A Neural Network for Cardinality Estimation

by

Xingchi Wang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

ABSTRACT

With the booming development of the Internet, the amount of data stored in databases are enormously growing. Databases retrieve relevant information in response to users' queries; the retrieved information is encoded in dynamically generated by databases in the form of structured data records. As the amount of data increases, the query time becomes longer and longer. So, there is an urgent need for database optimization.

Database optimization is the strategy of reducing database system response time. Databases provide us with information stored with a hierarchical and related structure, which allows us to extract the content and arrange it easily. Database optimization includes avoiding unused tables, using proper indexing, avoiding temporary tables and coding loops and so on. In our research, we choose to optimize cardinality estimation in database optimizer.

Cardinality estimation is a fundamental task in database query processing and optimization. However, the accuracy of traditional estimation techniques is poor resulting in non-efficient query execution plans. With the rise of deep learning, there is a general notion that data representation can lead to better estimation accuracy. Up to now, all proposed neural network approaches for cardinality estimation can only deal with inner joins between tables. To overcome this issue, we introduce a novel neural network (NN) in this paper. Through systematic experiments and scientific analysis results, it is proved that our model performs better than other models. This approach leads to better data representation and thus better estimation accuracy in multiple types of joins.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Dr. James Miller, for his guidance and patience throughout my master program. He taught me not only the way to conduct quality research but also professional skills useful for my future career.

I also would like to express my sincere thanks to my family, my friends, as well as my colleagues in Dr. James's research group.

TABLE OF CONTENTS

ABSTRACT ..... **ii**

ACKNOWLEDGMENTS ..... **iii**

TABLE OF CONTENTS ..... **iv**

LIST OF FIGURES ..... **vii**

CHAPTER 1 INTRODUCTION **1**

1.1 Research Background 1

1.2 Literature Review of Existing Models 2

1.3 Problem Statement 3

1.4 Research Objectives 4

1.5 Thesis Structure 4

CHAPTER 2 METHODOLOGY **6**

2.1 Deep Learning and Neural Network 6

2.2 Neural Network (NN) 8

2.3 Vectorization And One-hot Encoding 9

2.4 The Structure Of NN 11

CHAPTER 3 EXPERIMENT AND RESULTS **13**

3.1 Generating training data for IMDb 13

3.2 Generating training data for Pagila 14

3.3 Estimation accuracy 15

3.3.1 Estimation accuracy for IMDb 15

3.3.2 Estimation accuracy for Pagila 16

3.4 Statistical Significant Testing And Effect Size 17

3.5 Hyperparameter Tuning 20

3.6 Runtime Performance 22

CHAPTER 4 CONCLUSIONS AND FUTURE WORK 23

4.1 Future Work 23

4.2 Conclusions 24

BIBLIOGRAPHY 25

APPENDIX 28

Appendix A. The core code of the experiment. 28

LIST OF TABLES

Table 1 Detail information about tables in IMDb. 14

Table 2 Detail information about tables in Pagila. 14

Table 3 Experiment result comparison. 16

Table 4 Experiment result comparison. 17

Table 5 Mann–Whitney U test experiments results for IMDb. 18

Table 6 Effect Size results for IMDb. 19

Table 7 Mann–Whitney U test experiments results for Pagila. 20

Table 8 Effect Size results for Pagila. 20

Table 9 Runtime performance comparison. 22

LIST OF FIGURES

Figure 1 The structure of fully connected neural network. 8

Figure 2 Numerical query representation. 11

Figure 3 The architecture of Neural network (NN). 12

Figure 4 Different cardinality estimators evaluation results. 16

Figure 5 Different cardinality estimators evaluation results. 17

Figure 6 mean q-error with the number of epochs. 21

CHAPTER 1 INTRODUCTION

1.1 Research Background

Query optimization is based upon cardinality estimation. To be able to choose the optimal execution plan, the query optimizer needs to have good estimates for result sizes. Due to increasing data sizes, query optimization becomes a more difficult challenge. Most query optimization techniques are cost-based[2,14], where cardinality estimation plays a dominant role in the approach to approximate the number of returned tuples for every query within a query execution plan. These estimations are used in various optimization techniques. For this reason, it is essential to improve the accuracy of cardinality estimation.

Methods to improve the accuracy for cardinality estimation is an open area of research. Most traditional estimation approaches based on statistical models are not accurate enough. The reason is that they make unreasonable assumptions, which are data independent. Nowadays, we know that data carries important hidden information. So traditional approaches will lead to erroneous cardinality estimation.

A possible way to break the limitations is to use machine learning, especially neural networks, for cardinality estimation. With the rise of machine learning in many different fields, database researchers also have started to apply machine learning to cardinality estimation[7,8,9,11,13,16,19].

We argue that deep learning is a highly promising technique for solving the cardinality estimation problem. Deep learning can model complex data dependencies and correlations. Neural networks

have powerful representation learning capabilities. Estimation can be considered as a supervised learning problem. The input to neural network contains tables, query features and join types, the output of the neural network is an estimated cardinality. Under these circumstances, there are several groups of researchers who already combined neural networks with cardinality estimation; however, they neglect to consider the actual query situation of the database. In other words, their models can only deal with inner joins, ignoring left and right joins.

In this paper, we propose a neural network (NN) for multiple types of joins including inner join, left join and right join. NN addresses the aforementioned weak spot of the single join problem. Thus, even for large databases, our model is efficient because NN does not waste any capacity for memorizing. Combined with deep learning, NN has good performance with multiple types of joins. We evaluate our model using the IMDB and the Pagila datasets, which show that NN is more robust than other approaches in multiple types of joins. The comparison with other approaches is promising.

1.2 Literature Review of Existing Models

Traditional approaches of cardinality estimation in relational database management system rely on statistics that include histograms on each column in a table[4,5,6], min and max values[18], a list of most frequent values and their frequencies and the number of distinct values. However, with the development of data analysis, data independence has been shown to be a major problem for these statistical approaches, which leads to a non-efficient query execution plan.

To solve this issue, a promising way is to use deep learning for cardinality estimation. Twenty years ago, neural networks had already been used for cardinality estimation[10]. Also, people had already combined machine learning with cardinality estimation. Regression-based models have

used for cardinality estimation[1]. A semi-automatic alternative for explicit machine learning was presented in paper[12], where the feature space is partitioned using decision trees, and for each split a different regression model is learned. For a more sophisticated formulation way, reinforcement learning has been applied to query optimization in three articles[9,13,17]. Another two articles treat cardinality estimation as a supervised learning problem[7,21]. Woltmann[21] introduces a local neural network concentrating on sub-part of the whole schema, which makes the query sampling less sparse and makes the smaller neural network structure. A. Kipf et al.[7] introduce an approach called multi-set constitutional neural network (MSCN) which is a set-based neural network. MSCN uses bitmaps which are derived from samples given the truth values of the query's predicates. MSCN is capable of modeling joins and predicates over several tables and thus can cover correlations in the data. MSCN depends on large bitmaps to cover the whole schema in order to find query execution plans. If a query without bitmaps is passed to the neural network, the network will return an erroneous estimation. Additionally, the main assumption in these two works is that there are only inner joins in any database. However, in practice, this hardly happens. If a query with left or right joins is passed to their models, the networks' capability fails, and the estimate is erroneous. Their neural networks only work on the inner join situation.

1.3 Problem Statement

The discussion above reveals that the previous studies on improve the accuracy of cardinality estimation pay more attention to statistics that include histograms on each column in a table and pay less attention to the schema of the database. Whereas the schema of the database is essential for cardinality estimation. The existing models ignore the relationship between cardinality estimation and the schema of the database. So existing models maybe perform well on inner join situation, they perform bad on multiple type situations such as left join and right join situations.

Our approach overcomes this issue, NN is a neural network which can handle sophisticated multiple types of joins situations and hence derives better estimates.

1.4 Research Objectives

The main objective of this research is to improve the accuracy of cardinality estimation. The detailed objectives are as follows:

(1) To conduct a thorough literature review to screen the most promising models that can well applying deep learning to the cardinality estimation task;

(2) To overcome the problem that neural networks can only take numerical values as input vectors, we use vectorization and one-hot encoding to transform their string representations to numerical vectors;

(3) To propose a new model based on vectorization and one-hot encoding calculated by the neural network to yield more accurate cardinality estimation.

1.5 Thesis Structure

(1) **Chapter 1** presents research background, literature review, problem statement, research objectives, and thesis structure.

(2) **Chapter 2** introduces the methodology employed in this thesis, including the introduction of neural network, vectorization and one-hot encoding for neural network, and the new model proposed in this study.

(3) **Chapter 3** demonstrates the data set we use, the way we generate training data and validation data, hyperparameter tuning and runtime performance. This chapter also introduces the way we use to judge if our model is the most suitable model to solve cardinality estimation problem including statistical significant testing and effect size.

(4) **Chapter 4** summarizes the conclusions obtained in this study and the recommendations for future work.

CHAPTER 2 METHODOLOGY

2.1 Deep Learning and Neural Network

Deep learning is a subset of machine learning where neural networks learn from large amounts of data. The algorithms of neural networks are inspired by the human brain. Deep learning algorithms perform a task repeatedly and gradually improve the outcome through deep layers that enable progressive learning. Deep learning is making a big impact across industries. In life sciences, deep learning can be used for advanced image analysis, research, drug discover, prediction of health problems and disease symptoms, and the acceleration of insights from genomic sequencing. In transportation, it can help autonomous vehicles adapt to changing conditions. It is also used to protect critical infrastructure and speed response.

Deep learning methods is based on artificial neural networks with representation learning. The architectures of deep learning such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, machine vision, speech recognition, natural language processing, audio recognition and so on.

The artificial neural networks have good representation capabilities, which can learn the hidden information between data. For example, applying convolutional neural networks (CNN) to computer vision, by using convolution and pooling, CNN can learn the connection between pixels.

In order to improve the accuracy of cardinality estimation, we choose artificial neural network as our model. Our model is fully connected deep network. Fully connected networks are the

workhorses of deep learning, used for thousands of applications. The major advantage of fully connected networks is that they are “structure agnostic”. That is, no special assumptions need to be made about the input (for example, that the input consists of images or videos). We cannot know the hidden information between fields in each table, and the hidden information between tables. In this way, fully connected networks are applicable for handling cardinality estimation problem. The structure of fully connected neural network are shown in Figure 1. A fully connected neural network consists of a series of fully connected layers. A fully connected layer is a function from R_m to R_n. Each output dimension depends on each input dimension.

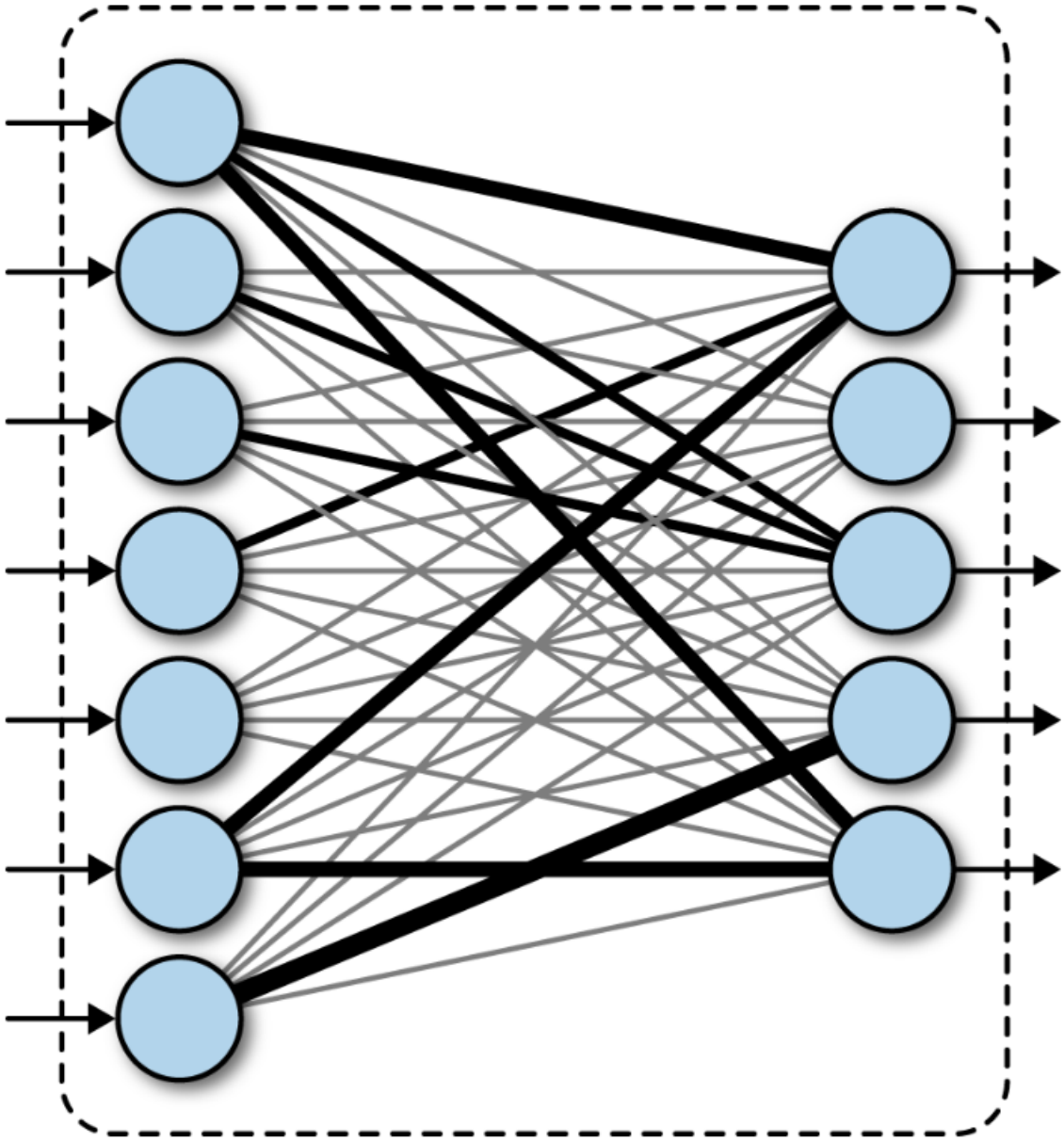

Figure 1 The structure of fully connected neural network.

2.2 Neural Network (NN)

A typical machine learning algorithm, like regression or classification, is designed for fixed dimensional data instances. Their extensions to handle the case when the inputs or outputs are permutation invariant sets rather than fixed dimensional vectors is not trivial. However, for our

task that improve the accuracy of cardinality estimation, just using vectors are not suitable. Vectors have orders, they carry hidden information through the orders. We plan to learn the correlations between tables in the datasets, the order of tables and the fields is meaningless. That means, the order carries no information we need.

For our neural network, we want the input instances are four parts rather than traditional vectors. Similar to fixed dimensional data instances, we use two learning paradigms in case of parts. In supervised learning, we have an output label for a part that is invariant or equivariant to the permutation of elements. We use the queries of the datasets for training and testing, the order of the queries carries nothing information. Therefore, we use four parts as our input.

From a high-level perspective, applying deep learning to the cardinality estimation task is straightforward. Placing the features of queries including tables, joins and columns etc. into the neural network, and then deriving the estimate as output later. After training a model, the model can be a query optimizer for other queries. Deep learning algorithms indeed enhance the accuracy of the cardinality estimation task compared to traditional approaches.

Before discussing NN directly, we introduce the techniques which do featurization for queries first including vectorization and one-hot encoding for queries. After that, we introduce the key idea of NN in Section 2.4.

2.3 Vectorization And One-hot Encoding

Neural networks can only take numerical values as input vectors. However, in real world databases, there are always string values instead of numerical values. Many deep learning algorithms cannot operate on string data directly. To overcome this problem, we use *Vectorization* and *one-hot*

encoding to transform the four parts (T_q, J_q, P_q, TJ_q) from their string representations to numerical vectors.

Each table $t \in T$ is represented by a unique one-hot encoding vector *v*. Since Table part *T* is unique which does not contain any duplicate table, every vector *v* is unique. This ensures that the neural network can learn the global schema. Similarly, each join $j \in J$ transforms to a unique one-hot encoding. Every vector of join is also unique. For predicate part *P*, the process becomes a little bit complex. Each predicate is an expression instead of a string. The form of *P* is (*column, operator, value*). We featurize *column* by a unique one-hot encoding. With three singular operators $>$, $<$, $=$, we need a vector of length three to model all possible operators. The presence of an operator dictates a 1 at the corresponding position in the vector. For example, $>$ generates (1,0,0) and $\geq$ generates (1,1,0). Finally, the *value* v_i is already a number. However, neural networks are usually used with min-max-normalized input vectors ranging from 0 to 1 to enhance their accuracy. In order to get the best performance of NN, we normalized v_i to [0,1] as shown in **Equation (1)**.

$$v_i = \frac{v_i - \min(p)}{\max(p) - \min(p)} \quad (1)$$

Where *min(p)* is the minimum boundary of the range of predicate *p* and *max(p)* is the maximum boundary. These can be obtained directly from the database.

The last part is encoding the types of join part *TJ*. NN supports the recognition of three types of joins, left, inner and right join. Thus, we also need a vector of length three to model all possible types of joins. The final vector is shown in **Figure 2**.

SELECT FROM title t LEFT JOIN cast_info ci ON t.id=ci.movie_id WHERE ci.person_id>284738;

Table part{[0 1 0 1...],[0 1 0 ...]} Join part{[0 1 0 0]} Predicate part {[0 0 1 0 1 0 0 0.56]}Types of join part {[1 0 0]}

Figure 2 Numerical query representation.

2.4 The Structure Of NN

Standard neural network architectures are not suitable for this type of data structure. Convolutional neural networks (CNNs) are applicable to images, recurrent neural networks (RNNs) are applicable to time-series data. However, we require multi-layer perceptrons (MLPs) with serialization to deal with queries. Multi-layer perceptions are defined by three types of layers: input, hidden and output layers. Each neuron is connected to all the neurons in both the previous and following layer. Thus, they are also called a fully connected layer.

We use MLPs to simulate the global schema. The structure of NN is shown in **Figure 3**. We generate queries from the datasets. The queries we generated is based on the frequency of the tables and the columns. Simultaneously, we would generate the queries based on the whole schema. The two types of queries will help the neural network learn the globe structure which is the whole schema and the local structure which is the local schema.

We get tables, joins, conditions and true cardinalities from the database. And then using vectorization and one-hot encoding to help queries generates its numerical representations. Our query representation consists of a collection of multiple parts, which including Tables part, Joins part, Predicates part and Types of join part. we merge the individual part representations by concatenation and then put them into a neural network. The hidden layers of NN are three fully connected layers, which use ReLU activation functions. For the output layer of NN, we use a sigmoid activation function, and the output is a scalar which between 0 and 1. Based on the normalization of the predicate value by using the minimum and maximum values.

We train our model to minimize the mean *q-error*[15]. This article demonstrates that *q-error* is more suitable for judging the accuracy of the estimates compared to mean-squared error. The smaller the q-error is, the better the neural network trained.

I use PyCharm to experiment. And using Torch to build the network. The core code of the experiment is detailed in **Appendix A**. It only shows the code of the main steps in the experiment, including preprocessing, training, and model structure. Does not include data set extraction and other complex tasks.

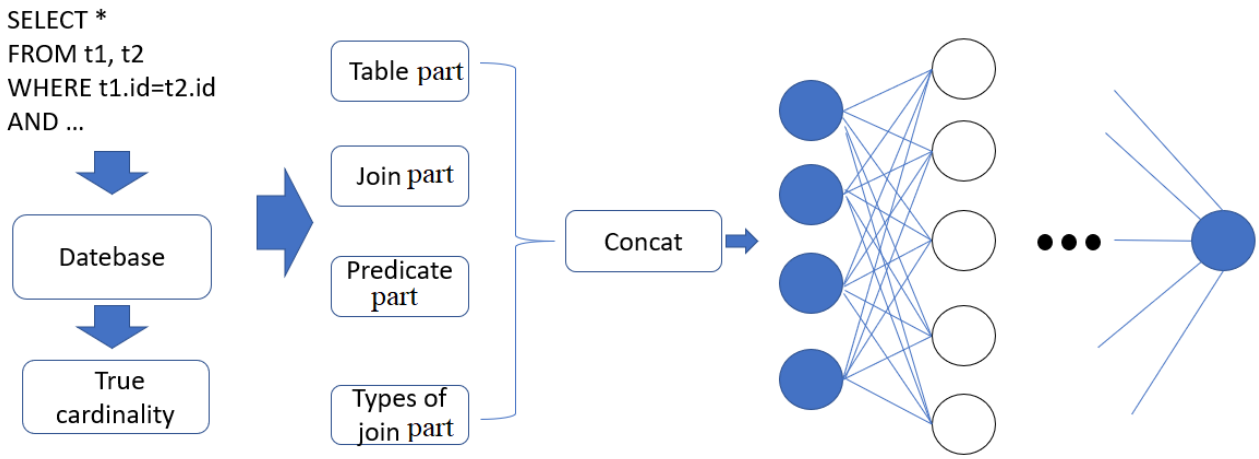

Figure 3 The architecture of Neural network (NN).

CHAPTER 3 EXPERIMENT AND RESULTS

We conduct an experimental study to evaluate the performance of our approach. We begin by the generating a training data (Section 3.1 and Section 3.2), including details about the data sets. In Section 3.3, we present details of the experiment results. In Section 3.4, we present the statistical significant testing and effect size for Section 3.3. In Section 3.5, we describe hyperparameter tuning. Finally, the runtime performance is shown in Section 3.6.

3.1 Generating training data for IMDb

We use the IMDb data set as our first experiment to evaluate our approach. IMDb is a well-known online Internet Movie Database, which captures approximately 6.5 million titles (including episodes) and 10.4 million personalities in its database. Therefore, IMDb contains various complex correlations between tables, and it is challenging for cardinality estimation.

We choose title, movie_info, movie_companies, movie_keyword and cast_info as our main tables. We focus on the complex correlations among them. For example, table title contains more than 4 million distinct values, table movie_companies contain nearly 3 million distinct values. More details are listed on **Table 1**. All properties are chosen in order to ensure comparability with other approaches. We select all columns from each join which can be represented as integers.

We generate a training set and a test set which focuses on the five tables. Both sets contain complex multiple types of join including left, right and inner joins. All queries in the training set and test set are two table or three table joins.

Left, inner and right joins each account for 1/3 both in the training and test set. Our model is different from MSCN[7] and local deep learning models[20], NN do not require bitmap sampling to improve its accuracy. Therefore, NN uses less memory in a shorter training time. This is another reason why NN can learn more complex database schemas.

Table 1 Detail information about tables in IMDb.

table	***column***	***min value***	***max value***	***number of distinct values***
title	kind_id	1	8	8
title	production_year	1874	2018	144
movie_info	movie_id	1	4730460	4519477
movie_info	info_type_id	1	98	61
movie_companies	company_id	1	319060	319060
movie_keyword	keyword_id	1	236627	236627
cast_info	person_id	1	6128554	6122920
cast_info	role_id	1	11	10

3.2 Generating training data for Pagila

We use Pagila as our second data set to evaluate our approach. Pagila is a sample database from PostgreSQL to show the basic functions of the database. Pagila is the DVD rental database, which contains 15 tables and over 30,000 pieces of data.

We choose rental, payment, custom, inventory, film as our tables. More details are shown in **Table 2**. In order for our model to learn complex correlations among them, we also generate a training set and a test set as before. All queries in the training set and test set are two table or three table joins. Again, left, inner and right joins each account for 1/3 both in the training set and test set.

Table 2 Detail information about tables in Pagila.

table	***coulumn***	***min value***	***max value***	***number of distinct values***
rental	rental_id	1	16049	16049
payment	id	1	16049	16049
payment	amount	0.00	11.99	12
custom	custom_id	1	599	599

inventory	inventory_id	1	4581	4581
film	length	1	4581	4581

3.3 Estimation accuracy

3.3.1 Estimation accuracy for IMDb

To evaluate the estimation accuracy of our model, we compare our approach with two other techniques. First, we select a traditional estimate approach which does not use machine learning. PostgreSQL use traditional estimations based on histograms on each column in a table and a list of most frequent values and their frequencies. The other approach is multi-set convolutional neural network (MSCN) which is a state-of-the-art approach using deep learning for cardinality estimation. In order to compare with MSCN, we use the same data set, and the queries are generated in the same tables. MSCN uses samples to help their model learn mappings and thus improve accuracy.

Figure 4 and **Table 3** detail the experiment results for the three different approaches. In the figure, the q-error of each estimate is scaled to a log-space on the y-axis. The underestimates give negative values and overestimates give positive values. The line inside the box is the median q-error. **Table 3** shows detailed q-errors of each model including median q-error, 90th q-error, 95th q-error, 99th q-error, max q-error and mean q-error on the same test set.

From **Figure 4** and **Table 3**, it suggests that our model is more accurate than PostgreSQL and MSCN in multiple types of join situations.

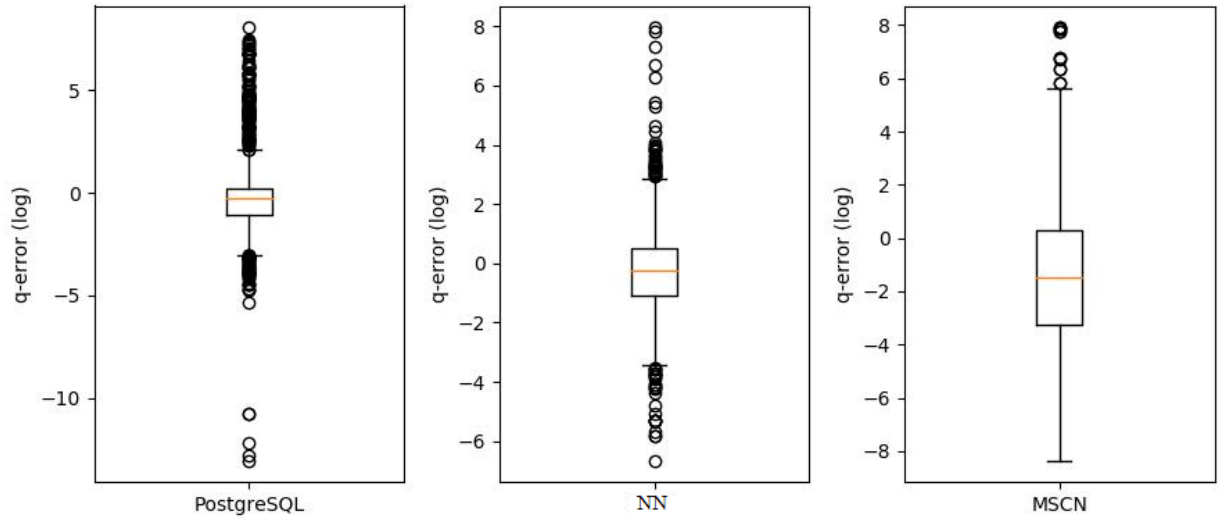

Figure 4 Different cardinality estimators evaluation results.

Table 3 Experiment result comparison.

model	median q-error	*90th q-error*	*95th q-error*	*99th q-error*	*max q-error*	*mean q-error*
PostgreSQL	2.16	43.34	112	1131.21	470132	976.07
MSCN	9.73	188.79	376.55	2400	4267.63	106.46
NN	2.29	12.25	26.47	201.36	2868	14.83

3.3.2 Estimation accuracy for Pagila

For the second experiment, our goal is to compare NN with traditional estimate approaches in PostgreSQL. Traditional estimate approaches based on histograms on each column in a table and a list of most frequent values and their frequencies to derive estimate. Pagila is a small data set compared to IMDb, it only contains over 30,000 pieces of data. Traditional approaches may perform well for small data sets. **Figure 5** and **Table 4** detail the experiment results for the two approaches. The MSCN open-source version is only a partial implementation of the article, hence it cannot be applied to Pagila.

From **Table 4**, we can see that NN performs better than traditional approaches in PostgreSQL. And from **Figure 5**, we can see that the estimates derived from NN are more accurate than

PostgreSQL. Most estimates are really near the true cardinality values. From the second experiment, we can draw a conclusion that NN performs better than PostgreSQL in multiple types of joins situations.

Table 4 Experiment result comparison.

model	median q-error	*90th q-error*	*95th q-error*	*99th q-error*	*max q-error*	*mean q-error*
NN	1.04137	6.42184	7.0	174.420	438.0	6.39075
PostgreSQL	2.36820	16.01582	72.99068	2726.760	9719.0	97.77848

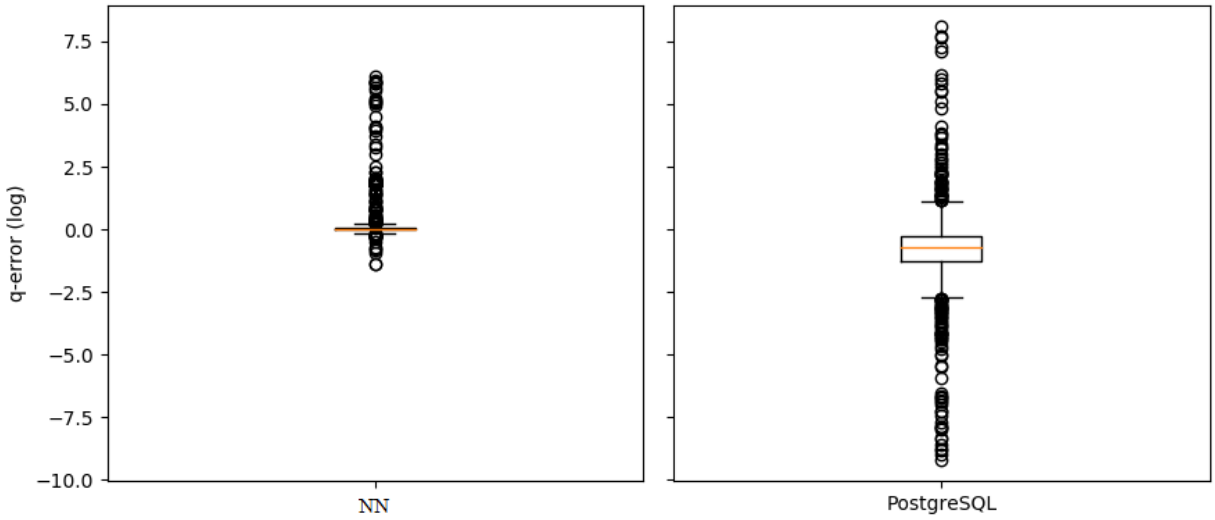

Figure 5 Different cardinality estimators evaluation results.

3.4 Statistical Significant Testing And Effect Size

3.4.1 Statistical Significant Testing And Effect Size for IMDb

Judging the quality of a model cannot be done by only using the mean q-error. We need more scientific methods to evaluate if NN is truly better than the other two models. We use the Mann–Whitney U test (MWW) and an Effect Size estimation[3] (Cliff's Delta) to determine whether our approach is statistically significantly more accurate than others. The Mann–Whitney U test is a

nonparametric test that can be used to determine whether two independent samples are selected from populations having the same distribution. If population 1 and 2 are unequal, then at any fixed α level of significance the probability of the MWW test being significant at the α level tends to 1 as the m, n sample size values tend to infinity. By contrast, if populations 1 and 2 are equal, then the probability of the MWW test being significant at the α level will not tend to 1[20].

The null hypothesis for $\alpha = 0.05$ is that the prediction data of NN, MSCN and PostgreSQL are from the same distribution. We make three comparing experiments here: the first experiment is NN VS MSCN, the second is NN VS PostgreSQL and the last one is MSCN VS PostgreSQL. The results are shown in **Table 5**.

Table 5 Mann–Whitney U test experiments results for IMDb.

Competing models	**statistic**	***p-value***
NN VS MSCN	551566.5	1.6541e-23
NN VS PostgreSQL	706647.0	2.1574e-07
MSCN VS PostgreSQL	558334.5	8.3136e-22

The size of our test sets is 1500 samples. From **Table 5**, we can see that the p-values are close to 0, with no value larger than 0.05 which means that the data of the three models are from different distributions. Also, we use Cliff's Delta to estimate the effect size. Effect size is a simple way of quantifying the difference between two groups that has many advantages over the use of tests of statistical significance alone[1]. It reflects the statistical effect size between two populations. Cliff's Delta *d* produces a value between -1 to 1. 0 indicates that the two groups distributions overlap completely. An effect size of 1 or -1 indicates the absence of overlap between the two groups. The

larger absolute value of *delta* the smaller the overlap between two distributions. The equation is shown in **Equation (2)**.

$$delta = \frac{\sum_{i,j}[x_i > x_j] - [x_i < x_j]}{mn} \quad (2)$$

Where the two distributions are of size n and m with items x_i and x_j respectively. and $[\cdot]$ is the *Iverson bracket*, which is 1 when the contents are true and 0 when false.

From **Table 6**, we can see that the *delta* of NN and MSCN is 0.3634 and the *delta* of NN and PostgreSQL is 0.8752. These imply the prediction data of NN is significantly different from the other two models. Combined with the q-error in Section 4.2, we can draw a conclusion that NN indeed improves the accuracy of cardinality estimation in multiple types of joins situations.

Table 6 Effect Size results for IMDb.

Competing models	Effect Size (d)
NN VS MSCN	0.3634
NN VS PostgreSQL	0.8752
MSCN VS PostgreSQL	0.4552

3.4.2 Statistical Significant Testing And Effect Size for Pagila

We also use Mann–Whitney U test (MWW) and Effect Size to determine whether NN is statistically significantly accurate for the Pagila data set. The p-value is shown in **Table 7**. Our test set is over 1200 queries. From **Table 7**, we can deduce that the prediction data of NN and PostgreSQL are not from the same distribution.

Next, we still use Cliff's Delta to estimate the effect size. The result is shown in **Table 8**. The delta $d = 0.37$, which is larger than 0. Combined with the q-error, we can conclude that the prediction data of NN is different from PostgreSQL, and NN improve the accuracy for the small data set.

Table 7 Mann–Whitney U test experiments results for Pagila.

Competing models	**statistic**	***p-value***
NN VS PostgreSQL	306025	2.258e-14

Table 8 Effect Size results for Pagila.

Competing models	**Effect Size (d)**
NN VS PostgreSQL	0.37

3.5 Hyperparameter Tuning

3.5.1 Hyperparameter Tuning for IMDb

For hyperparameter tuning of our model, we tuned the hyperparameters including epochs, batch size, the number of hidden units and the learning rate. The number of hidden units means the width of the neural network which influences performance of neural networks. More hidden units mean larger model sizes and increased costs in training and prediction. The learning rate and batch are both influence the convergence during training.

We varied the number of epochs (30,40,50,60,70,80,90,100), the number of batch size (16,32,64,128,256,1024), the number of hidden units (32,64,128,256) and the learning rate (0.1,0.01,0.001), resulting in 576 different configurations. For each configuration, we train the model three times using a training sets of 6750 queries and a validation set of 750 queries. After a

detail comparison, we found that the configuration with 40 epochs, 128 batch size, 256 hidden units and 0.001 learning rate performs best over others on the validation set. And after 70 epochs, the model becomes overfitted, this is due to the model capturing the noise in the training set and results in poor prediction performance. **Figure 6** shows the detail of the mean q-error with the number of epochs.

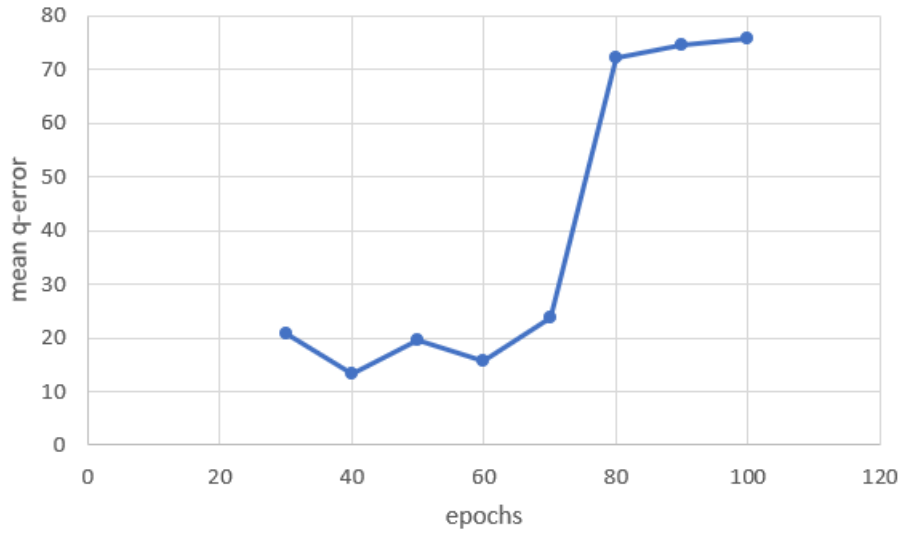

Figure 6 mean q-error with the number of epochs.

3.5.2 Hyperparameter Tuning for Pagila

As what did in hyperparameter tuning for IMDb, we also tuned the hyperparameters including epochs, batch size, the number of hidden units and the learning rate. Like what we did before, we still varied the number of epochs (30,40,50,60,70,80,90,100), the number of batch size (16,32,64,128,256,1024), the number of hidden units (32,64,128,256) and the learning rate (0.1,0.01,0.001), resulting in 576 different configurations. At this time, our training set is 4800 queries and a validation set 480 queries. In the end, we found the best hyperparameters for our model, which are 30 epochs, 32 batch size, 128 hidden units and 0.001 learning rate.

3.6 Runtime Performance

Runtime performance is a key feature of models that can be used for cardinality estimation in a database management system. Since the cardinality estimation for PostgreSQL is directly obtained from database, so we will not add PostgreSQL to comparison. To evaluate the runtime performance of different models, we use a training set containing 7500 queries and test set containing 1500 queries to train models. For MSCN and our model NN, we train the two model with a GPU and the batch size of 128. After the two models convergenced, we compare the training time and the test time for each query. **Table 9** shows the detail about the comparison.

From **Table 9**, it can be seen that the training time of our model is much faster than MSCN as well as the test time per query.

Table 9 Runtime performance comparison.

model	**Training time**	***Test time (per query)***
MSCN	720s	23ms
NN	193s	15ms

CHAPTER 4 CONCLUSIONS AND FUTURE WORK

4.1 Future Work

In this paper, we propose a new approach of using deep learning for cardinality estimation. Our model can beat state-of-the art approaches in multiple types of join situations. However, it can be extended to a more complex and powerful model. In the following, we discuss what we can improve in the future.

More Table Joins. NN can do an excellent job for two and three tables join. However, this is not enough in real work situations. More tables joins brings complex correlations between tables, adapting many tables in multiple types of join situations is a challenging problem.

Precise queries. We generate random queries based upon the database schema for training and testing. Usually, if you want to improve the accuracy of the model, it may need many queries. This kind of work is not efficient enough. The next stage of work is to generate more precise queries which can precisely reflect the relationship between queries. The advantage of precise queries is that the model can learn the correlations between tables faster and use fewer queries. Fewer precise queries can save costs.

Online Learning. Throughout this work, we have assumed an immutable(read-only) database. In the real world, the database is constantly changing. If data and schema change a little, NN can tolerate minor shifts in data distribution and correlation. However, if data and schema change heavily, we can either completely re-train NN or we can modify NN to adapt to that situation, a possible way is to combine incremental learning.

However, completely re-training NN is not easy. It brings considerable costs including re-executing queries to obtain new cardinalities or even generating new queries based on a new schema. If we can combine incremental learning, instead of re-training the whole model, we could use the model and just apply new samples.

Combining traditional approaches. Although traditional approaches perform poorly in cardinality estimation, these approaches are cost effective. If we can combine the histograms and a list of frequent values and its values, we may see great benefits.

4.2 Conclusions

In summary, we introduced a new approach NN which is a deep learning model for cardinality estimation. The experiment results show that our NN can handle multiple types of join situations and performs well in terms of accuracy when compared to other approaches. Our model is the first approach to concentrate on multiple types of join situation including left, inner and right joins.

BIBLIOGRAPHY

[1] Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., & Zdonik, S. B. (2012, April). Learning-based query performance modeling and prediction. *In 2012 IEEE 28th International Conference on Data Engineering* (pp. 390-401). IEEE.

[2] Bruno, N., & Chaudhuri, S. (2002, June). Exploiting statistics on query expressions for optimization. *In Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (pp. 263-274).

[3] Coe, R. (2002). It's the effect size, stupid: What effect size is and why it is important.

[4] Ioannidis, Y. E., & Christodoulakis, S. (1993). Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems (TODS)*, 18(4), 709-748.

[5] Ioannidis, Y. E., & Poosala, V. (1995). Balancing histogram optimality and practicality for query result size estimation. *Acm Sigmod Record*, 24(2), 233-244.

[6] Jagadish, H. V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K. C., & Suel, T. (1998, August). Optimal histograms with quality guarantees. In *VLDB* (Vol. 98, pp. 24-27).

[7] Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., & Kemper, A. (2018). Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*.

[8] Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018, May). The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 489-504).

[9] Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., & Stoica, I. (2018). Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*.

[10] Lakshmi, S., \& Zhou, S. (1998, August). Selectivity estimation in extensible databases-A neural network approach. In *VLDB* (pp. 623-627).

[11] Liu, H., Xu, M., Yu, Z., Corvinelli, V., & Zuzarte, C. (2015, November). Cardinality estimation using neural networks. *In Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering* (pp. 53-59). IBM Corp.

[12] Malik, T., Burns, R. C., & Chawla, N. V. (2007, January). A Black-Box Approach to Query Cardinality Estimation. In *CIDR* (pp. 56-67).

[13] Marcus, R., & Papaemmanouil, O. (2018, June). Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (pp. 1-4).

[14] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., & Cilimdzic, M. (2004, June). Robust query processing through progressive optimization. *In Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (pp. 659-670).

[15] Moerkotte G, Neumann T, Steidl G. Preventing bad plans by bounding the impact of cardinality estimation errors[J]. *Proceedings of the VLDB Endowment*, 2009, 2(1): 982-993.

[16] Mukkamala, R., & Jajodia, S. (1991). A note on estimating the cardinality of the projection of a database relation. *ACM Transactions on Database Systems (TODS)*, 16(3), 564-566.

[17] Ortiz, J., Balazinska, M., Gehrke, J., & Keerthi, S. S. (2018, June). Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning* (pp. 1-4).

[18] Piatetsky-Shapiro, G., & Connell, C. (1984). Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2), 256-276.

[19] Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017, May). Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1009-1024).

[20] Vargha, A., & Delaney, H. D. (2000). A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2), 101-132.

[21] Woltmann, L., Hartmann, C., Thiele, M., Habich, D., & Lehner, W. (2019, July). Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (pp. 1-8).

APPENDIX

Appendix A. The core code of the experiment.

Train.py

```
import argparse
import time
import os
import csv
import torch
from torch.autograd import Variable
from torch.utils.data import DataLoader

from mscn.util import *
from mscn.data import get_train_datasets, load_data, make_dataset
from mscn.model111 import SetConv
from mscn.model import NN


def unnormalize_torch(vals, min_val, max_val):
    vals = (vals * (max_val - min_val)) + min_val
    return torch.exp(vals)


def qerror_loss(preds, targets, min_val, max_val):
    qerror = []
    preds = unnormalize_torch(preds, min_val, max_val)
    targets = unnormalize_torch(targets, min_val, max_val)

    for i in range(len(targets)):
        if (preds[i] > targets[i]).cpu().data.numpy()[0]:
            qerror.append(preds[i] / targets[i])
        else:
            qerror.append(targets[i] / preds[i])
    return torch.mean(torch.cat(qerror))


def predict(model, data_loader, cuda):
    preds = []
    t_total = 0.

    model.eval()
    for batch_idx, data_batch in enumerate(data_loader):

        samples, predicates, joins, targets, types, sample_masks, predicate_masks, join_masks, type_masks = data_batch

        if cuda:
            samples, predicates, joins, targets, types = samples.cuda(), predicates.cuda(), joins.cuda(), targets.cuda(), types.cuda()
            sample_masks, predicate_masks, join_masks, type_masks = sample_masks.cuda(), predicate_masks.cuda(),
join_masks.cuda(), type_masks.cuda()
        samples, predicates, joins, targets, types = Variable(samples), Variable(predicates), Variable(joins), Variable(
            targets), Variable(types)
        sample_masks, predicate_masks, join_masks, type_masks = Variable(sample_masks), Variable(predicate_masks), Variable(
            join_masks), Variable(type_masks)

        t = time.time()
```

```
        outputs = model(samples, predicates, joins, types, sample_masks, predicate_masks, join_masks, type_masks)
        t_total += time.time() - t

        for i in range(outputs.data.shape[0]):
            preds.append(outputs.data[i])

    return preds, t_total


def print_qerror(preds_unnorm, labels_unnorm):
    qerror = []
    for i in range(len(preds_unnorm)):
        if preds_unnorm[i] > float(labels_unnorm[i]):
            qerror.append(preds_unnorm[i] / float(labels_unnorm[i]))
        else:
            qerror.append(float(labels_unnorm[i]) / float(preds_unnorm[i]))

    print("Median: {}".format(np.median(qerror)))
    print("90th percentile: {}".format(np.percentile(qerror, 90)))
    print("95th percentile: {}".format(np.percentile(qerror, 95)))
    print("99th percentile: {}".format(np.percentile(qerror, 99)))
    print("Max: {}".format(np.max(qerror)))
    print("Mean: {}".format(np.mean(qerror)))


def train_and_predict(workload_name, num_queries, num_epochs, batch_size, hid_units, cuda):
    # Load training and validation data
    num_materialized_samples = 1000
    dicts, column_min_max_vals, min_val, max_val, labels_train, labels_test, max_num_joins, max_num_predicates, train_data,
test_data = get_train_datasets(
        num_queries, num_materialized_samples)
    table2vec, column2vec, op2vec, join2vec, type2vec = dicts

    # Train model
    # sample_feats = len(table2vec) + num_materialized_samples
    sample_feats = len(table2vec)
    predicate_feats = len(column2vec) + len(op2vec) + 1
    join_feats = len(join2vec)
    type_feats= len(type2vec)

    # print("sample_feats " + str(sample_feats))
    # print("predicate_feats " + str(predicate_feats))
    # print("join_feats " + str(join_feats))

    # model = SetConv(sample_feats, predicate_feats, join_feats, type_feats, hid_units)

    model = NN(sample_feats, predicate_feats, join_feats, type_feats, hid_units)

    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    if cuda:
        model.cuda()

    train_data_loader = DataLoader(train_data, batch_size=batch_size)
    test_data_loader = DataLoader(test_data, batch_size=batch_size)

    model.train()
    for epoch in range(num_epochs):
        loss_total = 0.

        for batch_idx, data_batch in enumerate(train_data_loader):

            samples, predicates, joins, targets, types, sample_masks, predicate_masks, join_masks, type_masks = data_batch

            if cuda:
```

```
            samples, predicates, joins, targets, types = samples.cuda(), predicates.cuda(), joins.cuda(), targets.cuda(), types.cuda()
            sample_masks, predicate_masks, join_masks, type_masks = sample_masks.cuda(), predicate_masks.cuda(),
join_masks.cuda(), type_masks.cuda()
        samples, predicates, joins, targets, types = Variable(samples), Variable(predicates), Variable(joins), Variable(
            targets), Variable(types)
        sample_masks, predicate_masks, join_masks, type_masks = Variable(sample_masks), Variable(predicate_masks), Variable(
            join_masks), Variable(type_masks)

        optimizer.zero_grad()
        # print()
        # print("samples " + str(samples.shape))
        # print("predicates " + str(predicates.shape))
        # print("joins " + str(joins.shape))
        # print("sample_masks " + str(sample_masks.shape))
        # print("predicate_masks " + str(predicate_masks.shape))
        # print("join_masks " + str(join_masks.shape))
        # print()
        outputs = model(samples, predicates, joins, types, sample_masks, predicate_masks, join_masks, type_masks)
        loss = qerror_loss(outputs, targets.float(), min_val, max_val)
        loss_total += loss.item()
        loss.backward()
        optimizer.step()

    print("Epoch {}, loss: {}".format(epoch, loss_total / len(train_data_loader)))

# Get final training and validation set predictions
preds_train, t_total = predict(model, train_data_loader, cuda)
print("Prediction time per training sample: {}".format(t_total / len(labels_train) * 1000))

preds_test, t_total = predict(model, test_data_loader, cuda)
print("Prediction time per validation sample: {}".format(t_total / len(labels_test) * 1000))

# Unnormalize
preds_train_unnorm = unnormalize_labels(preds_train, min_val, max_val)
labels_train_unnorm = unnormalize_labels(labels_train, min_val, max_val)

preds_test_unnorm = unnormalize_labels(preds_test, min_val, max_val)
labels_test_unnorm = unnormalize_labels(labels_test, min_val, max_val)

# Print metrics
print("\nQ-Error training set:")
print_qerror(preds_train_unnorm, labels_train_unnorm)

print("\nQ-Error validation set:")
print_qerror(preds_test_unnorm, labels_test_unnorm)
print("")

# Load test data
file_name = "workloads/" + workload_name
# joins, predicates, tables, samples, label = load_data(file_name, num_materialized_samples)
joins, predicates, tables, label, types = load_data(file_name, num_materialized_samples)
# Get feature encoding and proper normalization
samples_test = encode_samples(tables, table2vec)
predicates_test, joins_test = encode_data(predicates, joins, column_min_max_vals, column2vec, op2vec, join2vec)
labels_test, _, _ = normalize_labels(label, min_val, max_val)
types_test = encode_types(types, type2vec)

print("Number of test samples: {}".format(len(labels_test)))

max_num_predicates = max([len(p) for p in predicates_test])
max_num_joins = max([len(j) for j in joins_test])

# Get test set predictions
test_data = make_dataset(samples_test, predicates_test, joins_test, types_test,labels_test, max_num_joins, max_num_predicates)
test_data_loader = DataLoader(test_data, batch_size=batch_size)
```

```
    preds_test, t_total = predict(model, test_data_loader, cuda)
    print("Prediction time per test sample: {}".format(t_total / len(labels_test) * 1000))

    # Unnormalize
    preds_test_unnorm = unnormalize_labels(preds_test, min_val, max_val)

    # Print metrics
    print("\nQ-Error " + workload_name + ":")
    print_qerror(preds_test_unnorm, label)

    # Write predictions
    file_name = "results/predictions_" + workload_name + ".csv"
    os.makedirs(os.path.dirname(file_name), exist_ok=True)
    with open(file_name, "w") as f:
        for i in range(len(preds_test_unnorm)):
            f.write(str(preds_test_unnorm[i]) + "," + label[i] + "\n")


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("testset", help="synthetic, scale, or job-light")
    parser.add_argument("--queries", help="number of training queries (default: 10000)", type=int, default=10000)
    parser.add_argument("--epochs", help="number of epochs (default: 10)", type=int, default=10)
    parser.add_argument("--batch", help="batch size (default: 1024)", type=int, default=1024)
    parser.add_argument("--hid", help="number of hidden units (default: 256)", type=int, default=256)
    parser.add_argument("--cuda", help="use CUDA", action="store_true")
    args = parser.parse_args()
    train_and_predict(args.testset, args.queries, args.epochs, args.batch, args.hid, args.cuda)


if __name__ == "__main__":
    # main()
    # train_and_predict("test", 4457, 2, 32, 128, False)
    X=[]
    Y=[]
    file_name = "results/predictions_test.csv"
    with open(file_name, 'r') as f:
        data_raw = list(list(rec) for rec in csv.reader(f, delimiter=','))
        for row in data_raw:
            X.append(float(row[0]))
            Y.append(row[1])

    print("\nQ-Error for postgreSQL :")
    print_qerror(X, Y)
```

Model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from mscn.model111 import SetConv
import numpy as np



# Define model architecture

class NN(nn.Module):
    def __init__(self, sample_feats, predicate_feats, join_feats, type_fests, hid_units):
        super(NN, self).__init__()
        self.SetConv = SetConv(sample_feats, predicate_feats, join_feats, type_fests, hid_units)
        self.Full_Connected1= nn.Linear(128,64)
        self.Full_Connected2= nn.Linear(64,16)
        self.Full_Connected3= nn.Linear(16,1)
        # self.Conv1 = nn.Conv1d(256,64,1)
        # self.relu= nn.ReLU()
        # self.maxpooling = nn.MaxPool1d(1)
        # self.Conv2 = nn.Conv1d(64,32,1)
        # self.flatten=nn.Flatten()
        # self.Drop= nn.Dropout(0.2)



    def forward(self, samples, predicates, joins, types, sample_mask, predicate_mask, join_mask, type_mask):
        # samples has shape [batch_size x num_joins+1 x sample_feats]
        # predicates has shape [batch_size x num_predicates x predicate_feats]
        # joins has shape [batch_size x num_joins x join_feats]
        # return self.SetConv(samples, predicates, joins, sample_mask, predicate_mask, join_mask)
        x=self.SetConv(samples, predicates, joins, types, sample_mask, predicate_mask, join_mask,type_mask)
        x = x.view(x.size(0), -1)
        x=self.Full_Connected1(x)
        x=self.Full_Connected2(x)
        x=self.Full_Connected3(x)
        # x= torch.unsqueeze(x,-1)
        # x=self.Conv1(x)
        # x = self.relu(x)
        # x= self.maxpooling(x)
        # x=self.Drop(x)
        #
        # x= self.Conv2(x)
        # x= self.relu(x)
        # x= self.maxpooling(x)
        # x=self.flatten(x)
        #
        # x=self.Full_Connected3(x)
        out=torch.sigmoid(x)
        return out
```

Util.py

```
import numpy as np


# Helper functions for data processing

def chunks(l, n):
    """Yield successive n-sized chunks from l."""
    for i in range(0, len(l), n):
        yield l[i:i + n]


def get_all_column_names(predicates):
    column_names = set()
    for query in predicates:
        for predicate in query:
            if len(predicate) == 3:
                column_name = predicate[0]
                column_names.add(column_name)
    return column_names


def get_all_table_names(tables):
    table_names = set()
    for query in tables:
        for table in query:
            table_names.add(table)
    return table_names


def get_all_operators(predicates):
    operators = set()
    for query in predicates:
        for predicate in query:
            if len(predicate) == 3:
                operator = predicate[1]
                operators.add(operator)
    return operators


def get_all_joins(joins):
    join_set = set()
    for query in joins:
        for join in query:
            join_set.add(join)
    return join_set

def get_all_type(type):
    type_set = set()
    for t in type:
        type_set.add(t)
    return  type_set

def idx_to_onehot(idx, num_elements):
    onehot = np.zeros(num_elements, dtype=np.float32)
    onehot[idx] = 1.
    return onehot


def get_set_encoding(source_set, onehot=True):
    num_elements = len(source_set)
    source_list = list(source_set)
    # Sort list to avoid non-deterministic behavior
```

```
    source_list.sort()
    # Build map from s to i
    thing2idx = {s: i for i, s in enumerate(source_list)}
    # Build array (essentially a map from idx to s)
    idx2thing = [s for i, s in enumerate(source_list)]
    if onehot:
        thing2vec = {s: idx_to_onehot(i, num_elements) for i, s in enumerate(source_list)}
        return thing2vec, idx2thing
    return thing2idx, idx2thing


def get_min_max_vals(predicates, column_names):
    min_max_vals = {t: [float('inf'), float('-inf')] for t in column_names}
    for query in predicates:
        for predicate in query:
            if len(predicate) == 3:
                column_name = predicate[0]
                val = float(predicate[2])
                if val < min_max_vals[column_name][0]:
                    min_max_vals[column_name][0] = val
                if val > min_max_vals[column_name][1]:
                    min_max_vals[column_name][1] = val
    return min_max_vals


def normalize_data(val, column_name, column_min_max_vals):
    min_val = column_min_max_vals[column_name][0]
    max_val = column_min_max_vals[column_name][1]
    val = float(val)
    val_norm = 0.0
    if max_val > min_val:
        val_norm = (val - min_val) / (max_val - min_val)
    return np.array(val_norm, dtype=np.float32)


def normalize_labels(labels, min_val=None, max_val=None):
    labels = np.array([np.log(float(l)) for l in labels])
    if min_val is None:
        min_val = labels.min()
        print("min log(label): {}".format(min_val))
    if max_val is None:
        max_val = labels.max()
        print("max log(label): {}".format(max_val))
    labels_norm = (labels - min_val) / (max_val - min_val)
    # Threshold labels
    labels_norm = np.minimum(labels_norm, 1)
    labels_norm = np.maximum(labels_norm, 0)
    return labels_norm, min_val, max_val


def unnormalize_labels(labels_norm, min_val, max_val):
    labels_norm = np.array(labels_norm, dtype=np.float32)
    labels = (labels_norm * (max_val - min_val)) + min_val
    return np.array(np.round(np.exp(labels)), dtype=np.int64)


def encode_samples(tables,  table2vec):
    samples_enc = []
    for i, query in enumerate(tables):
        samples_enc.append(list())
        for j, table in enumerate(query):
            sample_vec = []
            # Append table one-hot vector
            sample_vec.append(table2vec[table])
            # Append bit vector
```

```
            # sample_vec.append(samples[i][j])
            sample_vec = np.hstack(sample_vec)
            samples_enc[i].append(sample_vec)
    return samples_enc

def encode_types(types,  type2vec):
    type_enc = []
    for i, query in enumerate(types):
        type_enc.append(list())
        sample_vec = []
        # Append table one-hot vector
        sample_vec.append(type2vec[query])
        # Append bit vector
        # sample_vec.append(samples[i][j])
        sample_vec = np.hstack(sample_vec)
        type_enc[i].append(sample_vec)
    return type_enc

def encode_data(predicates, joins, column_min_max_vals, column2vec, op2vec, join2vec):
    predicates_enc = []
    joins_enc = []
    for i, query in enumerate(predicates):
        predicates_enc.append(list())
        joins_enc.append(list())
        for predicate in query:
            if len(predicate) == 3:
                # Proper predicate
                column = predicate[0]
                operator = predicate[1]
                val = predicate[2]
                norm_val = normalize_data(val, column, column_min_max_vals)

                pred_vec = []
                pred_vec.append(column2vec[column])
                pred_vec.append(op2vec[operator])
                pred_vec.append(norm_val)
                pred_vec = np.hstack(pred_vec)
            else:
                pred_vec = np.zeros((len(column2vec) + len(op2vec) + 1))

            predicates_enc[i].append(pred_vec)

        for predicate in joins[i]:
            # Join instruction
            join_vec = join2vec[predicate]
            joins_enc[i].append(join_vec)
    return predicates_enc, joins_enc
```