the optimizer would be capable of identifying its own inefficiencies, allowing it to self-correct and improve over time.

APPENDICES

Appendix A

Bao's hint sets

These are 48 hint sets Bao has selected[29]:

- hashjoin, indexonlyscan
- hashjoin, indexonlyscan, indexscan
- hashjoin, indexonlyscan, indexscan, mergejoin
- hashjoin, indexonlyscan, indexscan, mergejoin, nestloop
- hashjoin, indexonlyscan, indexscan, mergejoin, seqscan
- hashjoin, indexonlyscan, indexscan, nestloop
- hashjoin, indexonlyscan, indexscan, nestloop, seqscan
- hashjoin, indexonlyscan, indexscan, seqscan
- hashjoin, indexonlyscan, mergejoin
- hashjoin, indexonlyscan, mergejoin, nestloop
- hashjoin, indexonlyscan, mergejoin, nestloop, seqscan
- hashjoin, indexonlyscan, mergejoin, seqscan
- hashjoin, indexonlyscan, nestloop
- hashjoin, indexonlyscan, nestloop, seqscan
- hashjoin, indexonlyscan, seqscan
- hashjoin, indexscan
- hashjoin, indexscan, mergejoin
- hashjoin, indexscan, mergejoin, nestloop
- hashjoin, indexscan, mergejoin, nestloop, seqscan
- hashjoin, indexscan, mergejoin, seqscan
- hashjoin, indexscan, nestloop
- hashjoin, indexscan, nestloop, seqscan
- hashjoin, indexscan, seqscan
- hashjoin, mergejoin, nestloop, seqscan
- hashjoin, mergejoin, seqscan
- hashjoin, nestloop, seqscan
- hashjoin, seqscan
- indexonlyscan, indexscan, mergejoin
- indexonlyscan, indexscan, mergejoin, nestloop

- indexonlyscan, indexscan, mergejoin, nestloop, seqscan
- indexonlyscan, indexscan, mergejoin, seqscan
- indexonlyscan, indexscan, nestloop
- indexonlyscan, indexscan, nestloop, seqscan
- indexonlyscan, mergejoin
- indexonlyscan, mergejoin, nestloop
- indexonlyscan, mergejoin, nestloop, seqscan
- indexonlyscan, mergejoin, seqscan
- indexonlyscan, nestloop
- indexonlyscan, nestloop, seqscan
- indexscan, mergejoin
- indexscan, mergejoin, nestloop
- indexscan, mergejoin, nestloop, seqscan
- indexscan, mergejoin, seqscan
- indexscan, nestloop
- indexscan, nestloop, seqscan
- mergejoin, nestloop, seqscan
- mergejoin, seqscan
- nestloop, seqscan

References

[1] Bernhard Radke Viktor Leis Peter Boncz Alfons Kemper Andreas Kipf, Thomas Kipf. Learned cardinalities: Estimating correlated joins with deep learning. *CIDR '19*, 2019.

[2] TF-Agents Authors. Introduction to RL and Deep Q Networks. https://www.tensorflow.org/agents/tutorials/0_intro_rl, 2023.

[3] S. Sudarshan Avi Silberschatz, Henry F. Korth. *Database System Concepts*. McGraw Hill, 2019.

[4] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 1957.

[5] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[6] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48. Association for Computing Machinery, 2009.

[7] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. Efficient join order selection learning with graph-based representation. *KDD '22*, page 97–107, 2022.

[8] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. *CoRR*, 2020.

[9] The PostgreSQL Global Development Group. PostgreSQL 16.2 Documentation Chapter 76.1 Row Estimation Examples. https://www.postgresql.org/docs/current/row-estimation-examples.html, 2024.

[10] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *KDD '16*, page 855–864, 2016.

[11] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, LiangWei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Cardinality estimation in dbms: a comprehensive benchmark evaluation. *VLDB*, 15(4):752–765, 2021.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[13] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, 2022.

[14] IBM. Ibm db2. https://www.ibm.com/products/db2, 2024.

[15] IBM. What is a neural network? https://www.ibm.com/topics/neural-networks, 2024.

[16] Ixnay. Recurrent neural network. https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg, 2017.

[17] A.K. Jain, Jianchang Mao, and K.M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31–44, 1996.

[18] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. Roq: Robust query optimization based on a risk-aware learned cost model, 2024.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[20] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, 2016.

[21] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *ArXiv*, 2018.

[22] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6:86 – 101, 2021.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[25] Xiaohong Li. Expression recognition of classroom children's game video based on improved convolutional neural network. *Scientific Programming*, 2022:1–10, 04 2022.

[26] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.*, 14(11):1950–1963, 2021.

[27] G. Lohman. Is query optimization a solved problem? http://wp.sigmod.org/?p=1075., 2014.

[28] Tanu Malik, Randal C. Burns, and N. Chawla. A black-box approach to query cardinality estimation. In *Conference on Innovative Data Systems Research*, 2007.

[29] Ryan Marcus. Bao online appendix. https://rmarcus.info/appendix.html, 2021.

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *SIGMOD '21*, page 1275–1288, 2021.

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: a learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[32] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. *aiDM'18*, 2018.

[33] MariaDB. InnoDB Persistent Statistics. https://mariadb.com/kb/en/innodb-persistent-statistics/, 2024.

[34] Clara Meister, Tim Vieira, and Ryan Cotterell. Best-First Beam Search. *Transactions of the Association for Computational Linguistics*, 8:795–809, 2020.

[35] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *ICLR*, 2013.

[36] V. Mnih, K. Kavukcuoglu, Silver, and D. et al. Human-level control through deep reinforcement learning. *Nature*, 519(529–533), 2015.

[37] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. *CoRR*, page 1287–1293, 2016.

[38] Oracle. MySQL 8.0 Reference Manual Chapter 17.8.10.2. https://dev.mysql.com/doc/refman/8.0/en/innodb-statistics-estimation.html, 2020.

[39] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, 2015.

[40] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *SIGMOD Rec.*, 29(4):64–71, dec 2000.

[41] Richard Bellman. The Theory of Dynamic Programming. *American Mathematical Society*, 1954.

[42] Jason Roth, Randolph West, and Mark et al. Ghanayem. Cardinality Estimation (SQL Server). https://learn.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver16, 2024.

[43] savyakhosla. CNN — Introduction to Pooling Layer. https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/, 2024.

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

[45] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR*, 2019.

[46] Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty. Reinforcement learning algorithms: A brief survey. *Expert Syst. Appl.*, 231(C), 2023.

[47] Yunsheng Shi, Zhengjie Huang, Wenjin Wang, Hui Zhong, Shikun Feng, and Yu Sun. Masked label prediction: Unified massage passing model for semi-supervised classification. *CoRR*, abs/2009.03509, 2020.

[48] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *ArXiv*, 2015.

[49] Lixia Tian, Yuansheng Huang, Shuang Liu, Shize Sun, Jiajia Deng, and Hengfeng Zhao. Application of photovoltaic power generation in rail transit power supply system under the background of energy low carbon transformation. *Alexandria Engineering Journal*, 60(6):5167–5174, 2021.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, page 6000–6010, 2017.

[51] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *ICML'16*, page 1995–2003, 2016.

[52] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. *Cambridge*, 1989.

[53] Zhengtong Yan, Valter Uotila, and Jiaheng Lu. Join order selection with deep reinforcement learning: Fundamentals, techniques, and challenges. *Proc. VLDB Endow.*, 16(12):3882–3885, 2023.

[54] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 931–944, 2022.

[55] Jeremy Zhang. Reinforcement Learning — Implement Grid World. https://towardsdatascience.com/reinforcement-learning-implement-grid-world-from-scratch-c5963765ebff, 2019.