Deep Cardinality Estimation for Join Order Optimization

Master Thesis

Author:
Laxmi Balami

Examiner and Supervisor:
Prof. Dr. rer. nat. habil. Gunter Saake

Supervisor:
MSc. Gabriel Campero Durand

Magdeburg, 29.05.2020
Balami, Laxmi

*Deep Cardinality Estimation for Join Order Optimization*

Abstract

The remarkable breakthroughs in the area of AI and machine learning in the last few years have made significant contributions to various domains such as e-commerce, banking, security, government, to name a few. As such, they have also captured the interest of the database community with research and publications proposing machine learning based solutions for traditional database tasks. The extensively studied, yet unsolved task of highly reliable cardinality estimation for Join Order Optimization in relational databases, is justifiably one of the tasks selected, with an ever growing number of researchers and academia making progression towards the shared vision of learning-based estimators.

However, as of yet, there has not been any study on the impact of the hyper-parameters in the overall performance of the proposed state-of-the-art model, based on a Multi-set Convolutional Network (MSCN). Therefore, in this research, we question, given the model architecture what would be the “right” combinations of hyper parameters to implement and seek to validate why those combinations are the “right” ones to use. Furthermore, we examine if the use of word-embeddings for the query predicates with string values has an impact on training the model and how much of a difference does it make when using a pre-trained word embedding model versus the model trained on the database in use.

Our early experiments show the impact of various configurations of hyper-parameters, which is more noticeable after introducing the embeddings to the model. We show that the embeddings from the model we train on our database has a slight improvement over the default embeddings of the pre-trained word embedding model. We also evaluate the predictions made by the trained model against the estimations from PostgreSQL and baseline regressors, to make a comparison, finding that the MSCN model is able to outperform both alternative approaches, with a noticeable edge over the query optimizer of PostgreSQL.

Our experiments also show that the set of hyper-parameters tuned for one dataset (JOB) are similar one using another dataset (a synthetic one), and that we are able to achieve good results on the new dataset, with some generalization when testing on JOB queries, but still room for improvement.

Given the limited time and scope of this research, we scope our experiments with the hyper parameters to a fixed set of selections only, which leaves room for improvement and continuation of this research work in the future in this direction.
Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Gabriel Campero Durand, for his expert guidance, persistent support and patience, right from the inception of the thesis topic to the completion. His unwavering enthusiasm for this area of research motivated me immensely during the arduous phases of this thesis.

I would also like to extend my deep appreciation to Prof. Dr. rer. nat. habil. Gunter Saake for his profound support, approval and considerateness towards us and the research work. I also take this opportunity to thank Dr. Felicitas Hentschke and Sebastian Marggraff from Humboldt University of Berlin for their support, kindness and encouragement towards me and my studies.

My sincere thanks also goes to Mrs. Warnecke-Timme and the examination committee for extending the thesis timeline in regards to the Corona pandemic and the difficulties arising therefrom.

Last but not the least, I acknowledge my family and my friend Ramona Ottow, for their enduring moral and emotional support, especially during the time of writing this thesis.
Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

______________________________  ______________________________
Signature                        Place, Date
Contents

List of Figures 3

List of Tables 5

1 Introduction 7
   1.1 Motivation ................................. 7
   1.2 Main Contributions .......................... 9
   1.3 Thesis Structure ............................. 9

2 Background 11
   2.1 Structure of Literature Review ................ 11
   2.2 Join Order Optimization (JOO) ................ 11
   2.3 Query Plan Space Enumeration .................. 13
   2.4 Cardinality Estimation ........................ 18
   2.5 Deep Sets Learning ............................ 23
   2.6 Word Embeddings .............................. 24
   2.7 Summary .................................. 25

3 Prototype Design 27
   3.1 Research Questions ........................... 27
   3.2 Model Architecture ............................ 27
   3.3 Word representation models ..................... 29
   3.4 Database and benchmark workload ............... 30
   3.5 Baseline Regressors ........................... 31
   3.6 Summary .................................. 31

4 Experimental Setup 33
   4.1 Dataset Generation and Pre-processing .......... 33
      4.1.1 Random Query Generation .................. 34
      4.1.2 Query Featurization ....................... 34
      4.1.3 Word Embedding ............................ 38
   4.2 Model Hyper-parameters ........................ 39
   4.3 Experiment and Evaluation Environment ........... 40
   4.4 Summary .................................. 41

5 Evaluation and Results 43
   5.1 Research Questions ........................... 43
   5.2 Evaluation Measures for Research Question I .... 43
   5.3 Evaluation Measures for Research Question II ... 58
   5.4 Evaluation Measures for Research Question III ... 70
   5.5 Summary .................................. 71
## List of Figures

2.1 Two query plans with same relations but different join orderings. (Adapted from [Dit16]). .................................................. 12
2.2 Taxonomy of machine learning and non machine learning based approaches for query plan space optimization. .......................... 14
2.3 A 2 Dimensional Error-prone Selectivity Space (ESS) for a query in TPC-H. (Image sourced from [KHKP18]) .................................. 15
2.4 ReJOIN Framework. [MP18a]. Image sourced from [MP18a] ............. 16
2.5 Neo Model. Image sourced from [MN'M+19] .................................. 17
2.6 Taxonomy of machine learning and non machine learning based approaches for cardinality estimation. ................................. 19
2.7 State Representation [OBGK18]). .............................................. 21
2.8 MSCN Architecture [KKR+18]). .................................................. 22
2.9 Deep Sketches Framework. (Image sourced from [KVM+19]). ............. 23
2.10 Deep Sets Architecture (Image sourced from [WFE+19]). .................. 24

3.1 Set input based neural network architecture that uses mean aggregation function and fully connected hidden layers for transforming the input and later to process the order invariant representation of the transformed inputs. 28
3.2 Heat-map of all the join co-occurrences in the JOB workload .............. 31
4.1 IMDB database schema with foreign key relationship used to determine join pairs among tables during random query generation. The dotted line represents the parent-child relationship. ....................... 35
4.2 One hot encoding for columns in our model .................................. 35
4.3 One hot encoding for operators in our model .................................. 36
4.4 One hot encoding for tables in our model ...................................... 36
4.5 Example of encoding for tables in our model for Query 13a.sql from JOB 37
4.6 EXPLAIN ANALYZE command and the resulting query plan from Post- greSQL optimizer. .............................................. 38
5.1 Training loss per iteration for the four models with four different activation function but the same weights initializer: Glorot Uniform. ....... 45
5.2 Scaled plot for the training loss with weights initializer: Glorot Uniform. 45
5.3 Training loss per iteration for the four models with four different activation function but the same weights initializer: Glorot Normal. ......... 46
5.4 Training loss per iteration for the four models with four different activation function but the same weights initializer: HE Normal. .......... 47
5.5 Training loss per iteration for the configuration with HE Uniform. ........ 47
5.6 Training loss per iteration for the configuration with Adam at different learning rates. ........................................ 50
5.7 Training loss per iteration for the configuration with \textit{RMSProp} at different learning rates. .......................................................... 51
5.8 Training loss per iteration for the configuration with \textit{SGD with Momentum} with different learning rates. ....................................................... 51
5.9 Training loss per iteration for the configuration with \textit{Nesterov Accelerated Gradient} with different learning rates. ............................... 52
5.10 Training loss per iteration for models configured with 0.8 dropout rate at different layers. ................................................................. 55
5.11 Training loss per iteration for different configuration of individual input sets (networks) with varying units in the first hidden layer of each network. 58
5.12 Loss curve for dataset with embeddings from default pre-trained fasttext model and model trained on IMDB database ................................. 59
5.13 Annotated Heat-map of weights distribution at first hidden layer of the network that feeds into Tables Sets as input. .............................. 60
5.14 Prediction on test JOB dataset with embeddings from default pre-trained fasttext model ................................................................. 60
5.15 Prediction on test JOB dataset with embeddings from model trained on IMDB database ................................................................. 60
5.16 Model Predictions on Query Template 21 ........................................ 61
5.17 Model Predictions on Query Template 25 ........................................ 62
5.18 Model Predictions on Query Template 26 ........................................ 62
5.19 Model Predictions on Query Template 33 ........................................ 62
5.20 Model Predictions on Query Template 23 ........................................ 63
5.21 Model Predictions on Query Template 28 ........................................ 63
5.22 The SHapley explanation plot for 24 input features ............................ 64
5.23 SHAP Summary Plot for instance number 1 ..................................... 65
5.24 SHAP Feature Importance Plot for instance number 1 ........................ 66
5.25 Training losses from models configured with varying hyper-parameters on the dataset of randomly generated queries. .......................... 67

6.1 Model Architecture. Image sourced from [SL19] .................................. 73
6.2 Representation Layer. Image sourced from [SL19] ............................... 74
6.3 A prefix tree for a query predicate \textit{LIKE 'Dino%'}. Image sourced from [SL19] 74
List of Tables

5.1 Hyper-parameter Tuning: Four weights initializers with four activation functions each at 0.001 as learning rate with Adam as the optimizer.

5.2 Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with Glorot Uniform weights initializer.

5.3 Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with Glorot Normal weights initializer.

5.4 Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with He Normal weights initializer.

5.5 Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with He Uniform weights initializer.

5.6 Hyper-parameter Tuning: Best MSE and MAE on test dataset for the two weights initializers with Normal and Uniform distribution each with its associated activation function.

5.7 Hyper-parameter Tuning: Configuration of four optimizers with three different learning rate each, with HE weights initializer and ReLU as the activation function.

5.8 Hyper-parameter Tuning: MSE and MAE on test dataset with Adam as the optimizer with different learning rates.

5.9 Hyper-parameter Tuning: MSE and MAE on test dataset with RMSProp as optimizer at different learning rates.

5.10 Hyper-parameter Tuning: MSE and MAE on test dataset with SGD with Momentum as optimizer at different learning rates.

5.11 Hyper-parameter Tuning: MSE and MAE on test dataset with Nesterov Accelerated Gradient as optimizer with different learning rates.

5.12 Hyper-parameter Tuning: Best MSE and MAE on test dataset for each optimizer with corresponding learning rate.

5.13 Hyper-parameter Tuning: MSE and MAE on test dataset at different dropout rates applied on all layers in the network.

5.14 Hyper-parameter Tuning: MSE and MAE on test dataset at 0.8 dropout rate applied on selected layers of the network.

5.15 Hyper-parameter Tuning: Changing network width at first two hidden layers that transforms the sets inputs for further computation.

5.16 Hyper-parameter Tuning: Configuration of individual input sets (networks) with varying units in the first hidden layer of each network.

5.17 Word Embeddings: MSE and MAE from models with embeddings from pre-trained fastText model and from the model trained on IMDB database.

5.18 Hyper-parameter Tuning for the dataset with random generated queries.

5.19 MSE and MAE losses on the test dataset from JOB workload and MSE and MAE on the same dataset test split.
5.20 True cardinalities of JOB test dataset, predictions from trained model and estimations from PostgreSQL

5.21 Mean squared error (MSE) and Root Mean squared error (RMSE) for baseline Regressors sorted on regressor with least MSE to highest.

5.22 MSE from the two models we trained explained in 5.2 and 5.3
1 Introduction

We start our thesis with an overview of Join Order Optimization as an integral part of query optimization in relational database management systems. In Section 1.1, we explain the importance of cardinality estimation in Join Order Optimization and the state of progress in the recent five years which serves as the motivation for this research work. We outline the main contributions of our research work in Section 1.2 and conclude this chapter with Section 1.3, by providing the structure of the subsequent sections of this thesis.

1.1 Motivation

Databases form an integral part of modern computer systems. As computer systems have changed in recent years, with the rise of big data, data-centric applications and business analytics, there is an implied rise of complexities in database schemas, workloads and queries. Today, various commercial tools are deployed to support queries of great complexity in order to assist efficient analysis and reporting. The complexity of these workloads and queries thus now poses new challenges to currently existing query optimizers, which could fail to detect complex patterns often found in real world data-sets (like join-crossing correlations, skew and non-uniform data distributions), ultimately leading to bad query plans. Very often, database administrators, SQL programmers and database experts intervene and manually debug the optimizer’s 'optimal’ query plans, for example overriding join orders, injecting true cardinalities, making special parameter configurations or table denormalizations, and so on, to improve everyday performance. But given the increasing complexity of workload and queries, these tasks might become difficult and increasingly impractical (i.e., demanding efforts better spent on other tasks). This has created the need for query optimizers that can produce more accurate estimates, even in the presence of complex queries, and hence generate optimal query plans that reduce time, costs and resources in query processing.

Although the optimality of a query plan is determined by the cost model, only accurate cardinality estimates make the cost model a reliable predictor of query run-time. This highlights the need to have accurate cardinality estimates before perfect cost models. Moreover, since cardinality estimation errors can propagate exponentially with the increase in the number of joins in the query, they have been shown experimentally to significantly dwarf the errors introduced by the cost model[LGM+15]. This further emphasizes the importance of accurate cardinality estimation over cost models in determining optimal query plans.

Regardless of the significance of cardinality estimation in query optimization, virtually all current generation commercial or industrial strength systems rely on statistics like per-attribute histograms and/or sampling for the estimation. These optimizers based on
the traditional System R [SAC79] architecture make assumptions about uniformity, independence and containment and therefore fail to consider correlations among attributes, which leads to query plans that are far beyond optimal. Additionally, since synthetic query workloads do not represent the complexity of the real world dataset, research and experiments carried out using these benchmarks most often do not yield the same results on real world datasets.

Although cardinality estimation has been an active research subject for almost five decades now, with the recent surge in machine learning, many novel concepts coming out of academia and research likewise, have garnered more attention and interest towards this subject matter. In the past five years itself, various significant works have been carried out for Join Order Optimization motivated by the idea of building a learning-based query optimizer that is, if not better, at least as good as the query optimizers currently in use.

Preliminary works like the Deep reinforcement Learning based framework ReJOIN [MP18a], introduces the concept of providing positive/negative feedback on the query optimizer’s selection of query plans in order for the optimizer to learn from experiences how to distinguish good query plans from bad ones, eventually leading to optimal plans. Similarly, a deep neural network based model Neo [MNM19] was proposed, to bootstrap from an existing query optimizer to generate optimal query execution plans. While considering cardinality estimation, Kipf et al. [KVM19] introduced Deep Sketches, which provides a graphical user interface for users to select training parameters such as the number of epochs, the number of training queries and the number of materialized base tables, using a Multi-Set Convolutional Network (MSCN) based model from their earlier work [KKR18] as the underlying model. Ortiz et al. [OBGK18] also introduced a Deep Reinforcement Learning (DRL) based model that incrementally learns to generate a concise representation of sub queries of a SQL query, based on the observed cardinality estimates.

However, in spite of the sheer amount of ongoing research works and progress, there exists a significant lack of available studies on the impact of problem framing, model configuration and the design of the training process. Often if not always, there arise questions on the reproducibility of the claimed results. Factors like input featurization, choice of learning algorithm, datasets/workload selection and hyper-parameter configuration, all affect the results, consequently making it difficult to evaluate or reproduce the proposed solutions. In addition to this, the state-of-the-art in cardinality estimation report results using only queries on numeric predicates, completely ignoring the string predicates. These studies particularly overlook the importance of string predicates, an arguably frequent occurrence in everyday queries, thus questioning the generalization power of such works.

Motivated by these factors, in this thesis work, we conducted a re-implementation of a Deep Sets [ZKR17] based neural network model similar to MSCN [KKR18] but with full the JOB workload including queries on string predicates as well. In order to featurize the string predicates we use word embeddings, which employ a neural network model to map the input strings to a high-dimensional dense vector of floating point numbers. Through the word embedding method, the strings are mapped to a representation that can be fed to a learned model. Simultaneously the resulting representation preserves the semantic similarity of strings (i.e., strings that are similar in their semantics are mapped to
representations with a high cosine similarity between them, while semantically dissimilar strings lead to representations that are dissimilar).

We use pre-trained word embedding models to get word representations for the predicates on string values, one set with default embeddings and another with embeddings from the model trained on the IMDB database. We study the impact of different hyper parameter configurations to train the model, and the outcome of the trained model. Lastly, based on our experiments and evaluations, we propose future works in this area of research work (e.g. testing alternative text featurizations).

1.2 Main Contributions

1. We implement a Deep Sets model for cardinality estimation using full Join Order Benchmark queries including ones with string predicates as well.

2. We experiment with different configuration of hyper parameters to train the model and study their impact on the performance of the model.

3. We featurize query predicates on string values using vector representations generated by pre-trained word embedding models with default embeddings and embeddings generated from the model trained on the workload database. We compare the impact of these on the overall performance of the model.

4. We compare the trained Deep Sets model against baseline regressors and see how close or far off the models are.

1.3 Thesis Structure

The rest of the thesis is structured as follows:

- In Chapter 2, we start with the background of Join Order Optimization in query optimization and its two areas of optimization: cardinality estimation and query plan space optimization. We provide the state-of-the-art in both non-machine learning and machine learning based solutions for cardinality estimation as well as query plan space optimization. Similarly, we explain the concept of Deep Sets learning that our model is based on as well as a brief background on word embeddings.

- Chapter 3 chapter presents the architecture of our model as well as dataset and benchmark workload we use for the research. We also list the pre-trained word embedding model and baseline regressors we use in our evaluations.

- Chapter 4 provides details on query featurization and word embedding process as well as the details of the experiment environment and its eco-system of softwares and libraries.

- In Chapter 5 we state our research questions and for each research question, we state our hypothesis, experiments conducted and results achieved. Towards the end, we analyse and compare the obtained results against the estimations from PostgreSQL’s optimizer.
Finally, we conclude by summarizing our results and providing possible directions for future extensions to this research in Chapter 7, after a brief explanation on a recently published work closely related to ours in Chapter 6.
2 Background

In this chapter, we provide theoretical background relevant to the model architecture, experimental setup and evaluation results presented in the later chapters. The rest of the section is organized as follows:

- **Section 2.2** provides an overview of Join Order Optimization for query optimization in Relational Database Management Systems.

- **Section 2.3** explains query plan space optimization for Join Order Optimization and presents both machine learning and non machine learning based state-of-the-art in query plan space optimization in the last five years.

- In **Section 2.4**, we present the overview of cardinality estimation for Join Order Optimization and the state-of-the-art in cardinality estimation in the last five years.

- Lastly, in **Section 2.5**, we provide the fundamentals of Deep Sets learning and the relevant concept of word embeddings.

2.1 Structure of Literature Review

The background for this research work comes from our survey, carried out as part of a student conference at our University\(^1\). In this survey we covered cardinality estimation and query plan space optimization, reviewing research works on both machine learning as well as non machine learning based solutions in the past five years following the works of Victor Leis\(^2\).

2.2 Join Order Optimization (JOO)

A large body of extensive research work has been dedicated towards query optimization for more than five decades now, yet the existence of complex join queries involving a large number of relations and join-crossing correlations makes finding an optimal query plan still an open challenge and establishes Join Order Optimization (JOO) as one of the most studied query optimization tasks in Relational Database Management Systems.

---

\(^1\)Student Conference 2019: Towards Improving the State Of The Art in Join Order Optimization with Machine Learning

\(^2\)https://db.in.tum.de/people/sites/leis/
Join Order Optimization plays a fundamental role in cost based query optimization which is an integral part of database systems. A query optimizer goes through several optimization phases in order to determine the mapping between a logical query plan and its optimal query execution plan. Here, the optimality of the query plan is determined by the computational complexity as well as input/output costs, CPU time, memory and communication costs. The declarative nature of SQL simply implying what to execute and not how to execute, makes a query optimizer to be responsible for finding an optimal query plan for the given query. Since joins have commutative and associative properties, the sequence of join operations are logically equivalent and thus, do not affect the final output of the query but the join ordering itself significantly increases cost estimation for execution of the query plan.

A traditional cost-based optimizer is based on the classic System-R architecture consisting of three main components: Cardinality Estimation, Cost Model and Query Plan Space Enumeration. The optimality of a query plan is calculated by cost models that provide a simplified description of the main components of a system, giving an estimate for the actual cost of a query on such system where costs can be perceived as a metric to estimate query planning and execution time, as well as resource utilization.

The cost-based approach to query optimization starts with some join enumeration strategies (usually dynamic programming, but not limited to it), that enumerate the subsets of valid join orders in a query plan space or search space. Most commonly, statistical information like per-attribute histograms, occasionally multi-attribute histograms and sometimes base-table samples are used for cardinality estimation of the plan alternatives in the query plan space. These estimations serve as a basis to evaluate the cost model over each plan, which allows the optimizer to make the selection of an optimal query plan to be finally executed.

Let us consider a simple example with two query plans, namely Plan 1 and Plan 2, as illustrated in Figure 2.1; involving three relations A, B and C, each with 1000 tuples. In query plan 1, the first join is between the relations B and C, which forms a cross-join due to the absence of any join predicates, resulting in 100,000 tuples as an intermediate result size which is the input for the final join with A consisting 1000 tuples of its own. The top level join now has to process 100,000 + 1000 tuples. While in Plan 2, the relations A and C are joined first, resulting in only 100 tuples because of the existing join predicates between relation A and C in the query. Now, for the top level join with relation B, the
number of tuples to be processed would be $100 + 1000$ tuples of B. The huge difference in the number of tuples these two query plans have to process makes Plan 2 optimal in comparison to Plan 1. In practice though, the size of the intermediate results or the number of rows that has to be processed are only estimates rather than actual figures, and depending upon these estimates, the query optimizer selects an optimal join ordering for a given query.

The classical System-R optimizer uses dynamic programming for join enumeration and restricts the search space to only left-deep join trees. PostgreSQL’s\(^3\) optimizer, on the other hand, deploys a greedy strategy that searches for low cost pairs of relations until a tree is formed. In case of large and complex join queries, modern optimizers apply various heuristics to prune the search space such as greedy search, randomized or genetic algorithms. Pruning the search space and considering only some plans does increase efficiency as long as the cost model is linear. But in practice, many systems have non-linearity in join costs introduced by factors like large intermediate results, whose size exceed the memory threshold of the system. In such case, the real optimal plan might be a bushy one and the "optimal" plan generated by the optimizer may be far from this optimal. Searching a large plan space definitely increases the chances of finding an optimal join ordering, but it simultaneously increases the optimization cost as more time and resources need to be invested.

### 2.3 Query Plan Space Enumeration

Join Order Optimization starts with the enumeration of subsets of join orders in the search space using an enumeration algorithm like bottom-up dynamic programming. The join orders are represented as binary trees (query trees) whose leaf nodes are the relations in the `FROM` clause of a query and the internal nodes are the join operations. There are several types of query trees:  

- **a) left-deep trees:** where the result of the join is used as an outer input for the next join.
- **b) right-deep trees:** where the result of the join is used as an inner input for the next join.
- **c) bushy trees:** performs a join based on the result of two other joins.
- **d) Zig-Zag trees:** combination of left-deep trees and right-deep trees.

The problem of finding an optimal join order is same as finding an optimal binary tree whose leaf nodes are the relations in the `FROM` clause of a query.

Dynamic programming can have the worst time complexity and does not necessarily enumerate all possible tree shapes. While the classical System-R optimizer restricts the search space to only left-deep join trees, systems like Oracle\(^4\) do not consider bushy trees at all. Optimizers of modern systems or systems that follow System-R architecture, completely ignore join orders with cross products. These restrictions and heuristics significantly reduce the complexity of the search problem. Based on the principle of optimality, dynamic programming enumeration strategies proceed by incrementally constructing optimal sub-plans. These strategies provide an efficient exploration of the search space as long as the cost of a join is linear in the size of the input relations. But very often large intermediate result sizes exceeding memory triggering partitioning can easily introduce non-linearity in join costs, where the heuristics of considering only left-deep trees might fail. Many

\(^3\)https://www.postgresql.org/  
\(^4\)https://www.oracle.com/index.html
commercial systems implement further heuristics but could require the intervention of a Database Administrator (DBA) to control the plan alternatives by either restricting those to some sets or configure the system to terminate the enumeration at some given threshold.

In this section, we overview both Non-Machine Learning (non-ML) and Machine Learning (ML) based State-Of-The-Art (SOTA) research work in the area of query plan space optimization in the last five years.

**Figure 2.2:** Taxonomy of machine learning and non machine learning based approaches for query plan space optimization.

**Non Machine Learning Based State-of-the-art**

With the aim of improving join orderings in the query plan space, *PlanBouquet* [DH16], an approach based on the concept of incrementally discovering the actual selectivity of a query based on the partial execution of a chosen set of query plans is proposed. An improved version of *PlanBouquet*, *SpillBound* [KHK+17], provides a platform-independent performance guarantee and accelerates this process. Extending this approach, *Frugal-SpillBound* [KHKP18] is claimed to further reduce the overhead often more than three orders of magnitude.

Given a query with a set of *error-prone predicate* (epp) (predicates whose selectivity is difficult to estimate), the proposed approach begins by constructing an *error-prone selectivity space* (ESS) as shown in the Figure 2.3. The ESS contains a series of *Isocost Contours* (IC) along with an associated optimizer estimated cost (CC), depicted in the Figure 2.3 as $IC_1C, IC_2|2C, IC_3|4C, IC_4|8C$ and $IC_5|16C$ respectively. The $IC_1$ at the origin of the space corresponds to the minimum cost for the execution of the given query and the following contour represents double the cost of the previous contour and so on. The plan bouquet is the combination of all the query plans that lie on these ICs. In
order to find an optimal plan, the SpillBound algorithm begins by sequentially executing the plans on the cheapest contour first, and then moving on to the successive ICs. The corresponding CC determines the threshold for the execution time. Here, the idea is to learn the selectivity of the given EPP. Once the algorithm discovers all the selectivities in the ESS, the process completes and based on this, an optimal query plan is determined and executed. The authors evaluate FrugalSpillBound using the TPC-DS [PNW07] workload (21 SPJ queries) with four to ten relations and join predicates only along with a modified version of PostgreSQL V9.4 for the experiment. Although promising, a major drawback of this approach is the assumption of a fixed database, which means the contour sets have to be rediscovered from scratch when the database size changes. Furthermore, the evaluation for FrugalSpillBound does not consider how well it adjusts to the real world datasets which often have many join crossing correlations.

Sharing the same objective of optimizing the query plan space, Plan Stitch [DDW+18] improves upon the reversion based plan correction (RBPC) by constructing a stitched plan from previously executed plans that is cheaper in execution cost than any other previously executed plan, using dynamic programming. However, the RBPC technique only considers the overall cheapest plan and does not consider using any subplans from previously executed plans that can be used to construct a new cheaper plan. The experiment uses MS SQL server engine for storing all the executed plans, the statistics and costs of execution, as well as the plan structures with TPC-DS and three real world customer workloads. Although Plan Stitch is said to perform better than the reversion based methods in terms of plan quality and execution overhead, the proposed approach too falls behind when the data distribution or database size changes due to stale execution feedback.

Lookahead Information Passing (LIP) [ZPSP17] provides a query execution strategy that increases the robustness of query plans despite the bad plan selections made by the optimizer by using Bloom Filters that store the membership information and an adaptive reordering algorithm that minimize the overhead on the filter lookups. Here, a lookahead filter passes information about the later join predicates to the earlier ones, thereby reducing the hash table probe costs. The experiments uses QuickStep database, three datasets: Star Schema Benchmark workload, TPCH-H [BNE13] at scale factor 100 and a synthetic dataset and the evaluations are made against a naive strategy that does not use LIP. The
experiment uses a hash join algorithm for the join operation, and the Bloom Filter uses a single hash function (identity hash function) and 8 bits per tuple. The drawback of this approach is that the order in which the bloom filters are probed vastly affect the required number of lookups. Additionally, LIP is limited to star schema data warehouses, SPJ queries and left-deep join query plans only.

**Machine Learning Based State-of-the-art**

Implementing a machine learning solution towards the vision of learning based optimizer, ReJOIN illustrated in the Figure 2.4, is a significant preliminary work carried out by Marcus et al. [MP18a] based on Proximal Policy Optimization (PPO) [SWD17]. It is a Deep Reinforcement Learning (DRL) [FLHI18] based join order enumerator for learning optimal join orderings, which uses only logical features about the query for training. As a reward it uses the cost of the plans. The model relies on the cost model of a native optimizer for estimations and does not perform physical operator selection, predicate selection or index selection. It uses the JOB workload for training on 103 queries and testing on the remaining 10 queries. The cost of the generated query plan is compared against the cost of the plans generated by PostgreSQL’s optimizer. Furthermore the plans generated by the model are evaluated against the plans from PostgreSQL and Quickpick. However, similar to the non machine learning approaches we explained above, ReJOIN too cannot adapt to database changes and would require re-training since no feature related to the new data is used during training.

![Figure 2.4: ReJOIN Framework. [MP18a]. Image sourced from [MP18a]](image)

The authors of ReJOIN, propose a vision for a hands-free optimizer [MP18b] providing three novel approaches for developing DRL-based optimizers: a) Learning from demonstration b) Bootstrapping the cost model and c) Incremental learning. Learning from demonstration is an approach based on training a model by observing logs of an expert system like PostgreSQL and after reaching a certain level, implementing the trained model in a real scenario. After observing the query latency, the model fine tunes its policy. Bootstrapping a cost model of a native optimizer as a reward signal is another approach to aid in the development of DRL optimizers. The idea of this strategy is to train the model first by using cost models, and then switch it to real world signals such as
query execution time. Likewise, Incremental learning is an approach where the model is first trained on simple queries, and is gradually (incrementally) switched to more complex ones.

**Figure 2.5:** Neo Model. Image sourced from [MNM+19]

*ReJOIN* together with the research on hands free optimizer, serves as initial works for *Neo* [MNM+19]. Based on *Deep Neural Nets (DNN)* [SCYE17], *Neo* as depicted in Figure 2.5, extends *ReJOIN* by also including physical operator selection and index selection in the learning. It uses *tree convolution* [RPR20] as part of the neural network design to provide the model with a feature representation that better matches the graph nature of the query plan. To overcome the cold start problem that comes with machine learning, *Neo* uses *PostgreSQL*’s optimizer to bootstrap its learning process. The experiment uses two open source databases (*PostgreSQL* and *SQLite*) and two commercial ones (*Oracle 12c* and *Microsoft SQL Server 2017*) using synthetic (*TPC-H* and *corp*) and real world dataset (*JOB*), where 80% of the sample data is used for training and 20% for testing. Despite all the advancement, however, *Neo* is restricted to queries of *select-project-equijoin-aggregate* form only as well as cannot generalize from trained data or query workload to unseen ones. And like the other approaches, it also cannot adapt to data changes either.

On the same note of learning, an approach namely *DQ Optimizer* [KYG+18] is proposed, which learns search strategies for optimal join ordering. Like *Neo*, *DQ Optimizer* too relies on the native optimizer for training data. Given a query as an input to a native optimizer, it produces a final query plan along with the cost estimation as well as many sub plans during the optimization process. The *DQ optimizer* uses these data produced by the native optimizer as samples for training. In order to evaluate if the model can generalize to unseen queries or not, the training set consists of joins with up to nine relations and the rest of the samples are used for testing. The experiments use *PostgreSQL* and *SparkSQL* servers and the evaluations are made against three baselines search algorithms: a) *QuickPick-1000* b) *IK-KBZ* c) *Dynamic programming* (any but only one type of tree). It is not clearly stated whether *DQ Optimizer* can adapt to the changes in the database but since it too uses the native optimizer for training data like *Neo*, *DQ Optimizer* too falls short when it comes to adapting to changes in the database. Moreover, it is also restricted to *Select-Project-Join (SPJ)* queries only.
Taking a different approach to learning optimal join orderings, SkinnerDB [TWM+19] uses UCT algorithm [KS06] which is a Monte Carlo Reinforcement Learning. SkinnerDB starts by filtering the base table on unary predicates and then divides the remaining tuples into batches. The total execution time is partitioned to tens of thousands. At the start of each time slice, different join orders are selected by the learning optimizer based on the statistics about the quality of join orders. The selected join orders are passed to the join executor that executes different join orders in different time slices, all of which have a threshold on execution time. Resulted tuples are then merged and checked for redundant results. The progress tracker stores the state of the current join order and tracks input data that is already processed. The reward calculator assigns the reward based on the execution progress during each time slice, and is used as input by the learning optimizer to recognize the quality of different join orders to use in the next iteration. The experiments are carried out on top of MonetDB and PostgreSQL by using dedicated knobs or a brute force technique to force the join orders and the comparisons are made on the basis of execution time. The lingering question here is about setting the "right" threshold for the execution time as well as it is uncertain if the model can adapt to database changes as the previous statistics will no longer hold true. Also, doubts arise on whether alternative models (bandits) could improve the performance of the system.

A noteworthy preliminary research, SageDB [KAB+19], provides a different outlook to the solution which can be summed up in a sentence: "learned components can fully replace core components of a database system". The conceptualized RL based model learns data structures and algorithms for all components of a database based on another machine learning models built to represent the data and workload distributions. The initial results include evaluations carried out to showcase how learned models can replace traditional index structures and how sorting can be learned rather than implemented. Authors also discuss future implications for join processing and for scheduling policies tailored to data or workloads. Although the concept fully addresses the major shortcomings of other proposed solutions i.e. not being able to adapt to data changes, however, SageDB is still conceptual and not fully realized by the author as of yet.

### 2.4 Cardinality Estimation

Many experts in this field and academia have time and again implied the fundamental importance of cardinality estimation for query optimization, stating that cardinality estimation errors significantly dwarf the errors introduced by the cost model. Cardinality estimation, which has been called the "Achilles Heel" of query optimizers, is therefore critical for query performance. Despite this fact, the optimizers of virtually all current generation commercial or industrial strength systems make the assumptions of uniformity and independence of data. Thus, they fail to detect complex patterns like join-crossing correlations, skew and non-uniform data distributions often found in real world data-sets. In recent years Leis et al. [LGM+15] showed that cardinality estimation errors propagate exponentially with the increase in the number of joins in the query, eventually leading to disastrous query plans.

Cardinality estimation refers to the task of estimating the size of the result of single join between relations (i.e., one item between all the pairs of joins in a plan). This is used to
evaluate the cost models and decide on plans. Cardinality estimation has been of core interest to many researchers, as only accurate cardinality estimates make the cost model a reliable predictor of query run-time, eventually leading to an optimal query plan. Often per attribute histograms or synopses providing statistical information about the data are used for cardinality estimation of base tables in join operations. Histograms divide the values on an attribute into a number of buckets, where each bucket describes the range of values on that column along with its average frequency and number of distinct values on that column. Although they provide accurate information about a single attribute, they fail to consider correlations among attributes. Furthermore, they make assumptions about uniformity, independence and containment. Depending upon the number of joins, some systems use a damping factor, in order to represent the uncertainty about their independence. Instead of using per attribute histograms or even multi-attribute histograms, few systems like Hyper implement sampling techniques. But the downside of sampling is that it only yields accurate estimates as long as the selectivity is not too low (i.e., selections yielding zero tuples on the samples).

Histograms (synopses) and sampling based approaches are widely used techniques for estimating cardinalities. Unlike per attribute histograms, sampling method to some extent address the correlation between the attributes. A naive sampling approach draws a random number of samples from each relations in a given query and then joins these samples. Besides the high sampling overhead, a major shortcoming of this method lies in determining a right sample size in order to avoid empty results. In this section, we review some of the State-Of-The-Art (SOTA) research works in the last five years that proposes more than histograms and samplings for cardinality estimation. We categorize them into Non-Machine Learning (non-ML) and Machine Learning (ML) based SOTA.

Figure 2.6: Taxonomy of machine learning and non machine learning based approaches for cardinality estimation.

**Non Machine Learning Based State-of-the-art**

Although based on the sampling approach, *Index Based Join Sampling (IBJS)* [LRG+17] takes it a step ahead by also taking index structures into consideration. The approach
starts in a bottom-up manner (two way join, then three way join and so on), by taking a random sample of one of the relations in the join operation first, and then looking for the sample’s corresponding joins in the index of the other attribute. The result obtained from this now serves as a sample for the next join operation, which again uses the index on the join attribute, and so forth. The somewhat constant sampling size and a time threshold for the sampling phase makes it better than a naive sampling method in terms of sampling overhead but does not address the zero-tuple situation that comes with sampling approach. The true estimates are injected to the native optimizer which runs again to generate a new plan. IBJS makes no changes to the cost model or the query plan space enumeration strategy. The experiments use three variants of IBJS differing in index lookup budgets (100k, 10k or no budget) with JOB workload to compare against PostgreSQL in terms of cardinality estimations, sampling overhead, plan quality and number of indexes. The paper claims that IBJS is five times faster than the native PostgreSQL approach, performing better in all four aforementioned cases. However, the two shortcomings of this approach are setting the right time threshold for the sampling phase and the zero-tuple situation.

Similarly, Bandwidth-optimized Kernel Density Estimation (KDE) [KHBM17] uses probability density functions from data samples for a multi dimensional selectivity estimation. A KDE model places local probability density functions (kernels) around the sample points drawn from the data distribution of base table or a join sample. Based on this, the overall probability density function is estimated and a final estimate is computed by averaging over all kernels. A hyper parameter, bandwidth, is used to control the spread of the local probability distribution. Two variants of KDE (based on join sample and based on table sample) are compared against PostgreSQL, AGMS Sketch, naive Table Sample, Join Sample and Correlated Sample estimators on three datasets: IMDB dataset (5 tables), Shifted Normal (100k tuples) and Dept. of Motor Vehicles dataset (23 columns, 269 to 430k tuples). The accuracy of the estimators are compared against each other and the results of the model are shown to outperform other estimators while always significantly better than a naive sample-based estimator if not the best. The main point of concern here is setting the right value to the bandwidth parameter as too low of a value would cause over-fitting and too high would cause over-smoothing.

Taking a completely different direction by leaving out the synopses and sampling methods for the cardinality estimation, Exact Selectivity Computation (ESC) [SRS19] instead computes an exact selectivity by executing an additional aggregate query. The experiments are carried out by implementing ESC in the query optimizer of MapD’s open-source database, by executing 4 queries each with more than three relations from the TPC-H workload and all queries with more than two relations from the Star Schema Benchmark (SSB) at scale factor 80. Since ESC uses exact selectivity rather than sampling or statistics like histograms, it overcomes the downside of sampling and histograms, but the proposed approach is infeasible in regards to the time it takes for the computation. Furthermore, the work is intended for in-memory and GPU accelerated databases and is not expected to perform well in disk-based databases as per the paper [SRS19].

**Machine Learning Based State-of-the-art**

While there has been many approaches to query representations when it comes to applying machine learning solutions for cardinality estimation problem, Ortiz et al. [OBGK18]
proposes a DRL based model that incrementally learns to generate a concise representation of sub queries of a query based on the observed cardinality estimates. The model architecture as shown in the Figure 2.7, uses the properties of the whole database (min, max, distinct values and per attribute histograms) as its initial state and one single query operation (selection or join) as an action in the first phase. It outputs some representation of the intermediate results, which then serve as an input state in the next time step. Given this representation as a new state and a single relational operation on this subquery as an action, the model learns to generate a representation of this subquery’s intermediate results (properties) using a neural network and predicts the cardinality of this subquery given the action. Based on the predicted estimate vs the true cardinality, the weights are adjusted using back propagation so that the model learns to generate succinct state representations based on the observed cardinality estimates. The process continues recursively where the representation of a previous subquery is propagated further in the network as the next subsequent state. The approach addresses the difficulty of representing queries when implementing machine learning algorithms and provides a unique solution but suffers when the number of attributes in a query is more than five as it would require gathering aforementioned properties for each of these.

Figure 2.7: State Representation [OBGK18]).

Looking at the selectivity estimation as a neural density estimation problem, Hasan et al. [HTA+19] proposes an approach which uses an auto-regressive decomposition method to decompose the joint distribution into multiple conditional probability distributions instead of directly estimating the joint probability distribution. While Bayesian networks that increases the overhead by storing the conditional probability tables, the proposed model uses neural networks to instead learn these conditional probabilities rather than storing them. The model is based on MADE, the author’s own work, that encodes the queries and the selectivities in a form suitable as an input to a neural network. The authors also suggest an incremental learning approach where the existing model is trained only on the new data instead of retraining the model from scratch, in order to combat the scalability challenge. While doing so, the authors further suggest to reduce the number of training epochs and the learning rate and to use the dropout technique to prevent forgetting. Albeit the approach addresses issues like scalability, the major disadvantage of this approach is determining the right ordering of attributes for the auto-regressive decomposition as different decomposition would lead to different performances and hence
cannot guarantee a robust solution.

Addressing the zero-tuple situation with sampling methods and join-crossing correlations of real world databases, Kipf et al. [KKR+18] propose a model called as Multi-Set Convolutional Network (MSCN) that uses set-based query representations following the Deep Sets [ZKR+17] paradigm in conjunction with materialized base table samples. It is the first research work to implement Deep Sets for cardinality estimation problem. The model architecture as seen in the Figure 2.8, takes sets of Tables, Joins, Predicates in a query as input and feed them to the neural network to transform them. Mean aggregation is applied to the transformed outputs of each individual sets which is then concatenated into one single representation and passed as input to the neural network for computing the cardinalities of the queries. The authors use 90 thousand out of 100 thousand randomly generated queries with joins of up to two, and predicates consisting of only numeric values for training the model, while the remaining 10 thousand are used for validation. Additionally, for both training and testing workloads, the authors use sample bitmaps to provide the model with additional information about the data in the database. The model is evaluated on two synthetic workloads and on JOB Light. The model is said to be good at generalizing the queries of higher complexities and multiple joins in JOB Light but like other approaches, MSCN too would require re-training when the database changes. Moreover, the sample bitmaps used for the training would no longer reflect the state of the database and similarly would require a new set of samples as well when the data changes.

Extending the research on MSCN, the same authors also introduce Deep Sketches [KVM+19]. As depicted in Figure 2.9, it provides users with a graphical interface to select training parameters such as the number of tables, the number of epochs, the number of training queries and the number of materialized base table samples. The underlying model of Deep Sketches is still MSCN and hence it suffers from the same shortcomings of MSCN. But apart from the fact that the model would require re-training when the database changes, the work remains a State-Of-The-Art as of now.

5 A susbset of JOB workload with a total of 70 queries consisting predicates on numeric values only.
Figure 2.9: Deep Sketches Framework. (Image sourced from [KVM+19]).

After this survey new work has continued emerging (as of late 2019), but we decided not to include it, making the 1st of September of 2019 our cut-off date for the review, dismissing work after that date.

2.5 Deep Sets Learning

Application of machine learning has expanded to several domains from image classification to natural language processing and speech recognition but machine learning models are fundamentally based on fixed sized vector inputs. Although an exception to this, Recurrent Neural Networks (RNNs) cater to variable length input, RNNs implicitly assume sequential data. Despite this, there are several machine learning tasks that are naturally defined on a set such as point cloud classification [QSMG17], training set of reinforcement learning agents [SLG+17], unstructured set of tags [RAY+16] and so on. Hence, recognizing the need for order invariant neural architectures that are able to accept a variable size input set and compute an output that is invariant to the order of the input, Zaheer et al. [ZKR+17] in 2017 introduced permutation invariant function, which was later refined by Wagstaff et al. [WFE+19] in 2019. According to the proposed definition [ZKR+17], a function \( f(x) \) on population \( X \) from countable particle space is invariant if and only if there exists a decomposition, with appropriate functions \( \rho \) and \( \phi \). Such functions \( \rho \) and \( \phi \) are called sum decomposition.

\[
f(X) = \rho(\sum_{x \in X} \phi(x))
\]

Permutation invariant functions allow for sum decomposition and thus enable a direct application of neural networks. Aggregation functions like summation, mean aggregation, max aggregation, logsumexp are the core component of Deep Sets architecture that help to map set based inputs to fixed sized representation.

Here, the Figure 2.10 depicts the structure of Deep Sets Architecture where, \( \rho \) and \( \phi \) do not operate on sets but rather are modelled by the neural networks. Each element in the set of input are embedded individually using the neural networks to get such representation. Each of these embeddings are then aggregated using summation as aggregation function, into an order invariant fixed sized mapping, which is then processed with by the neural network again to derive final results.
2.6 Word Embeddings

Machine learning models take vectors as input, but tasks like text classification, natural language processing, document clustering all deal with textual data. In order to work with such data, first we have to transform them into numeric values which can then be fed to the model. A simplest transformation of a word could be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else. However, given the size of a vocabulary which is often millions of words in real world cases, one-hot encoding becomes impractical pretty soon as this will require thousands or millions of dimensions for sparse representations. Overcoming these issues, word embeddings not only maps words and phrases into dense vectors but does so with much lower dimensions. Furthermore, it creates a representation that captures the approximated similarity between words as well the hidden semantic relationship between them given the context. Thus, similar words appear closer together in the vector space.

In 2013, Tomas Mikolov et al. [MCCD13] at Google developed a prediction based embedding method, Word2vec, which has become a standard for various Natural Language Processing (NLP) tasks. Word2vec is a technique to learn word embeddings for a given text corpus using a window of surrounding context words, where the window is a configuration parameter of the model. Word2vec provides an option to choose between two models, which can be passed as a parameter to the model during training.

- **Continuous Bag-of-Words (CBOW) model**

  CBOW model learns the embedding for a target word by predicting the target word based on context words i.e. using a fixed sized window of neighboring word or words. It takes the average of the context words. They provide better representation for frequent words.

- **Skip-gram model**

  Opposite to CBOW, skip-gram model learns the embedding for the target word by predicting the probabilities of a word being a context word for the target word. They provide better representation for rare words.
Although one can train own embeddings from scratch, this could require training on a large corpus with possibly billions of words in order to derive meaningful embeddings. Therefore, various pretrained embeddings besides Google’s Word2vec are available today such as Stanford’s Global Vector (GloVe) introduced by Pennington et al. [PSM14] that combines feature based embedding and prediction based embedding together and fastText [BGJM17] from Facebook, which is pre-trained on a large corpus obtained from web crawl and Wikipedia dataset with 16 billion tokens. These models can be used as they are, out of the box, or depending upon the requirement of the project, they can be trained on a new dataset. Given a word, these models provide an embedding which is a vector of floating point values with a default dimension of 300, but can be configured to reduce upto 100 dimension.

2.7 Summary

In this chapter we reviewed the key concepts of join order optimization, complementing it with a focused survey of related work on join query plan improvements and cardinality estimation. For each of the tasks we presented the ML and non ML state of the art. We identified MSCN as the state of the art for cardinality estimation, and hence we concluded by presenting the related concept of deep sets and the related concept of word embeddings, which could be used in unison with the MSCN, to improve queries with string predicates. In the next chapter we describe the design of this thesis.
3 Prototype Design

This chapter comprises of the core structure and components of our model.

- We begin by establishing the research questions that will guide our evaluation and design, in Section 3.1.
- We elaborate on our model architecture based on the Deep Sets, in Section 3.2.
- Section 3.3 gives a brief explanation of the word embedding models we use to encode the predicates with string values.
- In Section 3.4 we state the database and benchmark workload we use for our experiment and evaluation.

3.1 Research Questions

In particular, this work addresses the following research questions:

1. How do different configurations of the hyper-parameters impact the overall performance of the cardinality estimation model?
2. What is the impact of using word embeddings for query representations with the cardinality estimation model?
3. How well does the model compare against baseline regressors?

3.2 Model Architecture

In order to address our research questions, the very first step towards modelling a given task at hand is to study the problem domain. Unlike images and large corpus of textual or numeric data, feature representation of SQL queries in order to achieve meaningful insights from the application of machine learning algorithms is still an open challenge. Given a SQL query, what do we consider as 'features' for our machine learning task? In a simplest form, we can consider a Select-Project-Join (SPJ) query as a collection of table/s, join/s, predicate/s and predicate operator/s. A query consists of varying number of tables, joins, predicates and implies no sequence or any kind of relationship among these components. Therefore, representing these queries as a collection of sets of tables, joins and predicates as proposed in the research works like MSCN [KKR+18], based on the concept of Deep Sets [ZKR+17], we can featurize them for the machine learning implementation. In our work, we proposed to implement this concept of using sets as input for our neural network model.
In order for our neural network model to be able to accept an input set with the properties of having variable size, and being order invariant, we have to implement permutation invariant functions (such as aggregation functions) that allow for sum decomposition. These aggregation functions induce invariance to any ordering of the elements in the set, by mapping variable-sized set inputs to a fixed-sized representation. Various aggregation functions exist such as mean aggregation, max aggregation, log-sum-exp, among others. According to the authors Soelch et al. [SAvdSB19], who propose learnable recurrent aggregation functions, the choice of aggregation function highly impacts the overall performance of Deep Sets networks. Though this is possibly a hyper-parameter, in this work we consider only a mean aggregation work, as explained in subsequent sections.

We present our model architecture in Figure 3.1 which lays out the concept of our network design.

![Figure 3.1: Set input based neural network architecture that uses mean aggregation function and fully connected hidden layers for transforming the input and later to process the order invariant representation of the transformed inputs.](image)

The architecture of our model as depicted in the above figure 3.1, takes sets of tables, joins and predicates as input for a given batch of queries and using separate fully connected neural networks (as components), transforms each input to an intermediate representation. After this each representation is averaged, leading to an output that is independent of the size of the inputs (e.g., the number of tables in a query). These representations are then concatenated and fed to a subsequent sequence of layers which finally produce an output a predicted cardinality. Hence, we adopt a model architecture that follows the key ideas of the MSCN, as proposed by Kipf et al. [KKR+18].

Further noteworthy aspects of this model are the featurization of the components. Tables
and joins can be featurized as one hot encodings. Numeric predicates can be mapped to a normalized representation, with operators and column names also being provided as one hot encoded values. Finally, for string predicates we propose to study word representation models (word embeddings), as explained in the next section 3.3.

In the following Chapter 4, we discuss in more detail how we featurize the different input components for our model.

### 3.3 Word representation models

To obtain the vector representation for the predicates with string values, we use the pre-trained fastText model which is trained on Wikipedia and web crawl dataset with 16 billion tokens. Currently it provides pre-trained models for 157 different languages. For our task, we use the model for English language, although our IMDB dataset contains more than a hundred different dialects and languages and hence, using a pre-trained model in English language is a questionable design choice, but without suitable alternatives we have decided on this choice.

The default fastText models are trained using **CBOW** with the window size of 5 and 10 negatives, character n gram of length 5 and vector length of 300 dimension, which are all default parameters of the model for the training as well. Training the model can be very straightforward with just the training corpus without providing any arguments for the hyper parameters. But to get a close and more meaningful vector representation and fine tune the model, it provides a list of configurable hyper parameters 3.1 for training as well as an option to compress the file sizes to keep the model size small.

The list of configurable hyperparameters available for the training of unsupervised fastText model: (Adapted from the fastText website:1), is presented in the code snippet 3.1.

---

1https://fasttext.cc/docs/en/python-module.html
3.4 Database and benchmark workload

Considered as de facto workload, synthetic benchmarks like TPC-H, TPC-DS are widely adopted benchmark for many research works in the field of databases. These benchmarks provide a quick and easy way to generate a large dataset and query workloads with desired scaling factors, by using the database generation tool: dbgen and query generation tool: qgen. But these synthetic datasets are not representative real world queries which are often very complex and consist of join crossing correlations among attributes. As such, experiments and evaluations carried out on such workloads, especially for cardinality estimation, may produce poor results when implemented on real world databases, as they make assumptions about the data (like independence among the attribute values generated, or the absence of correlations among them). Therefore, in our work, we use the real world dataset Internet Movie DataBase (IMDB)\(^2\) as our database. It has 3.6GB in size, and comprises of 21 tables, the largest of which ("cast.info") has 36 million rows.

For our benchmark workload, we use the Join Order Benchmark (JOB), which is based on the IMDB dataset, as introduced by Leis et al. [LGM\(^+\)15]. The JOB workload contains 33 query templates that form a total of 113 queries, each consisting of 3 to 17 joins with an average of 8 relations per query. The distinct presence of correlations in the dataset, as well as the significant number of joins in the JOB query workload, as depicted in the Figure 3.2, makes this benchmark challenging and suitable for the evaluation of our research work.

Figure 3.2 shows the occurrence of joins in the different 32 templates. It is possible to observe that templates with a higher number have more joins in them. Similarly, it is possible to observe that each join participates in many templates.

\(^2\)The CSV files for this version of the dataset can be found at http://homepages.cwi.nl/~boncz/job/imdb.tgz
Figure 3.2: Heat-map of all the join co-occurrences in the JOB workload

3.5 Baseline Regressors

In order to evaluate how close or far off our trained model is from simple baseline regressors, we select some regressors that are optimal for regression task and use the JOB dataset with the same train-test split as our model evaluated in experiment 5.2. We pick seven regressors mentioned below for this task.

- Decision Tree Regressor
- MLP Regressor
- Ridge Regressor
- Lasso Regressor
- SVR Regressor
- Random Forest Regressor
- Linear Regressor

3.6 Summary

In this chapter we presented the design for the model we used in our evaluation of cardinality estimation with deep learning. We base our design on the MSCN architecture, as proposed by Kipf et al. [KKR+18]. We presented our architecture, and we reviewed the featurization of the attributes. In a dedicated section we introduced the basic aspects of
the word embedding component of our architecture. Finally, we reviewed the database and benchmark workload as used in our work. In the following chapter, we explain our experimental setup.
4 Experimental Setup

The following sections provide an overview of the setup used in order to proceed with the experiments outlined by our research questions. We aim with this chapter to provide details that could contribute to making our evaluation reproducible.

- We begin by describing the database system and dataset used in Section 4.1.
- We describe our implemented solution for query featurization in Section 4.1.2.
- The aspects of training and employing word embeddings, is presented in Section 4.1.3.
- In Section 4.2 we discuss about the hyper-parameters used.
- In Section 4.3, the configurations of software libraries and hardware used is described.

4.1 Dataset Generation and Pre-processing

The first step in any machine learning task is to gather a training and test dataset. Since we are using JOB for benchmarking which is based on the IMDB database, our training dataset also needs to be representative of the test dataset. Synthetic datasets like TPC-H, TPC-DS come with tools to populate and generate queries of scaling factor of choice for performing the benchmark evaluations. But as of now, there are no such tools for generating random queries on the IMDB database.

We started our initial experiments using only JOB workload for both training and testing purpose. Therefore the experiments and evaluations presented in our work are carried out on the JOB workload with queries of template 1 to 20 in train dataset and all queries of template 21 to 33 in test dataset. Towards the end of our experiments, we make a small experiment with our own dataset consisting of the randomly generated queries to see if the dataset can improve the performance of the model since the size of JOB workload is very small.

Preparing a large number of random but valid SQL queries with varying number of joins, predicates and different combinations of tables depending upon the relationship between the tables in the database and also executing each one to get the true cardinality estimates is a challenging and time consuming task which adds up to the overhead of using the machine learning solution. Although a large training dataset could possibly improve the training process, additional features like samples, statistics about the database and data distributions would improve the generalization ability of the trained model.
4.1.1 Random Query Generation

In order to create a large dataset, we create a random query generator to generate queries based on the IMDB database. For this task, we rely on the existing relationships between tables in the database. The queries we generate, each vary in the number of tables, joins and predicates with a maximum of two joins and twelve predicates in a single query. We execute the generated queries on PostgreSQL 11.7 server to get true cardinality estimates (labels for our training dataset). On large tables like `cast_info`, `movie_info`, `movie_companies`, each with over 36 million, 14 million and 12 million tuples respectively, execution time for queries increase gradually with the increase in the number of joins and number of selection predicates. Given the time constraints for this research work, there is a trade-off between highly curated training queries with high complexities versus less complex but large numbers of queries for the training. We generate a total of 75,350 queries from all twenty one tables in the database, having a maximum of two joins only but including predicates on both string and numeric values. For the query generation, we filter out some columns which are either null or are not the attributes that are queried upon for example hashes like `md5sum`.

The Figure 4.1 shows the existing relationship between tables in the IMDB database based on the foreign key attributes. Our random query generation algorithm selects the tables for the join operation based on these existing relationships. For generating predicates of the selected table/tables, we pick one or more random values from the distinct values of the selected table/tables. It is important to mention that the large tables like `cast_info`, `movie_info`, `movie_companies` also have large number of distinct values and thus our randomly generated query dataset might not include a large number of these values.

4.1.2 Query Featurization

To featurize the SQL queries, we encode each individual query in our dataset into sets of Tables, Joins, String Predicates and Numeric Predicates as depicted in the model architecture diagram. Given a single query, there can be one to many table/s, none to multiple joins, one to many predicate/s with string values and/or numeric values. Hence, our dataset is a collection of sets of Tables, Joins, String Predicates and Numeric Predicates. Each of these individual sets (Tables, Joins, String Predicates and Numeric Predicates) can have one to many elements. We one-hot encode all 21 tables and 108 columns in our database. For a given query, we decompose the query clauses (`SELECT` clause, `FROM` clause, `WHERE` clause) into separate blocks and scan them to obtain information about tables, joins and predicates. From the `FROM` clause of a given query, we extract the names of the tables involved in the query and replace them with their respective one-hot encoded values. In a similar manner, the `WHERE` clause provides us the joins and predicates information. We extract the attributes participating in the join operation and for each pair of the join, we (a) Sort the participating attributes alphabetically so that they have same encoding after concatenation regardless of their position on the right or left hand side of the equation. (b) Get their respective one-hot encoding. (c) Concatenate the encoding.
For the predicate/s in a query, we categorized them into string predicates or numeric predicates. In both cases, we encode the columns and the operator in the predicate set with their respective one hot encoding as seen in the Figures 4.4 and 4.3. For predicates on numeric values, we normalize them in the range between 0 and 1 using the maximum and minimum values of the respective attribute. For the true cardinalities or the labels, we normalize them in the range of \([0,1]\) using the maximum and minimum of the logarithmic values. Since the cardinalities values range from 1 to 3 Million, we take the logarithm of the values first to even out the distribution.

We illustrate this process by presenting an example of this process with the query (13a.sql) from the JOB.

Figure 4.2: One hot encoding for columns in our model
Figure 4.3: One hot encoding for operators in our model

<table>
<thead>
<tr>
<th>Operator Names</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>&lt;</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>&lt;=</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>&gt;</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IN</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NOT IN</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>IS NOT</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>LIKE</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>NOT LIKE</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 4.4: One hot encoding for tables in our model

```
SELECT MIN(mi.info) AS release_date,  
       MIN(miidx.info) AS rating,  
       MIN(t.title) AS german_movie  
FROM company_name AS cn,  
     company_type AS ct,  
     info_type AS it,  
     info_type AS it2,  
     kind_type AS kt,  
     movie_companies AS mc,  
     movie_info AS mi,  
     movie_info_idx AS miidx,  
     title AS t  
WHERE cn.country_code ='[de]'  
AND ct.kind ="production companies"  
AND it.info ="rating"  
AND it2.info ="release dates"  
AND kt.kind ="movie"  
AND mi.movie_id = t.id  
AND it2.id = mi.info_type_id  
AND kt.id = t.kind_id  
AND mc.movie_id = t.id
```
AND cn.id = mc.company_id
AND ct.id = mc.company_type_id
AND miidx.movie_id = t.id
AND it.id = miidx.info_type_id
AND mi.movie_id = miidx.movie_id
AND mi.movie_id = mc.movie_id
AND miidx.movie_id = mc.movie_id;

Listing 4.1: Query 13a.sql from JOB

To encode the predicates on string values, we use the open source word embedding library from Facebook, fastText\(^1\), to get the vector representation of the given string value, which gives a vector with the default 300 dimensions unless parameterized specifically. FastText provides two methods \texttt{get\_word\_vector()} which is equivalent to \texttt{get\_words()} and \texttt{get\_sentence\_vector()} to get vectors for a given word/s. The string predicate values in our dataset ranges from only a character like \texttt{A} to words like Metro-Goldwyn-Mayer Pictures. Hence we use \texttt{get\_sentence\_vector} function to get the vectors. For some predicate values like \texttt{IS/IS NOT NULL} or \texttt{LIKE \%(co-production)\%}, it would benefit from a different approach like the works of Sun et al. \cite{SL19}. But due to the time and scope limitation of this thesis, we simply encode them with the fastText’s word representations. For the word embedding, we preprocess the values removing the special characters and symbols. Additionally we also convert the numbers to word representation first using the \texttt{num2words}\(^2\) library before passing to fastText for embedding. Furthermore, we also make adjustments like encoding values for attributes like \texttt{gender} with \texttt{male/female} instead of using \texttt{m/f}, in order to get more meaningful representations.

Addressing our second research question, we prepare two datasets with word embeddings from the default pretrained fastText model and another with the embeddings from the fastText model trained on the IMDB database. For the unseen words/words not in the model’s vocabulary, the default pretrained model computes the vectors by adding the \texttt{n-grams} that constitute the given word. This makes it interesting for us to observe whether this impacts the overall performance of the model or not. In the evaluation chapter 5, we show the impact of these embeddings and the results obtained from the model.

We use PostgreSQL’s \texttt{EXPLAIN ANALYZE} command to get the estimations from PostgreSQL’s optimizer for the queries in the JOB. We replace the aggregate function \texttt{MIN} in the queries with \texttt{SELECT asterisk} to get the estimates. These estimated values are compared against the predictions made by the model in the Chapter 5.

\(^1\)https://fasttext.cc/
\(^2\)https://pypi.org/project/num2words/
4.1.3 Word Embedding

To train the fasttext model on our IMDB database, we first prepare the training corpus to feed the model by filtering-out the columns with character/text data from the integer ones. From this list we further discard the columns like \texttt{md5sum hash}, postal codes, columns with only null values and columns that have blocks of sentences/paragraphs as values, since these are not queried upon, and some of them are simply categorical. From all the tables with the selected columns, we retrieve all the tuples from the table selecting these columns.

For the data we gathered, we apply basic pre-processing to remove special characters from the data as well as convert numbers to words for example: \texttt{39106 Magdeburg} would be \texttt{thirty nine thousand one hundred six magdeburg} which is one of the pre-processing requirement for using fastText. The pre-processed data is then added to the training corpus one word per line with \texttt{UTF-8 Encoding}, which is another mandatory configuration for \texttt{fastText}.

```python
import fasttext

model = fasttext.train_unsupervised("imdb_corpus.txt",
    model="cbow",
    epoch=1000,
    minn=2,
    maxn=6,
    dim=300)

model.save_model("imdb_model.bin")
```

Listing 4.2: Training and saving a FastText model
As shown in the code snippet 4.2, we train the model with `train_unsupervised` function, with `cbow` model to compute the word representations. The total size of the training corpus was 145 Million words with 289410 unique words. It is important to note that the model comes with a limitation on vocabulary size and maximum number of tokens per line at 30 Million and 1024 respectively as well as expects the words to be UTF-8 encoded. It also appends an EOS token to the line when a newline character is encountered. The default epoch value for the model is set to five which means every example in the training corpus is seen five times only, and hence, should be configured given the size of the training corpus at hand. After the training, we once again get the embeddings for all the JOB queries having predicates with string values and out of the 179 distinct words in the queries, we find 107 exact matches in our model’s vocabulary.

### 4.2 Model Hyper-parameters

Among the hyper-parameters of our model we considered the following:

- **Optimizer**: When training neural networks, the optimizer (i.e., the algorithm used to minimize the loss function) is a core component. For our work we evaluate the impact of this hyper-parameter.

- **Learning rate**: Apart from the optimizer we evaluate the impact of the learning rate, which is a number that indicate the strength for which new observations make the model update its beliefs.

- **Activation function**: At the end of each layer, there is the possibility of modulating the output computed through the use of an activation function which takes the output and applies a linear change on it. Some examples are Tanh, ReLU, Leaky ReLU, among others. In our work we test the impact of this hyper-parameter.

- **Batch size**: Neural networks are trained on batches of input data. For our case we used a fixed batch size, of 32 items, randomly sampled from the set of train queries.

- **Iterations**: These are the number of passes and batches we form. We used a default of 3000 iterations, with 4 calls to the train operation per iteration.

- **Dropouts**: This refers to a regularization technique where we drop out randomly (given a pre-defined probability) the contribution of certain units to the output of the overall network. We evaluated some configurations of this hyper-parameter.

- **Vector lengths**: This refers to the size of the internal representation (pre-concatenation) of the components. We evaluated changing this hyper-parameter, considering the same size for all components or different sizes.

- **Layers (number and sizes)**: We used 2 layers for the components (tables, joins, predicates), with an input size equal the encoding size of the components, and an output size equal to the vector length. Internally we used layers of size 32.

- **Layer types**: We used fully-connected layers, and did not evaluate tuning this hyper-parameter.
It should be noted that we developed a method for steering the queries selected for training (to up-sample without repetition), such that queries for which a large error was achieved were selected more often. This method was used for all cases where the complete queries were used for training.

### 4.3 Experiment and Evaluation Environment

We use PostgreSQL V.11.7 server to create and store IMDB database and use psycopg2\(^3\) to execute queries on the PostgreSQL database to get true cardinalities as well as PostgreSQL’s estimations.

On our local machine with NVIDIA GeForce GTX 1050 Ti GPU, we use Anaconda\(^4\) to set up virtual environments and create ecosystem of libraries and modules needed for our task. For our experiments we use Tensorflow V1.14, Cuda 10.0, Jupyter\(^5\) Notebook V6.0 with Python\(^6\) V.3.7 runtime, Tensorboard V.1.14 to visualize the tensorflow summaries and scikit-learn\(^7\), pandas, numPy\(^8\), matplotlib\(^9\), seaborn\(^10\) to pre/post process dataset and results and for visualizations. For the model interpretation, we use SHAP\(^11\) and intercepts V.0.3.1\(^12\) libraries.

For the most part, we used Colaboratory\(^13\) or just Colab for our experiments. It is a cloud based service from Google research which is based on the open source project Jupyter. These web based notebooks by default use Python V.3 runtime and support its ecosystem of third-party tools. They can be created and saved on Google Drive under Colab Notebooks but one can use Github\(^14\) to load the notebook as well save them too. But with fluctuating limitations on the usage of the resources as well as hardware availability, oftenly executing high computational processes or using more resources leads to temporary access restrictions to both GPUs and TPUs. In such cases, the notebook can be executed using the local machine for runtime. Although there is no option for the user to choose the type of GPU, according to the service providers of the Colab, Google itself, the available GPUs as per Google are often NVIDIA K80s\(^15\), T4s\(^16\), P4s and P100s\(^17\), some of them were used in our training.

---

\(^{3}\)https://pypi.org/project/psycopg2/

\(^{4}\)https://www.anaconda.com/

\(^{5}\)https://jupyter.org/

\(^{6}\)https://www.python.org/

\(^{7}\)https://scikit-learn.org/

\(^{8}\)https://numpy.org/

\(^{9}\)https://matplotlib.org/

\(^{10}\)https://seaborn.pydata.org/

\(^{11}\)https://shap.readthedocs.io/en/latest/

\(^{12}\)https://pypi.org/project/intercepts/

\(^{13}\)https://colab.research.google.com/

\(^{14}\)https://github.com/

\(^{15}\)https://www.nvidia.com/en-gb/data-center/tesla-k80/


4.4 Summary

In this chapter we considered the set of details we identified as crucial for the reproducibility of our evaluation. In specific, we described the database system and dataset used, the approach to query featurization, details on our word embedding models, the hyperparameters of the model, and finally the hardware/software used.
5 Evaluation and Results

In this chapter, we present the results of our prototypical evaluation to understand the impact of hyper-parameters on the deep sets based model that we study, and to understand possible approaches towards extending such model to support queries with string predicates. The chapter is organized as follows.

- We begin by re-instating the research questions in Section 5.1
- We present the evaluation results for our first research question about hyper-parameter tuning in Section 5.2 consisting of the experimental setups, the results we achieved and brief discussions on the results.
- Similarly, Section 5.3 provides the evaluation findings for our second research question regarding word embeddings. At the end of the section, we provide summary plots obtained from the model explanation functions as well as a comparison of the results against PostgreSQL.
- Section 5.3 covers the evaluation results for our last research question concerning baseline regressors.

5.1 Research Questions

In particular, this work addresses the following research questions, as stated previously:

1. How do different configurations of the hyper parameters impact the overall performance of the cardinality estimation model?

2. What is the impact of using word embeddings for query representations with the cardinality estimation model?

3. How well does the model compare against baseline regressors?

5.2 Evaluation Measures for Research Question I

Hyper-parameter Tuning- Weights Initializer and Activation Function: Setup

Given the network architecture 3.1 illustrated in the previous Chapter 3, as a starting point, in our first experiment we explore the right combination of weights initializer and activation function to use in order to initialize the network. Poor weights initialization
or some combination of weights and activation function result in vanishing or exploding gradients leading to model learning nothing at all even after training. Hence, instead of using a naive normal initialization, we start our experiment with two popular weights initializers: the Glorot/Xavier initializer and the HE initializer with both normal and uniform distribution, each paired with one of the four activation functions: ReLU [Aga18], Leaky ReLU [RZL17], TanH [LBH15] and Swish [RZL17] as depicted in the Table 5.1. Research shows that Glorot initializer is good when using squashing activation functions like TanH and Sigmoid whereas the variance scaling or the HE initializer perform better with ReLU and derivative of ReLU like LeakyReLU as the activation function.

For this initial setup, we choose the learning rate of 0.001 which is not too high or not too low on the spectrum and Adaptive Moment Estimation (Adam) [KB14] as our choice of optimizer since Adam is known to converge faster without much tuning making it a good option to start the training with.

Unless otherwise stated, as mentioned in the initial configuration 4.2, all of the experiments covered in this section are carried out using 3000 iterations with 4 calls to the training operation per iteration with a batch size of 32 randomly selected queries from the training dataset. For the loss function, we use Mean Squared Error (MSE) of predicted cardinality values and target cardinality values on the test dataset where target refers to the actual cardinalities of the queries. To reiterate once again, we split the JOB workload into 60:40 ratio for training and testing purposes. All the tests in this section are carried out on the test split of the JOB queries.

<table>
<thead>
<tr>
<th>Weights Initializer</th>
<th>Activation Function</th>
<th>Optimizer</th>
<th>Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glorot Initializer (Normal)</td>
<td>ReLU</td>
<td>Adam</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>LeakyReLU</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TanH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Swish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Glorot Initializer (Uniform)</td>
<td>ReLU</td>
<td>Adam</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>LeakyReLU</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TanH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Swish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HE Initializer (Normal)</td>
<td>ReLU</td>
<td>Adam</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>LeakyReLU</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TanH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Swish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HE Initializer (Uniform)</td>
<td>ReLU</td>
<td>Adam</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>LeakyReLU</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TanH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Hyper-parameter Tuning: Four weights initializers with four activation functions each at 0.001 as learning rate with Adam as the optimizer.
Hypothesis

We expect the model initialized with HE initializer with uniform distribution to be better when \textit{ReLU} or \textit{LeakyReLU} is the activation function and the model initialized with Glorot initializer with uniform distribution to be better with \textit{TanH} as the activation function.

Experiment Results

We conduct our first experiment with weights initialization with configuration stated in the Table 5.1. For each weights initializer, we pick the best activation function i.e. the activation function with the least loss or MSE and MAE.

We start with the first configuration from the table 5.1 with Glorot Initializer with normal weights distribution. Figure 5.1 shows the training loss for the four activation functions. Each of the colorful lines in the figure represents an activation function labelled in the legend. Here the y-axis represents the loss function: \textit{MSE} for each of the four configuration. We see that all four activation functions are able to minimize the loss as the training progresses.

![Figure 5.1](image1.png)

\textbf{Figure 5.1:} Training loss per iteration for the four models with four different activation function but the same weights initializer: \textit{Glorot Uniform}.

To get better sense of the loss graphs we can look at the Figure 5.2, where we enable the option of ignoring the outliers in the chart scaling in \textit{Tensorboard}. For each loss plots presented in this chapter, the associated graphs with zoomed in views are added to the Appendix A..

![Figure 5.2](image2.png)

\textbf{Figure 5.2:} Scaled plot for the training loss with weights initializer: \textit{Glorot Uniform}. 
Similarly, the Table 5.2 shows the loss on the test dataset in terms of MSE and MAE for these four configurations. We see that although Swish has the least loss among all others, with TanH close by considering both MSE and MAE.

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU</td>
<td>0.02330442305</td>
<td>0.1285350548</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td>0.02627053701</td>
<td>0.1282378393</td>
</tr>
<tr>
<td>TanH</td>
<td>0.02488231234</td>
<td>0.1225472111</td>
</tr>
<tr>
<td>Swish</td>
<td><strong>0.02380513171</strong></td>
<td><strong>0.1252128014</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with Glorot Uniform weights initializer.

In a similar manner, Figure 5.3 shows the training loss for the four activation functions but with Glorot Initializer with normal weights distribution. Here, in contrast to the 5.1, the training loss for the model with activation function TanH is significantly higher than the rest of the models with three other activation functions. Moreover, except at the very beginning, it remains almost constant over the whole training period.

![Image](image_url)

Figure 5.3: Training loss per iteration for the four models with four different activation function but the same weights initializer: Glorot Normal.

Likewise, the test losses for each of those configurations are presented in the Table 5.3. As seen in the training loss curve 5.3, the MSE and MAE on test dataset for the model with TanH as the activation function is very high compared to the rest of the others. Overall model with LeakyReLU as the activation function has the least losses with Swish and ReLU after it.

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU</td>
<td>0.02863816359</td>
<td>0.1377537869</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td><strong>0.02360761919</strong></td>
<td><strong>0.1273013868</strong></td>
</tr>
<tr>
<td>TanH</td>
<td>0.1799907159</td>
<td>0.386758722</td>
</tr>
<tr>
<td>Swish</td>
<td>0.02681144934</td>
<td>0.1336442441</td>
</tr>
</tbody>
</table>

Table 5.3: Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with Glorot Normal weights initializer.
As seen in the training loss plot for the Glorot Normal initializer 5.3, once again with the He initializer with normal weights distribution, as seen in the Figure 5.4, the model with \( tanH \) as the activation function has higher training loss than the others as seen by the curve steeping upwards over the course of training.

The model with \( TanH \) activation and He Normal initializer has the highest errors on the test dataset as well as listed in the Table 5.4, while the other three activation functions are very close to one another with ReLU having the lowest MSE but Swish having the lowest MAE and second lowest MSE.

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU</td>
<td>0.023088586959911</td>
<td>0.126242226318997</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td>0.0249848659023128</td>
<td>0.124149077718742</td>
</tr>
<tr>
<td>TanH</td>
<td>0.406473271907012</td>
<td>0.613241277985177</td>
</tr>
<tr>
<td>Swish</td>
<td>0.0243918379215931</td>
<td>0.123585399914479</td>
</tr>
</tbody>
</table>

Table 5.4: Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with He Normal weights initializer.

The last remaining configuration in the Table 5.1 is the He Initializer with uniform weights distribution and the four activation functions. Figure 5.5 shows the training loss for this configuration where we see the loss for the model with \( TanH \) represented in color red, has slightly higher losses than the others.
The losses on the test dataset for the four models with He Uniform weights initializer but four different activation functions are listed in the Table 5.5. While Swish has the lowest MSE and MAE among the rest, TanH has the highest MSE similar to the higher training loss seen in the Figure 5.5.

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU</td>
<td>0.0294811856239851</td>
<td>0.141258083890746</td>
</tr>
<tr>
<td>LeakyReLU</td>
<td>0.0276367419021485</td>
<td>0.139207283031772</td>
</tr>
<tr>
<td>TanH</td>
<td>0.0584094313169552</td>
<td>0.168700465141576</td>
</tr>
<tr>
<td>Swish</td>
<td>0.0258350283965637</td>
<td>0.128608841499151</td>
</tr>
</tbody>
</table>

Table 5.5: Hyper-parameter Tuning: MSE and MAE on test dataset for different activation functions with He Uniform weights initializer.

**Discussion**

From our first experiment with various configurations of weights initialization and activation functions, we see that for both Glorot and He weights initialization (both normal and uniform distribution), with TanH as the activation function, the losses are higher than with other activation functions. The training losses are significantly higher particularly for the normal distribution. But given TanH as the activation function, in between Glorot and He, Glorot initialization has the lower losses (considering both MSE and MAE), also with the loss curve being somewhat constant as the training progresses. Whereas in case of HE with TanH, not only the losses are higher, the loss curve is also steeping upwards with successive training iterations. As stated in our hypothesis, Glorot initialization with TanH as the activation function has lower MSE and MAE whereas for the He initialization, ReLU has the consistent lower MSE and MAE. For both initializers with normal or uniform distribution, we use a seed of 42, which can be yet further tuned to see how much the performance can be improved.

The initializer and the activation function combination that yields the lowest MSE and MAE for each of the configuration above is listed in the Table 5.6 to summarize the outcome.

<table>
<thead>
<tr>
<th>Weights Initializer</th>
<th>Activation Function</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glorot Normal</td>
<td>LeakyReLU</td>
<td>0.023607619190121</td>
<td>0.127301386797997</td>
</tr>
<tr>
<td>Glorot Uniform</td>
<td>Swish</td>
<td>0.0238051317146398</td>
<td>0.125212801449652</td>
</tr>
<tr>
<td><strong>HE Normal</strong></td>
<td><strong>ReLU</strong></td>
<td><strong>0.023088586959991</strong></td>
<td><strong>0.126242226318997</strong></td>
</tr>
<tr>
<td>HE Uniform</td>
<td>Swish</td>
<td>0.0258350283965637</td>
<td>0.128608841499151</td>
</tr>
</tbody>
</table>

Table 5.6: Hyper-parameter Tuning: Best MSE and MAE on test dataset for the two weights initializers with Normal and Uniform distribution each with its associated activation function.

From the Table 5.6 above, we can observe that He initialization with normal weights distribution paired with ReLU as the activation function has the lowest MSE among the rest. We also observe that while TanH has the poorest performance of all given the dataset and other hyperparameters, Swish activation has a consistent performance as well. One
reason for that could also be the lower learning rate, with *Swish* favoring lower learning rates. At the same time, squashing activation functions like *TanH* have issues of gradient vanishing and/or exploding gradients for which activation functions like *Swish, ReLU*, derivatives of *ReLU* like *LeakyReLU* helps to prevent.

**Hyper-parameters Tuning - Optimizer and Learning Rate: Setup**

Selecting the best weights initializer and activation function: He Normal and *ReLU* respectively, from our initial hyper-parameters tuning, we move on to *Optimizers* and *Learning Rates*. For this experiment, besides *Adam*, we select three other optimizers based on the gradient descent optimization algorithm: *RMSProp* [GBM+17], *Stochastic Gradient Descent (SGD) with Momentum* [Qia99] and *Nesterov Accelerated Gradient (NAG)* [Nes83].

*Adam* is one of the adaptive learning method which means it computes the adaptive *Learning rates* for each parameter which stores the decaying average of the past gradients and hence is a favorable option to use. *RMSProp* is also based on the adaptive learning method developed in order to resolve the diminishing learning rates that store the decaying average of squared gradients. Similarly, *Stochastic Gradient Descent (SGD) with Momentum* accelerates the gradient descent in the relevant direction as a result avoiding the chances of being stuck in local minima as well as helping in minimizing the oscillations. While *Stochastic Gradient Descent (SGD)* accelerates the descent in the direction of the updated accumulated gradient in a big step, *Nesterov Accelerated Gradient (NAG)* accelerates in the direction of the previous accumulated gradient first and then corrects it after measuring the gradients. We see how each of these optimizers perform in combination with the different learning rates. The setups for the experiments with the various configurations of *Optimizers* and *Learning Rates* is depicted in the Table 5.7.

<table>
<thead>
<tr>
<th>Weights Initializer</th>
<th>Activation Function</th>
<th>Optimizer</th>
<th>Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>HE Initializer (Normal)</td>
<td>ReLU</td>
<td>Adam</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.0001</td>
</tr>
<tr>
<td>HE Initializer (Normal)</td>
<td>ReLU</td>
<td>RMSProp</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.0001</td>
</tr>
<tr>
<td>HE Initializer (Normal)</td>
<td>ReLU</td>
<td>SGD Momentum</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.0001</td>
</tr>
<tr>
<td>HE Initializer (Normal)</td>
<td>ReLU</td>
<td>Nesterov Accelerated Gradient (NAG)</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.001</td>
</tr>
</tbody>
</table>
|                      |                     |                                        | 0.0001        | 0.0001

**Table 5.7**: Hyper-parameter Tuning: Configuration of four optimizers with three different learning rate each, with HE weights initializer and *ReLU* as the activation function.
Hypothesis

We expect that among the four optimizers, Adam converges faster but with poor generalization capability and RMSProp would improve the model’s performance at higher learning rates than SGD with momentum and Nesterov Accelerated Gradient.

This expectation is based on published experiments showing that more general optimizers (i.e., Adam or RMSProp) tend to not under-perform their more specialized cases [CSN+19].

Experiment Results

The training losses for the configurations with Adam as the optimizer with three different learning rates is shown in the Figure 5.6. Given the three models, we see that the model with Learning rate 0.01 has the highest training losses over the training period, which starts at lower value but increases at the beginning and remains constant as the training progresses.

![Figure 5.6](image)

**Figure 5.6:** Training loss per iteration for the configuration with Adam at different learning rates.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>0.01</td>
<td>0.406473271907012</td>
<td>0.613241277985177</td>
</tr>
<tr>
<td>Adam</td>
<td>0.001</td>
<td><strong>0.023088586959911</strong></td>
<td><strong>0.126242226318997</strong></td>
</tr>
<tr>
<td>Adam</td>
<td>0.0001</td>
<td>0.0615345146441487</td>
<td>0.186732403525007</td>
</tr>
</tbody>
</table>

**Table 5.8:** Hyper-parameter Tuning: MSE and MAE on test dataset with Adam as the optimizer with different learning rates.

Similarly, the test losses for the same configuration is presented in the Table 5.8, which is similar to the training losses with the model configured with a higher learning rate (0.01) has the highest MSE and MAE on the test dataset as well. The model with the learning rate 0.001 has the least MSE and MAE among the three.
In case of RMSProp, we can observe from the loss plot 5.7 and the test losses 5.9 that the losses for all models, having three different learning rates are very close to each other despite Learning rate 0.01 having the lowest MSE and MAE.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMSProp</td>
<td>0.01</td>
<td>0.0441051414472302</td>
<td>0.165844546489744</td>
</tr>
<tr>
<td>RMSProp</td>
<td>0.001</td>
<td>0.048241497661214</td>
<td>0.174316157810262</td>
</tr>
<tr>
<td>RMSProp</td>
<td>0.0001</td>
<td>0.0462379770585041</td>
<td>0.16836361170204</td>
</tr>
</tbody>
</table>

Table 5.9: Hyper-parameter Tuning: MSE and MAE on test dataset with RMSProp as optimizer at different learning rates.

Similar to the configurations with Adam as seen in the 5.6, the loss plot 5.8 for SGD with Momentum too has very high losses for the model configured with higher learning rate (0.01) amongst others. The loss increases at the beginning and remains somewhat constant over the course of training.

The losses on the test dataset for the three models with SGD with Momentum as the optimizer with different learning rates is presented in the Table 5.10. Similar to the training losses, the MSE and MAE for learning rate 0.01 is significantly higher than the rest with learning rate 0.001 having the lowest MSE and MAE yet again.
<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD with Momentum</td>
<td>0.01</td>
<td>0.179990715936658</td>
<td>0.386758722014822</td>
</tr>
<tr>
<td>SGD with Momentum</td>
<td>0.001</td>
<td>0.0393745332805587</td>
<td>0.158791222357572</td>
</tr>
<tr>
<td>SGD with Momentum</td>
<td>0.0001</td>
<td>0.0935121426373819</td>
<td>0.253849880031415</td>
</tr>
</tbody>
</table>

Table 5.10: Hyper-parameter Tuning: MSE and MAE on test dataset with SGD with Momentum as optimizer at different learning rates.

Different from the previous loss plots of the optimizers, for Nesterov Accelerated Gradient, lower learning rate (0.0001) produces more loss compared to the others, although not so significant and steeply down over the course of the training.

![Figure 5.9: Training loss per iteration for the configuration with Nesterov Accelerated Gradient with different learning rates.](image)

Table 5.11: Hyper-parameter Tuning: MSE and MAE on test dataset with Nesterov Accelerated Gradient as optimizer with different learning rates.

**Discussion**

From this experimentation with configurations for the four Optimizers and three different learning rates for each, we observe that in case of Adam, the lower learning rates are significantly worst but higher learning rates are not better either. The suitable learning rate here seems to be in between. Likewise, for SGD with Momentum, the higher value for the learning rate produce significant errors during the training as well as on the test dataset, with 0.001 being the favorable value. But in case of Nesterov Accelerated Gradient, the higher learning rates produce less errors on both training and test datasets.
It is important to note here that each of these optimizers can be tuned further by passing their individual hyper-parameters for the tuning. Due to the time and scope limitations, we only use the default parameters for each. Table 5.12 shows the best learning rate for each optimizer on the basis of MSE and MAE on the test datasets.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>0.001</td>
<td>0.023088586959911</td>
<td>0.12624226318997</td>
</tr>
<tr>
<td>RMSProp</td>
<td>0.01</td>
<td>0.04410514414472302</td>
<td>0.16584456489744</td>
</tr>
<tr>
<td>SGD with Momentum</td>
<td>0.001</td>
<td>0.0393745332805587</td>
<td>0.158791222357572</td>
</tr>
<tr>
<td>Nesterov Accelerated Gradient</td>
<td>0.01</td>
<td>0.0418806779068961</td>
<td>0.156743241706912</td>
</tr>
</tbody>
</table>

Table 5.12: Hyper-parameter Tuning: Best MSE and MAE on test dataset for each optimizer with corresponding learning rate.

Hyper-parameters Tuning- Regularization via Dropout: Setup

Like we mentioned before, since the experiments we carried out so far uses the queries from the JOB workload for training and testing. But since the dataset is rather small, we have high chances of the model over-fitting the data. Over-fitting occurs when the model almost "memorizes" the training data, thus yielding very less errors during the training but failing to generalize to the unseen dataset. In order to overcome these issues, various regularization techniques like $L_1$, $L_2$, data augmentation, or dropout, are often used especially when the dataset is very small. In our experiments, we implement $L_2$ regularizer from the very beginning but in this experiment, we also introduce dropouts in the network. The dropout probability ranges from 0.5 to 0.9 meaning drop 50% or 90% of the nodes in the layer respectively with 1 meaning no dropout.

By using Dropouts, we can enable the model to randomly select some nodes and remove them with all of its corresponding incoming and outgoing connections in the network that is linked to those particular nodes. This prevents the model from assigning higher weights to some nodes that it has learned to contribute to the outcome most. Thus it takes away the reliance on some components and in turn prevents over-fitting.

The Table 5.13 shows the different configuration of dropouts we experiment in our model, and (to reduce the amount of pages needed) it also includes the results we achieved in terms of MSE and MAE losses on the test dataset. The values in the Dropout Rate column in the Table 5.13 signifies the percentage of nodes to drop in a given layer of the network. For this experiment, we use the same dropout rates on all layers of the network as seen in the Layers with dropout column in the above mentioned table.

Hypothesis

From our experiments until here, we have seen significant losses during the training where most of the losses almost plateaus but there has not been much improvement on the test losses. Hence, we expect that implementing dropouts on some layers of our model would improve the losses on the test dataset as well.
### Experiment Results

<table>
<thead>
<tr>
<th>Dropout Rate</th>
<th>Layers with dropout</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>All Layers</td>
<td>0.023088586959911</td>
<td>0.1262226318997</td>
</tr>
<tr>
<td>0.9</td>
<td>All Layers</td>
<td>0.0264429440459879</td>
<td>0.126459208341902</td>
</tr>
<tr>
<td>0.8</td>
<td>All Layers</td>
<td>0.0264429440459879</td>
<td>0.126459208341902</td>
</tr>
<tr>
<td>0.65</td>
<td>All Layers</td>
<td>0.0435178174711309</td>
<td>0.163259485649608</td>
</tr>
<tr>
<td>0.5</td>
<td>All Layers</td>
<td>0.033217123044657</td>
<td>0.147312213981658</td>
</tr>
</tbody>
</table>

**Table 5.13:** Hyper-parameter Tuning: MSE and MAE on test dataset at different dropout rates applied on all layers in the network.

We gather the results from this experimentation with introducing dropouts in the network and present them in the Table 5.13. The possibilities are vast so we experiment with only a few dropout rates for this research work but this could definitely be expanded in further studies in future works. From the results in the table, we see that a dropout of 80% on all layers of the network produces the lowest losses on the test dataset in comparisons to other dropout rates but the losses of the model with 80% dropout are not better than the model having no dropouts at all represented with 1.0 as *Dropout Rate*.

In order to see rather than having dropouts on all layers of the network, whether having dropouts on only some specific layers of the network would give us better results, we set up another configuration as seen in the Table 5.14. In this setup, we introduce the *Dropouts* on only the respective layers listed under column *Layers with dropout* rather than on all layers.

<table>
<thead>
<tr>
<th>Dropout Rate</th>
<th>Layers with dropout</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>Layer 1 and 2</td>
<td>0.0236749285033449</td>
<td>0.129370139525991</td>
</tr>
<tr>
<td>0.8</td>
<td>Layer 3 and 4</td>
<td>0.0261968525935961</td>
<td>0.130834971745229</td>
</tr>
<tr>
<td>0.8</td>
<td>Layer 2 and 3</td>
<td>0.0245282173479279</td>
<td>0.13527760030955</td>
</tr>
<tr>
<td>0.8</td>
<td>Layer 2 and 4</td>
<td>0.0254558365568724</td>
<td>0.129989062510348</td>
</tr>
<tr>
<td>0.8</td>
<td>Layer 4</td>
<td>0.0222769736739355</td>
<td>0.123729531963858</td>
</tr>
</tbody>
</table>

**Table 5.14:** Hyper-parameter Tuning: MSE and MAE on test dataset at 0.8 dropout rate applied on selected layers of the network.

From our observation on the obtained results as presented in the Table 5.10, we derive that a dropout of 80% on only the fourth hidden layer of the network, which is the layer right before the last output layer, produces the lowest errors on the test dataset.
Figure 5.10: Training loss per iteration for models configured with 0.8 dropout rate at different layers.

The test losses from models with the various configurations of dropouts are close to one another which can also be observed in the training loss plot depicted in the Figure 5.10.

Discussion

From the results stated in 5.14 and 5.13 we can observe that the models where we introduce the dropouts on all layers of the network however produce more errors on the test dataset while introducing the dropout on only some layers tend to give better results. This could very well be because our data does not have prominent "features" per say. Hence, when we drop the nodes from all layers the model suffers. But dropping only some nodes in one layer or few layers however produces better estimates as the number of nodes dropped is lower than the previous one.

We also mention that our tests are however in no means exhaustive due to the time and scope of this research work as well as the vast possibilities of configurations for these experiments.

Hyper-parameters Tuning- Changing the network width: Setup

The next hyper-parameters we tune is the network width. The width of the network refers to the number of units/nodes/neurons on the hidden layers of the network. In this experiment, we change the width of our neural network model i.e. we add and remove the number of units on the first two hidden layers of our network. The first two layers of our model are responsible to feed into the input and output the transformed input for further computations. The combinations and possibilities are vast so we test with a rather small variations for this research work. We use the variations of 32, 64 and 128 units and test with different combinations of these on the first two hidden layers of the network.

Hypothesis

We naively expect that increasing the units in a layer positively impacts the model output and vice-versa.
### Experiment Results

<table>
<thead>
<tr>
<th>Hidden Layer 1</th>
<th>Hidden Layer 2</th>
<th>Hidden Layer 3 and 4</th>
<th>Mean Squared Error (MSE)</th>
<th>Mean Absolute Error (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>128</td>
<td>512 each</td>
<td>0.0227769726739355</td>
<td>0.1372631963858</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>512 each</td>
<td>0.025557502563552</td>
<td>0.12231834627335</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>512 each</td>
<td>0.0265116608580705</td>
<td>0.140142784712074</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>512 each</td>
<td>0.026036920294765</td>
<td>0.13729068922659</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>512 each</td>
<td>0.0340929854035209</td>
<td>0.148956846510005</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>512 each</td>
<td>0.0301971518567011</td>
<td>0.1369664938552</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>512 each</td>
<td>0.0240153261894744</td>
<td>0.1272051886404</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>512 each</td>
<td>0.026057003944801</td>
<td>0.13653441471539</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>512 each</td>
<td>0.0232132017051568</td>
<td>0.125954441871536</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>512 each</td>
<td>0.023963863238979</td>
<td>0.12881718864654</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>256 each</td>
<td>0.025655003944801</td>
<td>0.13675643642217</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>256 each</td>
<td>0.025210992411627</td>
<td>0.1307087988452</td>
</tr>
</tbody>
</table>

**Table 5.15:** Hyper-parameter Tuning: Changing network width at first two hidden layers that transforms the sets inputs for further computation.

In Table 5.15, the values under the columns *Hidden Layer 1, 2, 3 and 4* respectively represent the number of units in that layer. As mentioned earlier, in this setup we only change the network width for the first two hidden layers *Hidden Layer 1 and 2* and keep *Hidden Layer 3 and 4* constant at 512 units. We can observe from the MSE and MAE losses produced by the models with configurations presented in the table that, 32 and 64 units in the first hidden layer and 128 units in the second hidden layer yields the best MSE and MAE among all other variations. However although the MSE for the configuration with 32 units is less, the MAE is higher for this configuration in comparison to configuration with 64 units.

**Discussion**

Based on the MSE and MAE losses on different configurations as seen in the Table 5.15, we see that our initial setting of 32 units in *Hidden Layer 1* and 128 units in *Hidden Layer 2* remains the best configuration among the others with 64 units and 128 unit in *Hidden Layer 1 and 2* respectively comes closest to it. Although not an exhaustive exploration, our model seems to perform well with 32 or 64 units in the first hidden layer and 128 units on the second hidden layer. We also observe that the model fares well as long as the units in the first hidden layer is closer or lesser than the units in the second hidden layer.

**Hyper-parameters Tuning- Changing the individual units for input sets: Setup**

In order to understand this experiment, it is important to remember our model architecture which is explained in the Chapter 3. Unlike the traditional deep learning models, we do not have a single input and single output model but rather a collection of individual networks (models) for each input sets. The output from these individual networks are then concatenated and passed as a single input to the network that is responsible for predicting the final cardinalities. Although the researches proposing this type of model architecture insists that regardless of the choice of deep learning model, it is important...
that all input sets share the same network architecture. In simple terms, it means that for all our individual input sets: Tables Set, Joins Set, Predicate String Set and Predicate Numeric Set, the number of hidden layers, the number of units in each of these layers, activation functions, optimizers, learning rates and all other hyper parameters remain the same for all of these individual networks. In this experiment, we see if it is really important to have the individual input sets share the same network architecture and parameters or not.

We proceed with this experiment using the heat-maps (included in the Appendix B.), depicting the weights assigned by the model to the respective units in the hidden layers of the individual networks. The weights signifies how much the given unit contributed in the final outcome or in other terms how much important the given unit is in determining the final predictions. Higher the weight higher the importance of the given unit (cell in the heat-map). We add more units to the layer which has higher weights and remove units from the layer with lower weights.

The setup with various configurations we tested alongside the respective results, are listed in the Table 5.16.

### Hypothesis

As per the general consensus, we too expect that changing the individual parameters, in this case the units in each layer, for the model architecture like ours, specifically creating an unbalanced situation (with more nodes for the internal representation of some inputs than for others) would have a negative impact on the model output.

### Experiment Results

The results we obtain from looking up the heatmaps and adjusting (adding/removing) the units in the individual set inputs (individual networks) are listed in the Table 5.16. Here, we see that the individual changes in the individual input sets (network on their own) do not yield better MSE and MAE than the default configuration sharing same parameters of 64 units in the first hidden layer of each networks.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Tables Set</th>
<th>Joins Set</th>
<th>Predicate String</th>
<th>Predicate Numeric</th>
<th>MSE</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>0.022555</td>
<td>0.122351</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>32</td>
<td>128</td>
<td>64</td>
<td>0.026588</td>
<td>0.134463</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>64</td>
<td>0.031960</td>
<td>0.152944</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>64</td>
<td>0.024040</td>
<td>0.130064</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>0.026369</td>
<td>0.135523</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>64</td>
<td>0.026829</td>
<td>0.138249</td>
</tr>
</tbody>
</table>

Table 5.16: Hyper-parameter Tuning: Configuration of individual input sets (networks) with varying units in the first hidden layer of each network.
The training loss for this experiment is shown in the Figure 5.11, where we see that these configurations have somewhat similar losses during training.

**Discussion**

The possible combinations of various configuration is vast and we carry out only a few configurations following the heat-maps of individual input set or network, but it is evident from the losses on the test dataset listed in the Table 5.11 that changing the individual parameters or network structure do not have positive impact in the model performance. The hypothesis we state for this experiment proves to be true, which means given the network architecture like ours, the model performs better if the individual networks that transforms the input, all share the same parameters and network structure.

**5.3 Evaluation Measures for Research Question II**

In our second research question, we use two dataset having embeddings from two different word embedding models, to represent the query predicates on string values: the first dataset uses the embeddings from the default pre-trained fastText model and the second one uses the embeddings from the model that we train on our IMDB database as stated in Chapter 4.1.3. We compare the two models based on the MSE and MAE values on test dataset. We provide side by side comparison of the predictions the two models make on the test dataset and further probe into predictions on individual queries to analyze the models further.

**Hypothesis**

We expect the model trained on the dataset with the embeddings from the model trained on the IMDB database performs better than the model trained on the dataset with the embeddings from the default pre-trained fastText model.
Setups used

We use the model we configured from our experiments mentioned in Section 5.2 and add the embeddings to the dataset. Then we train the model adjusting the width of the first layer and compute the losses to see if these changes have any impact on the model’s outcome. We present the configuration and MSE in the Table 5.17.

<table>
<thead>
<tr>
<th>Loss Function</th>
<th>Model with default fastText embeddings</th>
<th>Model with embeddings trained on IMDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Squared Error (MSE)</td>
<td>0.022552562563552</td>
<td>0.0215927693519236</td>
</tr>
<tr>
<td>Mean Absolute Error (MAE)</td>
<td>0.122351834627335</td>
<td>0.116265659255076</td>
</tr>
</tbody>
</table>

Table 5.17: Word Embeddings: MSE and MAE from models with embeddings from pre-trained fastText model and from the model trained on IMDB database

Experiment Results

The Figure 5.12 shows the training loss for the various configuration we have in the Table 5.17 that shows the training we carried out with the dataset embedded with the default pre-trained fastText model and with the model trained on the IMDB database. We see that the training loss lower with the embeddings trained on the IMDB database than with the default embeddings.

![Loss curve](image)

Figure 5.12: Loss curve for dataset with embeddings from default pre-trained fasttext model and model trained on IMDB database

Figure 5.13 shows the heat-map of the weights distribution at the first hidden layer of the network that feeds the Tables Set inputs consisting of two hidden layers. Here, the x-axis represent the number of units in the layer and the y-axis represents the number of vectors representing the input. The gradient scale on the right side of the figures represents the range of assigned weights where darker hue represents high value of the weights and lower weights are depicted by lighter hues. Each cell in the figure is annotated with its respective weight assigned by the network for that unit in the layer. Heat-maps for other individual networks corresponding to other input sets are included in the Appendix B.
Figure 5.13: Annotated Heat-map of weights distribution at first hidden layer of the network that feeds into Tables Sets as input.

Similarly, the Figure 5.14 shows the predictions from the model with the default fastText embedding on the test JOB dataset and similarly Figure 5.15 shows the predictions made by the model trained on the dataset with embeddings from the word embedding model that is trained on the IMDB database.

Figure 5.14: Prediction on test JOB dataset with embeddings from default pre-trained fasttext model

Figure 5.15: Prediction on test JOB dataset with embeddings from model trained on IMDB database
Discussion

Although the training loss improves with the introduction of embeddings as seen in the loss plot 5.12, the model cannot generalize well to the unseen and complex queries in the test dataset which is evident when we see the individual prediction the models make on the queries in the test dataset as seen in Figure 5.14 and Figure 5.15. We probe further into this by separating and grouping these queries on the basis of the common query template and observe how close or far off the predictions made the two models are.

From here on, in order to avoid confusions, we refer the model trained on the dataset with embeddings from default fastText as default embedding model and for the model trained on the dataset with embeddings from the fastText model trained on the IMDB dataset, we refer it as trained embedding model.

Figure 5.16 shows two sub-figures representing predictions made by the two models on the queries based on the template number 21 of the JOB dataset. We see that the predictions on query numbers 21a and 21c by the trained embedding model are closer to the actual values than the predictions made by the default embedding model. Although in case of query number 21b both models are far from the true value. It is important to mention here that if we look at the actual queries, although all three queries share the same tables, joins, predicate columns, query number 21a and 21c have similar predicate values than 21b. This proves that the model is able to distinguish the estimated cardinality regardless of the same template they share.

![Figure 5.16: Model Predictions on Query Template 21](image)

(a) Predictions by default embedding model    (b) Predictions by trained embedding model

The predictions on query template 25 are somewhat similar for both models where the default embedding model gives a more accurate prediction on 25a compared to the prediction by trained embedding model but the trained embedding model gives a better prediction on 25b in comparison to the default model.
On the contrary, in case of query template number 26, as shown in the Figure 5.18, the default embedding model has closer predictions to the true values with query 26b and 26c having almost accurate predictions whereas the trained embedding model suffers with the query 26b.

Similarly, the trained embedding model has better prediction on query 33b and 33c in comparison to the default embedding model as seen in the Figure 5.19.
In case of some queries as shown in the figures below, both models give bad predictions but both predictions are similar in terms of how they do not distinguish the queries apart from one another given they are all from the same query template.

All queries in the templates 23 and respectively in template 28, have almost exact predictions by the model. When we look into these queries, we see that not only these queries have larger number of joins and predicates, they have same tables, joins and predicates on same attributes. Understandably, this makes it difficult for the models to distinguish the cardinality based on just the tables, joins and predicate sets. Hence, in order for the model to improve the predictions, we have to add more features representative of queries and subtleties like these that differentiates one query from another just based on the predicate values for example samples, some statistics from the database and so on as the additional features. We could also use a larger dataset to see if the model can improve the predictions and generalize further.

![Figure 5.20: Model Predictions on Query Template 23](a) Predictions by default embedding model ![Figure 5.21: Model Predictions on Query Template 28](b) Predictions by trained embedding model

It is important to also find out why the model predicts what it predicts. Model interpretation and explanation functions like \textit{SHAP} \footnote{https://shap.readthedocs.io/en/latest/} helps to break down what is actually happening inside these black box models and figure out which feature contributes most/least. Because of the unconventional model architecture we have, implementing model explainers like \textit{SHAP} is a bit tricky and requires further probing. Nevertheless, we make our
initial attempts at achieving this for model architecture that does not have the conventional 'features' but rather a concatenated transformed vector from individual networks that feeds into the input sets. Features 1-256 correspond to the component of Tables Set, 256 to 512 to that of Joins Set, 512-768 to that of Predicate Numeric Set, 768-1024 to that of Predicate String Set with emebbedings.

Figure 5.22: The SHapley explanation plot for 24 input features

The SHapley explanation presented in the Figure 5.22 shows how much each feature is responsible for the model output (in our case the predicted cardinality). The features colored in red represents the features that push the output value to higher values from the base value whereas the features in blue push the output towards lower values. The base value is calculated as an average of the output produced by the model during training.

On the y-axis we see the features names for example $0.033 = \text{Feature 179}$. Here the input value 0.033 is termed as Feature 179. The input with value 0.033 or the feature...
number 179 is in blue, which means that it pushes the cardinality value to lower values. Likewise, Feature 121 pushes the cardinality value to go higher since it is in red. Since, we have concatenated inputs from individual networks of input sets, the plot here is for one instance (SQL query) only.

The SHAP summary plot in Figure 5.23 provides us the information about the importance of each feature for the model. The plot shows the SHAP values as dotted points for each feature depicting the impact of the feature on the model prediction. Features are positioned on the y-axis while on the x-axis, the SHAP values that represents the impact on the model output are presented. The color represents the value of the feature with colors close to red implies higher values while colors leaning towards blue implies low values.

For example, low values of Feature 263 have higher impact on the cardinality predicted by the model whereas higher values of this feature have low impact on the model predicted output.

Figure 5.23: SHAP Summary Plot for instance number 1.
Similarly Figure 5.24 is a SHAP feature importance plot. Here the features that have the large SHAP values are more important than the ones with low SHAP values.

Figure 5.24: SHAP Feature Importance Plot for instance number 1

Overall a first analysis suggests that tables contribute to move the cardinality towards higher values, whereas join predicates to lower values. Furthermore results show that joins, rather than tables or predicates seem to be the values contributing the most to the predicted cardinalities.

**Hyper-parameter Tuning- Dataset with random generated queries: Setup**

We started our experiments with the JOB workload for both training and testing purpose but as we have seen from the results presented above, although the model is able to minimize the training losses quite significantly, it has difficulty generalizing to queries with same tables, joins, predicate columns but different predicate values. Thus to alleviate this, later generated our training dataset with more than 75 thousand queries to train the model with. But due to the time limitations we could not carry out a complete evaluations on this dataset. Hence, for future work, this dataset could be used to train the model and make an exhaustive evaluation on it.

For this experiment we use this dataset with 75350 queries and split it into train and test dataset. We take 70,000 queries for the training and the rest for the test dataset. We also use the 30% queries from the JOB workload, the same split we use in all our experiments earlier in the evaluations 5.2 and 5.3 as the second test dataset.

We take the best (configurations with the least MSE and MAE) hyper-parameters from from our earlier experiments described in 5.2 and 5.2 and use it as our initial configuration
for the model we train on this randomly generated training dataset. From then on-wards depending upon the MSE and MAE losses we steer the hyper-parameters in the direction we think the model would improve upon. Table 5.18 shows the various configurations we sue on this dataset.

<table>
<thead>
<tr>
<th>Exp. No.</th>
<th>Batch Size</th>
<th>Iteration</th>
<th>Epochs</th>
<th>Layer 1</th>
<th>Weight Init.</th>
<th>Learning Rate</th>
<th>Optimizer</th>
<th>Activation</th>
<th>Dropout</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>3000</td>
<td>4</td>
<td>32</td>
<td>he normal</td>
<td>0.01</td>
<td>Adam</td>
<td>ReLU</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>3000</td>
<td>4</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>Adam</td>
<td>ReLU</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>3000</td>
<td>4</td>
<td>64</td>
<td>he uniform</td>
<td>0.001</td>
<td>Adam</td>
<td>ReLU</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>glorot uniform</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>7000</td>
<td>2</td>
<td>64</td>
<td>glorot uniform</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>128</td>
<td>glorot norm</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>128</td>
<td>glorot norm</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>0.8</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>0.8 on all layers</td>
</tr>
<tr>
<td>9</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>128</td>
<td>he normal</td>
<td>0.001</td>
<td>Adam</td>
<td>LeakyReLU</td>
<td>0.8 on all layers</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>glorot uniform</td>
<td>0.001</td>
<td>RMS</td>
<td>LeLU</td>
<td>0.8 on layer 4 only</td>
</tr>
<tr>
<td>11</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>glorot uniform</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>0.8 on layer 4 only</td>
</tr>
<tr>
<td>13</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>0.8 on layer 4 only</td>
</tr>
<tr>
<td>14</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>glorot uniform</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>Adam</td>
<td>LeakyReLU</td>
<td>0.8 on layer 4 only</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>3000</td>
<td>2</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>0.8 on layer 4 only</td>
</tr>
<tr>
<td>17</td>
<td>1000</td>
<td>5000</td>
<td>4</td>
<td>64</td>
<td>he normal</td>
<td>0.001</td>
<td>RMS</td>
<td>LeakyReLU</td>
<td>0.8 on layer 4 only</td>
</tr>
</tbody>
</table>

Table 5.18: Hyper-parameter Tuning for the dataset with random generated queries.

Hypothesis

We expect the model trained on this workload to have better generalization ability than the model trained on the JOB workload (70% of total queries) when tested on the JOB workload (remaining 30% queries).

Experiment Results

Table 5.25 shows the training losses for all the configurations listed on the Table 5.18. We see the loss curve extend to more iterations for some configurations as we also experimented with different number of training iterations for this workload.

Figure 5.25: Training losses from models configured with varying hyper-parameters on the dataset of randomly generated queries.

For each experiment represented by the number under the column Exp. No. both in the Table 5.18 and Table 5.19, we compute the MSE and MAE on both test workloads that we mention in the setup section. Here by "own split", we refer to the test split we
make on this dataset of 75350 queries. Here we see that Experiment number 1, which is
the only experiment with the batch size of 100, has the worst MSE and MAE on both
test workloads. We also see experiment 13 showing the best generalization to the JOB
benchmark, and experiment 11 showing the best results on the test split. This experiment
verifies that on a more diverse dataset it is possible to achieve better accuracy; it also
serves to verify the generalization ability of the model (although, more a different set of
queries is needed).

<table>
<thead>
<tr>
<th>Exp. No.</th>
<th>MSE on JOB test set</th>
<th>MAE on JOB test set</th>
<th>MSE on own split</th>
<th>MAE on own split</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.40647321907012</td>
<td>0.253910179575756</td>
<td>0.613241279985177</td>
<td>0.419505811473088</td>
</tr>
<tr>
<td>2</td>
<td>0.108563152158644</td>
<td>0.0079789029593957</td>
<td>0.04157251877904</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.12807606200618</td>
<td>0.0052732870578186</td>
<td>0.036324006922378</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.0842772222054093</td>
<td>0.0073920995215899</td>
<td>0.032149274857142</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.126943267073785</td>
<td>0.00461308092252809</td>
<td>0.0351557699390408</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.10613606387551</td>
<td>0.0057902275677603</td>
<td>0.0437013063388266</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.0875157517649184</td>
<td>0.00732345952416984</td>
<td>0.052277079010103</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.10571202545067</td>
<td>0.0058140349303297</td>
<td>0.042052808927923</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.142246754852694</td>
<td>0.00519318070739393</td>
<td>0.041390039750651</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.0976561107087603</td>
<td>0.0044609736341349</td>
<td>0.031479401489057</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0.082262513351143</td>
<td>0.00455981315300182</td>
<td>0.029796075051938</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.082262513351143</td>
<td>0.00455981315300182</td>
<td>0.029796075051938</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0.0813327382684732</td>
<td>0.0051133984217402</td>
<td>0.03822021356815091</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0.100725861366418</td>
<td>0.00609400523470137</td>
<td>0.042309765924343</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0.165177949580307</td>
<td>0.00463506108276513</td>
<td>0.034466560498522</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.0874343070356875</td>
<td>0.00578734042765853</td>
<td>0.04060907969277</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0.0825826397839527</td>
<td>0.00525831863437046</td>
<td>0.0372685869364281</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.19: MSE and MAE losses on the test dataset from JOB workload and MSE and
MAE on the same dataset test split

Comparison against PostgreSQL

To retrieve the PostgreSQL estimates for the JOB queries, we run the EXPLAIN ANALYZE
command. The ANALYZE command makes use of the statistics like the most
common value in a column, distribution of values in the columns taken from table samples.
Although configurable, the default statistics target is 100 rows, which is very less when
tables contain a large number of rows with highly distinct values. PostgreSQL’s optimizer
also does not automatically consider that the multiple columns in the query clauses are
in fact highly correlated. The per column based statistics do not contain this information
and hence it suffers from poor estimates. Although since PostgreSQL V.10, it provides a
manual way to generate multi attribute statistics by using the CREATE STATISTICS
command for columns that are correlated. This extends the default statistics on columns
which in turn provides more statistics for the estimation. For our experiments we did not
alter the configuration from the default settings. Likewise, the estimation suffers when
queries have filters like WHERE, nested queries and especially when there are large num-
bers of joins. In case of JOB, the queries have joins upto 15 as well as the columns in
IMDB database are highly correlated. Thus the independence assumption, poor statistics
contribute to the poor estimations from the PostgreSQL.

In Table 5.20 we present the predicted cardinality values from PostgreSQL and our trained
model on each of the individual queries in the test dataset from JOB and how much they
deviate from the actual values. The PostgreSQL’s optimizer estimates the cardinality to
be only 1 for all queries due to the high number of joins except for the queries 23c and 32b for which it predicts 5 and 10 which are still way off from the true values 628 and 4388 respectively. Compared to these estimations from PostgreSQL, our model predicts more closer value to the true cardinality values.

<table>
<thead>
<tr>
<th>Query Number</th>
<th>True Cardinality</th>
<th>Model Predictions</th>
<th>PostgreSQL Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>21b.sql</td>
<td>38</td>
<td>585</td>
<td>1</td>
</tr>
<tr>
<td>21c.sql</td>
<td>1874</td>
<td>1503</td>
<td>1</td>
</tr>
<tr>
<td>22a.sql</td>
<td>2851</td>
<td>171</td>
<td>1</td>
</tr>
<tr>
<td>22b.sql</td>
<td>31</td>
<td>163</td>
<td>1</td>
</tr>
<tr>
<td>22c.sql</td>
<td>21489</td>
<td>263</td>
<td>1</td>
</tr>
<tr>
<td>22d.sql</td>
<td>46281</td>
<td>1259</td>
<td>1</td>
</tr>
<tr>
<td>23a.sql</td>
<td>618</td>
<td>2409</td>
<td>1</td>
</tr>
<tr>
<td>23b.sql</td>
<td>16</td>
<td>1355</td>
<td>1</td>
</tr>
<tr>
<td>23c.sql</td>
<td>628</td>
<td>929</td>
<td>5</td>
</tr>
<tr>
<td>24a.sql</td>
<td>275</td>
<td>99</td>
<td>1</td>
</tr>
<tr>
<td>24b.sql</td>
<td>45</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>25a.sql</td>
<td>4407</td>
<td>533</td>
<td>1</td>
</tr>
<tr>
<td>25b.sql</td>
<td>6</td>
<td>74</td>
<td>1</td>
</tr>
<tr>
<td>25c.sql</td>
<td>26153</td>
<td>4154</td>
<td>1</td>
</tr>
<tr>
<td>25c.sql</td>
<td>26153</td>
<td>4154</td>
<td>1</td>
</tr>
<tr>
<td>26a.sql</td>
<td>1728</td>
<td>58</td>
<td>1</td>
</tr>
<tr>
<td>26b.sql</td>
<td>93</td>
<td>160</td>
<td>1</td>
</tr>
<tr>
<td>26c.sql</td>
<td>5400</td>
<td>5578</td>
<td>1</td>
</tr>
<tr>
<td>27a.sql</td>
<td>477</td>
<td>1147</td>
<td>1</td>
</tr>
<tr>
<td>27b.sql</td>
<td>247</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>27c.sql</td>
<td>743</td>
<td>1504</td>
<td>1</td>
</tr>
<tr>
<td>28a.sql</td>
<td>4803</td>
<td>302</td>
<td>1</td>
</tr>
<tr>
<td>28b.sql</td>
<td>148</td>
<td>93</td>
<td>1</td>
</tr>
<tr>
<td>28c.sql</td>
<td>8373</td>
<td>237</td>
<td>1</td>
</tr>
<tr>
<td>29a.sql</td>
<td>1620</td>
<td>175</td>
<td>1</td>
</tr>
<tr>
<td>29b.sql</td>
<td>15</td>
<td>81</td>
<td>1</td>
</tr>
<tr>
<td>29c.sql</td>
<td>16308</td>
<td>161</td>
<td>1</td>
</tr>
<tr>
<td>30a.sql</td>
<td>757</td>
<td>6729</td>
<td>1</td>
</tr>
<tr>
<td>30b.sql</td>
<td>28</td>
<td>2067</td>
<td>1</td>
</tr>
<tr>
<td>30c.sql</td>
<td>8024</td>
<td>3208</td>
<td>1</td>
</tr>
<tr>
<td>31a.sql</td>
<td>1273</td>
<td>1840</td>
<td>1</td>
</tr>
<tr>
<td>31b.sql</td>
<td>84</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>31c.sql</td>
<td>2825</td>
<td>3451</td>
<td>1</td>
</tr>
<tr>
<td>32b.sql</td>
<td>4388</td>
<td>1930</td>
<td>10</td>
</tr>
<tr>
<td>33a.sql</td>
<td>8</td>
<td>177</td>
<td>1</td>
</tr>
<tr>
<td>33b.sql</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>33c.sql</td>
<td>114</td>
<td>463</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.20: True cardinalities of JOB test dataset, predictions from trained model and estimations from PostgreSQL
5.4 Evaluation Measures for Research Question III

A natural concern to provide context to our research would be to understand how well our model fares not only when compared to a query optimizer, but also to alternative ML models that are presented with the same data. In this section we consider such concern.

Hypothesis

We expect our model to be better than baseline regressors.

Experiment Results

As mentioned in the previous Chapter 3.5, we experiment with the seven baseline regressors and compute the MSE and RMSE losses on the test dataset from the JOB workload. The queries in the train and test datasets are same as the ones we use in our model above. In Table 5.21 we list out the regressors sorted on the basis of lowest MSE and RMSE to highest.

<table>
<thead>
<tr>
<th>Regressor</th>
<th>Mean squared error (MSE)</th>
<th>Root mean squared error (RMSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lasso Regressor</td>
<td>0.0320821098928755</td>
<td>0.17911495293062</td>
</tr>
<tr>
<td>Support Vector Regressor (SVM)</td>
<td>0.0323541210183722</td>
<td>0.179872513237493</td>
</tr>
<tr>
<td>Random Forest Regressor</td>
<td>0.0371281230314902</td>
<td>0.192686592765273</td>
</tr>
<tr>
<td>Decision Tree Regressor</td>
<td>0.0714226552514874</td>
<td>0.267250173529387</td>
</tr>
<tr>
<td>MLP Regressor</td>
<td>0.0796497844746359</td>
<td>0.282222933998348</td>
</tr>
<tr>
<td>Ridge Regressor</td>
<td>0.0838234232279687</td>
<td>0.28952750795112</td>
</tr>
<tr>
<td>Linear Regressor</td>
<td>0.151635978213431</td>
<td>0.389404645854966</td>
</tr>
</tbody>
</table>

Table 5.21: Mean squared error (MSE) and Root Mean squared error (RMSE) for baseline Regressors sorted on regressor with least MSE to highest.

Discussion

Lasso gives the best MSE and RMSE with Support Vector Regressor coming in the second position. Our deep learning based model is still better when we compare the MSE loss of our model against the MSE of Lasso regressor. But at the same time, it is important to take into consideration that we tuned our deep learning based model but have not tuned any hyper parameters for these regressors. It is quite possible that regressors like Lasso, SVM if tuned properly could compare well against the deep learning based model.

<table>
<thead>
<tr>
<th>Model Version</th>
<th>Model Type</th>
<th>Mean Squared Error (MSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Model with default fastText embeddings</td>
<td>0.0225552502563552</td>
</tr>
<tr>
<td>2</td>
<td>Model with embeddings trained on IMDB</td>
<td>0.0215927693519236</td>
</tr>
</tbody>
</table>

Table 5.22: MSE from the two models we trained explained in 5.2 and 5.3
5.5 Summary

From our experiments we see that although tuning the Hyper-parameters and embeddings from the model trained on the same database impact the overall performance of the model, the predictions are still off by large margins for some queries. But given the small dataset and no other features than the queries themselves, the model would definitely have the tendency to over-fit as we see from our experiments as well.

We find that activation functions, optimizers and learning rates, configured in unison can achieve performance improvements. We are able to identify a proper layer size configuration, and that variable internal representation sizes for the components (i.e., joins, tables, etc.) are not beneficial. When looking at the use of word embeddings to featurize the string predicates present in the queries, we observe that they help the model to learn to distinguish between queries of the same template, with pre-trained embeddings faring slightly worse than embeddings trained over the IMDB dataset. When studying the causes for the predictions we found that features from all components seemed to contribute but join features seemed more highly impacting. Finally when compared to the query optimizer of a database system (PostgreSQL) we found a large difference with respect to the predictions of our model, with the system under-predicting by large margins and our model consistently closer to the mark. Compared with default baseline regressors we were also able to identify that our model performs the best; though naturally this observation needs to be further verified with careful hyper-parameter tuning on the baseline regressors.
6 Related Work

The research works we covered in the survey for SOTA in Join Order Optimization all study the cardinality estimation, cost estimation and query plan space enumeration optimization individually although these three components of the optimizers are highly interdependent upon each other. Taking this factor into consideration that the SOTA can either estimate cardinality or cost or optimize the query plan space, authors Sun et al. recently proposed "an end-to-end cost estimator" [SL19] based on deep learning more specifically tree structured model, to estimate both cardinality and cost. In addition to this, the proposed solution supports predicates with string values as well whereas the SOTA support predicates with numeric values only. The proposed tree structured model learns representations of the query sub-plans as well as representations of predicates with both numeric and string values to estimate cost and cardinality of a given query.

Figure 6.1: Model Architecture. Image sourced from [SL19]

The architecture of the proposed model as seen in the Figure 6.1 consists of three components. The Training Data Generator generates large number of queries based on the join graphs and predicates in the query workload. It also executes the generated queries to get the query plans, real cost and cardinalities of the queries. The second component Feature Extractor is responsible for extracting the features from the query plans which includes query operations used such as nested loop, hash join, sequential scan or index scan and query predicates. These features are all one hot encoded and passed to the Tree Structured Model for further computation. The Tree Structured Model is further composed of three layers:

- Feature Embedding Layer: It is a Fully Connected Network with ReLU as an activation function that consists of encoded one-hot encoded vectors for query operations and query predicates, bitmaps for query metadata and samples. In order to represent the AND semantic in query predicates, it uses min pooling layer to combine the participating predicates whereas for the OR semantic, it uses max pooling layer to represent the participating predicates.

- Representation Layer: The tree structured representation layer as shown in the Figure 6.2, consists of many representation models (nodes) which are all neural networks sharing same structure and parameters. The layer trains representation of query sub plans recursively starting from the (bottom) leaf nodes to (up) root node so that it can capture the correlation among nodes. Each representation model
(node) receives three input: the embedding vector from the feature embedding layer, representation vector of the left child nodes and representation vector of its right child node. The representation vectors of the child nodes for the leaf node is 0.

It uses LSTM neural networks in this layer in order to avoid gradient vanishing and explosion problem.

- Estimation Layer: This the last layer that estimates the cost and cardinality of the queries. It uses Fully Connected Network with ReLU as activation function and Sigmoid as activation function for the output layer that estimates the cardinality and cost. Both the cardinality and cost estimation share the same common embedding layer and representation layer.

In order to get the vectors for the query predicates with string values, the authors employ a specific rule to extract all the strings and sub strings in query workload and train a word2vec model with skip-gram method to learn representation of these strings with sub-word information. The mappings of these strings to their respective vector representation are stored in tree indexes where the sub-strings extracted by the prefix function is stored in the prefix tree index and the sub-strings extracted by the suffix function in the suffix tree index. Figure 6.3 shows an example of the predicate (LIKE 'Dino%') where it looks up the prefix tree index and uses the vector representation of Din for the given predicate.

Figure 6.2: Representation Layer. Image sourced from [SL19]

Figure 6.3: A prefix tree for a query predicate LIKE 'Dino%’. Image sourced from [SL19]
The rule used to extract the string and sub-string information is expressed as $rule = \langle P, F, L \rangle$ composed of three parts:

- **Pattern matching ($P$):** It matches the sub-strings based on uppercase, lowercase, white-space, numerical values and exact matches.
- **String Function ($F$):** The function extracts the prefix and suffix of the string.
- **Size ($L$):** This parameter defines the length of the sub-string to extract.

By presenting a specific design that addresses the prefix matches found in common SQL queries, we believe that the work of Sun et al., represents the closest related work to our approach in supporting string predicates for cardinality estimation with ML models.
7 Conclusion and Future Work

In this chapter we summarize the main contributions of our research. We also discuss some possible improvements to the presented evaluation setup and outline promising directions for future research.

7.1 Conclusion

Addressing our Research Question 1, we experimented with various hyper parameters of a deep learning model in our cardinality estimation model to study their impact on the model’s overall performance. Due to the vast options and combinations of hyper-parameters, a separate study dedicated entirely on this subject matter would yield more conclusive results compared to our limited choice of selections.

From our study we see that changing just one parameter can often lead to entirely different outcome and thus, it becomes much important for research works to specify the details about these configurations for others to take the research forward. Since our work was limited due to the lack of a large training dataset, adding more features or a large training dataset possibly with other additional data profiles would help to produce more insightful results.

Overall we find that configurations like using the Adam optimizer, and applying selective dropout can improve the model. We also find some relations between activation function and learning rate. By looking at the learned weights of the trained model, we observe that all components affect the output, without one component standing out. We also observe that on a query level, the presence or not of a specific join or table, can have a large role on the predictions (leading to a corresponding high weight). For the case of normalized values we find that the value itself does not create noticeably high weights, and that the operator and column also play a role in the cardinality estimation. Using SHAP values we also identify that joins are a component playing a large role in determining the cardinality.

From our second research question, we observe that the features captured by the word embeddings improve the average errors made by the model and we also find that embeddings from the model trained on the IMDB database improved the model to some extent if not significantly than with the embeddings from the pre-trained model. Although the cardinality estimations by the model are not perfect but we see that they are better by orders of magnitude than the estimations from PostgreSQL’s optimizer. We also find that when compared with baseline regressors our model behaves consistently better.
In our research we also validate on a synthetic query dataset, developed over the IMDB data, that the model is also able to achieve high accuracy after training, with a similar set of hyper-parameter configurations as the one established for the JOB benchmark. We also see that the agent is able to generalize between the training synthetic dataset and the JOB benchmark, but that in this regard there is still room for improvements.

To conclude, assuming a fixed database, given a large dataset and additional data profiles, we are optimistic that a deep learning based model can achieve close to exact estimations with a set of tuned hyper parameters and succinctly featurized queries. Although gathering the dataset, pre-processing the queries and tuning the model takes time, the predictions of the trained model are if not exact, better than the underestimations made by the PostgreSQL’s optimizer and can outperform more traditional regressors.

7.2 Future Work

Continuing our research, further works that explore into the configuration of wide range of hyper-parameters would benefit the research community as a whole, with published results by not having to re-invent the wheel for the same task, helping the maintenance and adoption of models. Alternatively, AutoML solutions to help the tuning of the process seem of interest.

Similarly, further studies can be conducted to succinctly and effectively featurize the SQL queries and more specifically alternatives to using embeddings for string values in the query predicates. More specifically, alternatives to properly capture prefix match queries, instead of embeddings only, could be of great value.

Additional features, such as samples, histograms, could be of interest, perhaps helping the models to adapt to changing data.

To make most out of JOB, something similar like dbgen and qgen provided by TPC benchmarks would greatly help to overcome the cold start problem of machine learning solutions. Training datasets would also be of use.

Additional deep learning models, such as transformers, graph, or memory-based solutions, also deserve consideration for their potential role of understanding context of expressions, structure, and correlations of items in a query. These are details that seem to be missing from the model of deep sets. The robustness of models to distributional shifts is also an important area of study.

A natural consequence of the increasing improvements of this kind of model is its necessary integration into real world database systems, such that these models become slowly a new standard for data management.
Bibliography


Appendices

A. Training loss graphs with chart scaling

Training loss per iteration for the four models with four different activation function but the same weights initializer: Glorot Uniform.

Training loss per iteration for the four models with four different activation function but the same weights initializer: Glorot Normal.

Training loss per iteration for the four models with four different activation function but the same weights initializer: HE Normal.
Training loss per iteration for the configuration with HE Uniform.

Training loss per iteration for the configuration with Adam at different learning rates.

Training loss per iteration for the configuration with RMSProp at different learning rates.

Training loss per iteration for the configuration with SGD with Momentum with different learning rates.
Training loss per iteration for the configuration with Nesterov Accelerated Gradient with different learning rates.

Training loss per iteration for models configured with 0.8 dropout rate at different layers.

Hyper-parameter Tuning: Configuration of individual input sets (networks) with varying units in the first hidden layer of each network.
B. Heatmap of weights distribution at each hidden layers.

Heatmap of weights distribution at first hidden layers for Tables Input Set.

Heatmap of weights distribution at second hidden layers for Tables Input Set.
Heatmap of weights distribution at first hidden layer for Joins Input Set

Heatmap of weights distribution at second hidden layer for Joins Input Set
Heatmap of weights distribution at the first two hidden layers for Predicate with String Input Set
Heatmap of weights distribution at first hidden layer for Predicate with Numeric Input Set

Heatmap of weights distribution at second hidden layer for Predicate with Numeric Input Set
Heatmap of weights distribution at the third and fourth hidden layer