DeepQT : Learning Sequential Context for Query Execution Time Prediction

Jingxiong Ni[1], Yan Zhao[2], Kai Zeng[3], Han Su[1], and Kai Zheng[1*]

[1] University of Electronic Science and Technology of China, Chengdu, China
nijingxiong@std.uestc.edu.cn, {hansu,zhengkai}@uestc.edu.cn
[2] School of Computer Science and Technology, Soochow University, Suzhou, China
zhaoyan@suda.edu.cn
[3] Alibaba Group, HangZhou, China
zengkai.zk@alibaba-inc.com

Abstract. Query Execution Time Prediction is an important and challenging problem in the database management system. It is even more critical for a distributed database system to effectively schedule the query jobs in order to maximize the resource utilization and minimize the waiting time of users based on the query execution time prediction. While a number of works have explored this problem, they mostly ignore the sequential context of query jobs, which may affect the performance of prediction significantly. In this work, we propose a novel Deep learning framework for Query execution Time prediction, called DeepQT, in which the sequential context of a query job and other features at the same time are learned to improve the performance of prediction through jointly training a recurrent neural network and a deep feed-forward network. The results of the experiments conducted on two datasets of a commercial distributed computing platform demonstrate the superiority of our proposed approach.

Keywords: Deep Learning · Query-time Prediction · Distributed Database · Jointly Training.

1 Introduction

Modern database management can greatly benefit from the prediction of query execution time, which aims to predict the time between the start of a SQL query job in database management and the end of the job. The prediction of query time is an important research topic and can be used in many scenarios, such as admission control decisions [23], query scheduling decisions [6], query monitoring [19], system sizing [25], and so forth. Within the generation of distributed computing platforms such as MaxCompute developed by Alibaba, we can obtain high volumes of information about users' query jobs which contains different SQL statement, query execution plans, and job configurations. Based on these

* Corresponding author: Kai Zheng.

historical data, it is possible for us to make an accurate prediction about query execution time using deep neural networks.

In existing works, query time prediction problem is conventionally tackled by cost-based analytical modeling approach [27] or traditional machine learning techniques, such as Multiple Linear Regression (MLR) [2] and Support Vector Regression (SVR) [24]. While the cost-based analytical modeling approaches are good at comparing the costs of alternative query execution plans, they are poor predictors of query execution time, especially in the commercial distributed database management where there are more sophisticated factors to consider and less information about the query execution cost than centralized database management. As for the previous works using traditional machine learning techniques, there are two limitations. First, due to the lack of real-world data, the training data used in previous work is generated by a small amount of benchmark query statement like TPC-H, ignoring the complexity and diversity of users' real queries. Second, traditional machine learning techniques lack the ability to model complex patterns in a large amount of real-world data. Moreover, most of the previous works take the number of operators in the execution plan as the input feature without considering the order and dependency between the operators in query execution plan. However, such information in query jobs can greatly affect the query execution time.

In this paper, we take the order and strong dependency between operators in a query's execution plan into consideration. Figure 1 presents an example of query execution plan, which can be expressed as sequences after topologically sorting. Thus, we model the query execution plan as a sequential context to learn the information in the order and dependency between operators. In addition, since the number of operators in a query execution plan is not known, the length of the execution plan topology is not available in advance as well. Therefore we adopt a Recurrent Neural Network (RNN) in modeling the sequential context of a user's query job, because RNN has the powerful ability to handle sequential data whose length is not known beforehand and has superiority in encoding dependencies [21].

In addition to the query execution plan, we can extract many critical features from the job configuration generated by the query optimizer, such as PlanMem, PlanCpu (see Table 1) and so on. We leverage the Deep Neural Networks (DNN) to learn the information provided by these features. DNN has been successfully applied to various difficult problems and has achieved excellent performance, due to its ability to perform arbitrary parallel computation for a modest number of steps.

In summary, RNN is good at learning the sequential features, and DNN has the superior performance in learning information from non-sequential features. Therefore we aim to combine them together to improve the performance of query execution time prediction.

Our main contributions are summarized as follows:

– We propose a learning framework based on deep neural network, namely DeepQT, to solve the problem of query execution time prediction. By lever-

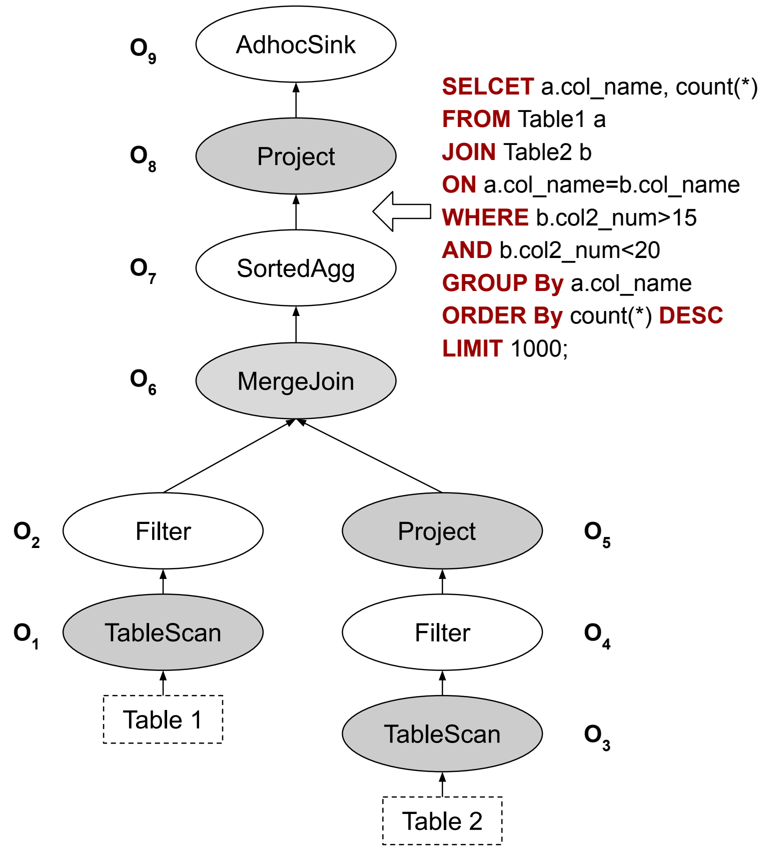


Fig. 1. Illustration for the query execution plan

aging the large volume of historical user query job data stored in MaxCompute [4], DeepQT has the ability of learning sequential context to make an accurate query time prediction by jointly training a recurrent model component and a deep model component.

– Taking the sequential context of query jobs into consideration, we devise a novel neural network architecture to learn the pattern across different job configurations (e.g., PlanMem, PlanCpu and RunningMode (see Table 1)) and the sequential contextual information (i.e., the order and dependency between the operators in query execution plan) simultaneously. Such sequential contextual information can help to significantly improve the accuracy of the prediction.

[4] MaxCompute (previously known as ODPS) is a general purpose, fully managed, multi-tenancy data processing platform for large-scale data warehousing. https://www.alibabacloud.com/product/maxcompute

– We evaluate our model on the data about real-world users' query jobs, which is more important and challenging than evaluating on the data about benchmark query jobs. The experimental results demonstrate the advantage of our model over existing methods.

2 Problem Definition

Table 1. The features of query job configuration

Features	Comment
PlanMem	the memory allocated to the job
PlanCpu	the cpu allocated to the job
RunningMode	the running mode of the job
ExecutorNum	the number of executor allocated to the job
InputRecord	the number of records the query need to scan
TableNum	the number of table relevant to the job
taskNum	the number of task in execution plan
instNum	the number of instance in execution plan
RunningCluster	the running cluster of the job

When users submit query jobs to the system, query optimizer of the system would generate an execution plan of the query job and the corresponding job configuration. The job configuration includes general descriptions about the execution environment of query jobs (as shown in Table 1). The query execution plan consists of many operators in tree-based structures (see Figure 1), which is also named Physical Operators Tree (POT). Thus, the topology of a query execution plan is used to represent the query job.

Let $U = \{u_1, u_2, ..., u_N\}$ denote a set of users and $P_n = \{q_1, q_2, ..., q_{c_n}\}$ denote all query execution plans (POTs) for each user u_n, where $q_{i=1,2,...,c_n}$ is the representation of query execution plan and c_n is the number of plans. There should be a corresponding time spent by execution plan, which is defined as $T_n = \{t_1, t_2, ..., t_{c_n}\}$. Moreover, we use $J_n = \{j_1, j_2, ..., j_{c_n}\}$ to denote the corresponding job configuration of execution plan.

Problem Statement: Given the historical data P and J of users in U, as well as their corresponding execution time T, our goal is to learn a predictor to estimate the execution time of the newly query job of users based on the configuration and execution plan of the job.

3 DeepQT Model

3.1 Model Overview

In our study, we divide the input features into two categories: 1) sequential feature (i.e., POT) and 2) non-sequential features (e.g., InputRecord and PlanMem). Inspired by the DeepFM model [11] that can well utilize the advantages

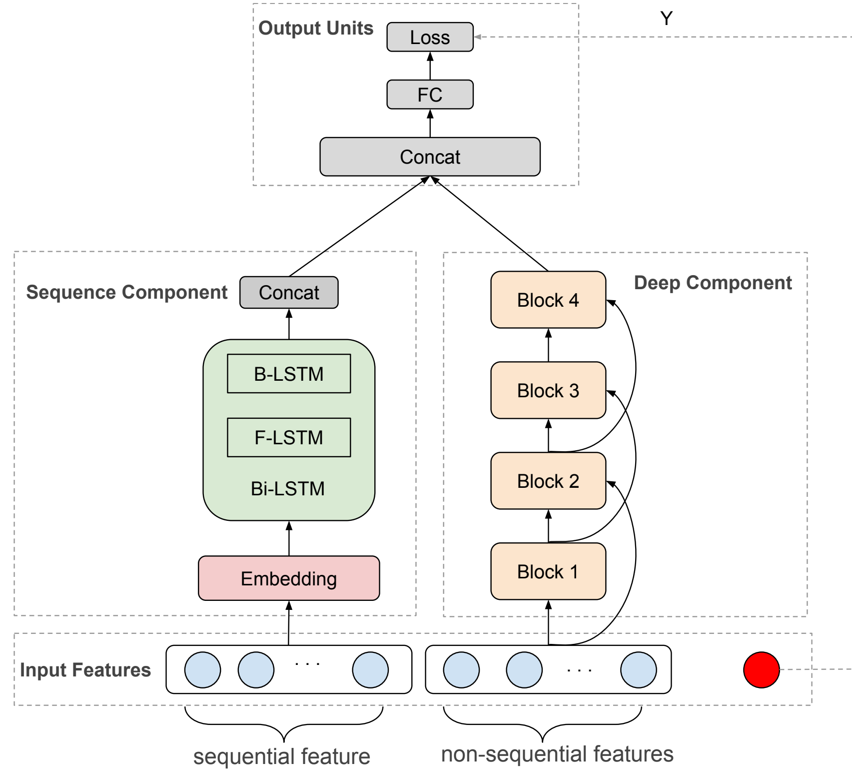


Fig. 2. Overview of the DeepQT architecture for query execution time prediction(F-LSTM: Forward Long Short-Term Memory; B-LSTM: Backward Long Short-Term Memory; Bi-LSTM: Bidirectional Long Short-Term Memory; Block: the combination of a series of consecutive operations whose details are shown in Figure 4)

of Factorization Machine (FM) and feed-forward neural network to learn both low-order and high-order feature interactions, we propose a new neural network model, DeepQT, to handle the above two kinds of features.

As shown in Figure 2, our model consists of two main components, a sequence component and a deep component. Taking the topology of POT as input in sequence component and other non-sequential features as input in deep component, the two components are jointly trained for learning sequential contextual information and learning non-sequential features. The structure of the joint training framework we devised has the ability to integrate the knowledge learned by the above two components.

Specifically, through a lookup table operation, the POT is first mapped to the embedding vectors with a fixed dimension, then fed into RNN layers to learn the sequential pattern of POT. The deep component is a feed-forward neural network with skip connection, which aims to learn the pattern of the other non-

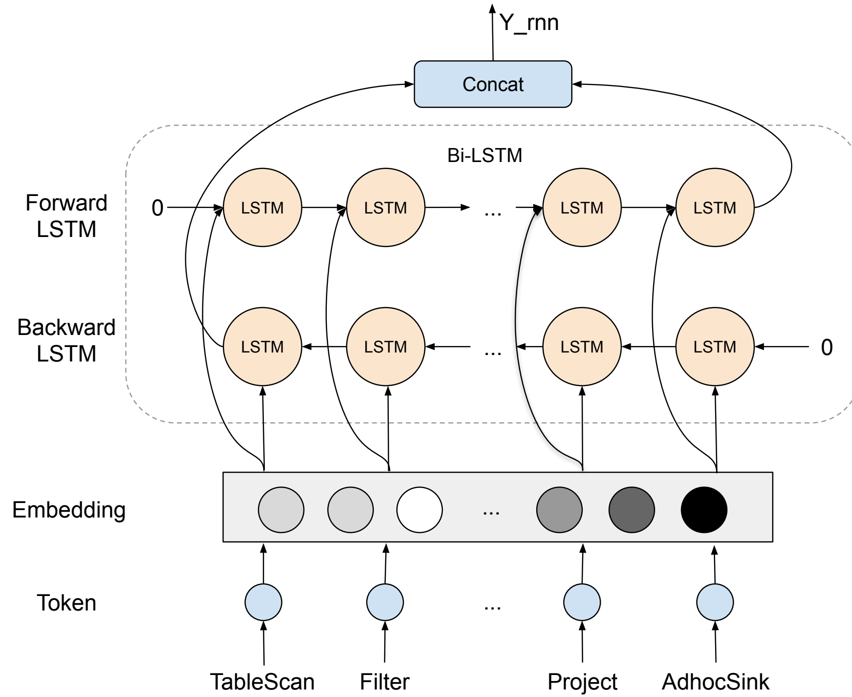


Fig. 3. The detail of Bi-LSTM in architecture overview

sequential features. In the output units, the outputs of the sequence component and the deep component are concatenated first, then fed into a fully connected layer to get the final output, which can be defined as follows:

$$\widehat{y} = W^{out}[y_{deep} : y_{sequence}] + b^{out}, \tag{1}$$

where $\widehat{y}$ is the predicted execution time, y_{deep} is the output of deep component, $y_{sequence}$ is the output of sequence component, $[y_{deep} : y_{sequence}]$ is the concatenation of the outputs, W^{out} and b^{out} are the learnable parameters. Then, the $\widehat{y}$ will be fed into a loss function for joint training.

3.2 Sequence Component

The sequence component is used to learn the dependency between operators in the query execution plan, which is illustrated in Figure 3. Through tokenization, we represent the original operators in a query plan q_n as $O^n = [o_1^n, ..., o_{c_{q_n}}^n]$, and apply the commonly used embedding layer to map each operator into an embedding vector with fixed-sized dimension, denoted by $E^n = [e_1^n, ..., e_{c_{q_n}}^n]$. The embedding vectors can be learned during training to obtain more accurate representations of each operator. Then, we feed the embedding vectors into RNN layers which are sensitive to operator order and can learn the complex sequential

and dynamical relationship between embedding vectors well. Since the standard RNN would be hard to train as result of the long-term dependencies in sequence and the gradient vanishing [3], we apply the Long Short Term Memory (LSTM) cell to solve this problem for its powerful ability to learn long-term dependencies and prevent vanishing gradient problem.

The LSTM cells are building units for layers in RNNs, which introduce the gate mechanism. In LSTM, there are three gates (i.e., input gate, forget gate and output gate), each of which contains its own individual learnable variables [15]. These multiple gates allow the cells in LSTM to control the proportion of information to forget and to store. As a result, LSTM shows a significant improvement in addressing long-term dependency problem. Specifically, the hidden layers of LSTM can be computed as:

$$h_l^n = LSTM(e_l^n, h_{l-1}^n), \tag{2}$$

Where h_l^n is the output of LSTM hidden layer, h_{l-1}^n is the output of previous LSTM hidden layer and e_l^n is the input of LSTM hidden layer, which refer to embedding vectors of operators.

Moreover, to learn the sequence information better, a bidirectional LSTM consisting of a forward and a backward LSTM is applied to learn both forward sequence and reversed sequence, which is depicted in Figure 3. Through learning reversed order of sequence simultaneously, many short-term dependencies can be introduced to make the process of optimization much easier. The final output of the sequence component can be computed as:

$$y_{sequence} = [\overleftarrow{h_{c_{q_n}}^n} : \overrightarrow{h_{c_{q_n}}^n}], \tag{3}$$

where $\overleftarrow{h_{c_{q_n}}^n}$ is the last hidden state of backward LSTM, $\overrightarrow{h_{c_{q_n}}^n}$ is the last hidden state of forward LSTM, and the concatenation of these two states is the final output of sequence component.

3.3 Deep Component

The deep component is a feed-forward neural network using the skip connection, which aims to create short paths from previous layers to the subsequent layers. In our implementation, we combine some consecutive operations into a block, which is illustrated in Figure 4. In a block, we adopt the Batch Normalization (BN) [14] right after Fully Connected (FC) layer and before Rectified Linear Unit (ReLU) [10] activation function to increase the speed of model training. The Batch Normalization (BN) layer is a novel mechanism for reducing the internal covariate shift, which refers to the change in the distribution of network activation caused by the change in network parameters during training. Inside layers, the normalization is performed for each input mini-batch. Besides, two additional learnable parameters are introduced to ensure the representation ability of the network [14]. In previous researches, it has been proved that the BN is an effective and promising way for improving the gradient propagation and the training speed of the network.

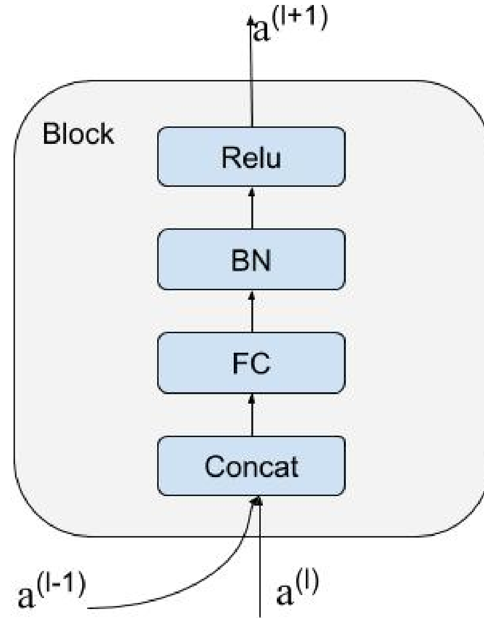


Fig. 4. The detail of the block (FC: Fully-connected; BN: Batch Normalization)

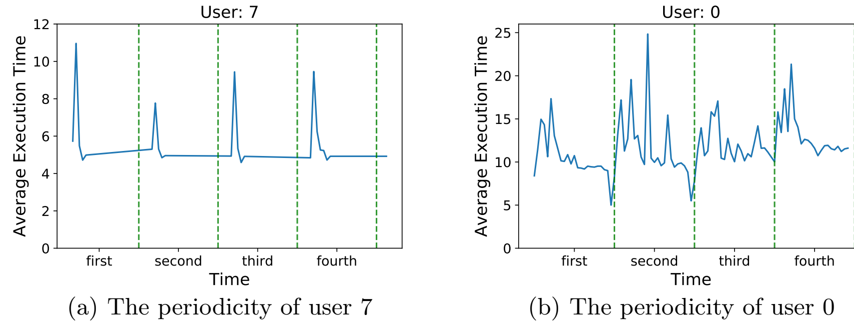
In addition, compared to traditional feed-forward neural network that is composed of fully connection, the deep component in DeepQT adds residual skip connections (called RES) between all blocks (see Figure 2). For each block, the input is the concatenation of the outputs of the precedent block, and its output is used as input in the later layers. Due to the success of residual skip connection in alleviating the vanishing gradient problem and enhancing the feature propagation [12, 13], we adopt it as our connection mode, which is beneficial for both convergence rate of the deep component and the performance of prediction. Specifically, the forward process can be denoted as:

$$a^{(l)} = H^{(l)}(W^{(l)}[a^{(l-2)}, a^{(l-1)}] + b^{(l)}), \tag{4}$$

where $a^{(l)}$ refers to the output of the l-th block. $a^{(0)}$ denotes the input vector, and $[a^{(l-1)}, a^{(l-1)}]$ is the concatenation of the outputs produced in $(l-1)$-th block and $(l-2)$-th block. We define a function $H^{(l)}$, which begins with the batch normalization, followed by a ReLU activation function.

3.4 Periodicity Analysis

In this section, we extract periodicity information to get a more accurate prediction result. According to our observations, the execution time of query jobs submitted by the same user usually changes periodically, which means that the execution time of query jobs at a certain time interval is similar to the same



(a) The periodicity of user 7 (b) The periodicity of user 0

Fig. 5. Periodicity Information

time interval of the previous day for the same user. In Figure 5, we collect the query jobs of two users submitted in four days to analyze the periodic characteristics of the execution time. Figure 5(a) depicts the daily periodicity with four obvious peaks. Although the execution time of query jobs is more turbulent in Figure 5(b), we can still find the daily periodicity, especially at the early part of the day where there are two to three peaks. Moreover, it is observed that the execution time in the late part of the day is obviously less than that in the early part of the day, which implies that users tend to submit large query jobs in the morning and small query jobs late in the day. The periodicity information can be regarded as the supplementary of the learnt features. Therefore, we add the periodicity data as a non-sequential feature into the deep component.

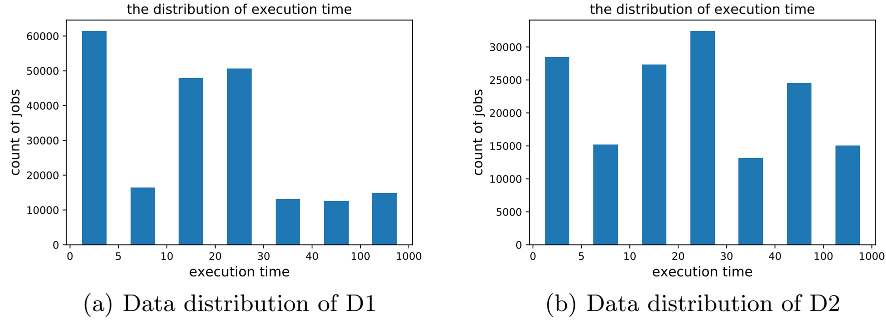
For extracting the periodicity information, the granularity of a timestamp is first set as 1 hour (i.e., 10:00 am-11:00 am) and then represented as a one-hot vector. Finally, we feed all the one-hot vectors into the deep component as an input feature to capture the pattern of daily periodicity for execution time.

4 Experiment

In this section, we first describe our training datasets and training details. Moreover, we conduct extensive experiments to evaluate our proposed model on these datasets.

4.1 Training Details

The model is trained to minimize the L2 loss between the training data and the predictions. Formally, the loss function can be computed as $L = \sum_{q \in Q} \left(\widehat{y}^q - y^q\right)^2$, where $\widehat{y}^q$ is the predicted value, y^q is the real value and Q is the training data. In addition, for the implementation of our model, we utilize TensorFlow. We use the ReLU [20] activation function after every fully connected layer. The weights in DeepQT are initialized with Xavier initialization [9]. The model is trained with the batch size of 1024 and a mini-batch stochastic gradient descent using



(a) Data distribution of D1 (b) Data distribution of D2

Fig. 6. Distribution of execution time in datasets

Adaptive Moment Estimation (Adam) optimizer [16] with a learning rate 0.001. Moreover, the hyperparameters of our best-performing model in experiments are shown as follows. The dimension of the word embedding is 64; the dropout is applied before all fully connected layers with the ration of 0.3; the size of each hidden recurrent layer in the sequence component is 128; sizes of fully connected layers are 258, 126, 64, and 32 respectively. Besides, the early stopping strategy is adopted to terminate the training process when the model achieves the best performance on validation datasets. For the hardware environment, one NVIDIA GTX 1080ti GPU with 8 GB of RAM is used to train the model.

4.2 Datasets

MaxCompute is a commercial distributed computing platform for large-scale data warehousing, in which users submit millions of queries every day. When users' queries are submitted to the platform, the MaxCompute will generate a corresponding job. Such query job contains an execution plan of the query and a job configuration file which describes the execution environment of the job. We generate two training datasets by extracting query logs from MaxCompute in a period of time. In particular, the first dataset, **D1**, contains more than 217,000 query jobs of users, allocated from 13th November to 13th December in 2019. The second dataset, **D2**, contains about 156,000 query jobs of users, allocated from 13th October to 9th December in 2019. In order to highlight the generality of the model, the dataset **D2** we obtain has different data distribution from the **D1**, as shown in Figure 6. We can find that the execution time of most query jobs on the **D1** is less than 30 seconds, and the distribution of execution time on the **D2** is more homogeneous. Two datasets have the same fields of features and these 29 fields of features in the datasets are divided into three categories: 1) sequential context (i.e., execution plan (POT)), 2) job configuration (such as PlanMem, PlanCpu and InputRecord (see Figure 1)) and 3) the submitting time of query jobs. All fields of the features are used in the experiment. Furthermore, 30% of each dataset are randomly selected as test data and the rest as training data.

4.3 Baselines

The method proposed in this paper is compared with the following baseline methods:

- **SVR** [24]: As a traditional machine learning technology, Support Vector Regression (SVR) has been proved to be able to achieve good performance in the regression problem.
- **MLR** [2]: As the most common form of linear regression analysis, Multiple Linear Regression (MLR) is used to predict the query execution time.
- **DTR**: By using Decision Tree Regression (DTR), a regression model is built in the form of a tree structure to perform the prediction.
- **RFR**: Random Forest Regression (RFR) is an ensemble technique, which can perform the regression task using multiple decision trees and a technique called Bootstrap Aggregation. The RFR implementation from the scikit-learn library is used in this paper.
- **GBR**: Using Gradient Boosting Regression (GBR), a prediction model is built in the form of an ensemble of weak prediction models. It has been proved that it has the powerful ability in regression tasks.
- **XGBoost**: As a popular and efficient ensemble method, XGBoost [4] has been widely applied in many regression problems because of its remarkable performance.

Except for the job execution plan (i.e., POT), the input features for baseline methods are the same as those for the method proposed in this paper for the purpose of fair comparisons. The number of different operations in the POT is used as input features in baseline methods. In contrast, the whole topology of POT is used as an input feature in the proposed method.

4.4 Effectiveness Comparisons

Table 2. Comparison among different methods on D1 dataset

Method	RMSE
MLR	9.498
SVR	9.756
DTR	9.973
RFR	8.310
GBR	8.249
XGBoost	8.087
DeepQT-4-DNN	8.234
DeepQT-4-DNN-RES	8.118
DeepQT-4-DNN-RNN	8.005
DeepQT-4-DNN-RNN-RES	**7.998**
DeepQT-5-DNN-RNN-RES	8.016
DeepQT-3-DNN-RNN-RES	8.116

Performances of our method and other baseline methods are evaluated by Root Mean Square Error (RMSE). Formally, $\widehat{y}^q$ is used to denote the predicted value of y^q, and Q is used to denote the test data. Then, the RMSE can be defined as follows:

$$RMSE = \sqrt{\frac{1}{|Q|} \sum_{q \in Q} \left(\widehat{y}^q - y^q\right)^2} \tag{5}$$

We first compare DeepQT with the baselines on the **D1** dataset, which is depicted in Table 2. Meanwhile, we compare 6 variants of DeepQT with different components and layers to evaluate the effect of each component and the number of hidden layers. Taking DeepQT-4-DNN-RNN-RES as an example, it adds RES connections between 4 block layers in the deep component, where each block contains a BN layer, an FC layer, a concatenation layer and an activation function ReLU. Besides, it has a sequential component for learning sequential context. On the contrary, DeepQT-4-DNN means the model that only has a deep component.

It can been seen from Table 2 that the results of some tradition machine learning methods, such as MLR, SVR, and DTR, are obviously larger than other methods. The advanced methods based on ensemble learning (i.e., RFR, GBR and XGBoost) all provide a result of well-performance prediction, among which XGBoost achieves the best prediction accuracy among the baseline methods in terms of RMSE. The method proposed in this paper outperforms other existing baseline methods. It can be seen from the experiments that the RMSE of DeepQT-4-DNN- RNN-RES is 7.998, which significantly improves the prediction accuracy. The results of DeepQT-4-DNN and DeepQT-4-DNN-RES show that the deep feed-forward neural network with skip connections can achieve a well-performance result compared with other traditional methods; that is, skip connections indeed can improve the performance of DeepQT. The reason may be that the skip connections can enhance the gradient propagation and avoid losing some shallow information during propagation. Moreover, taking sequential context into consideration, the topology of POT is fed into the sequence component (BiLSTM in our framework) in DeepQT-4-DNN-RNN. The results show that DeepQT-4-DNN-RNN is further promoted, which indicate that the sequence component can exactly capture the pattern of order and dependency in sequential context. Meanwhile, the joint training framework in this paper has the ability to supplement the knowledge learned in the deep component with the knowledge learned in the sequential context. In addition, from the results of the experiments, we can find that DeepQT with 4 blocks can obtain a better result than DeeQT with 3 or 5 blocks. Although the effectiveness of DeepQT-4-DNN-RNN-RES is slightly improved compared with DeepQT-4-DNN-RNN, its efficiency of convergence is greatly improved, which we will show in **Section 4.5.**

Table 3 shows the results of experiments on the **D2**. From the results, we can see that our proposed approach achieves the best performance on this dataset. We can find that the RMSE of the DeepQT-4-DNN-RNN-RES has relatively from 3.8% up to 29.9% lower than these baselines on the **D2**, which demonstrates

Table 3. Comparison among different methods on D2 dataset

Method	RMSE
MLR	12.339
SVR	12.867
DTR	13.583
RFR	12.567
GBR	11.174
XGBoost	10.858
DeepQT-4-DNN	11.412
DeepQT-4-DNN-RES	10.887
DeepQT-4-DNN-RNN	10.475
DeepQT-4-DNN-RNN-RES	**10.452**
DeepQT-5-DNN-RNN-RES	10.566
DeepQT-3-DNN-RNN-RES	10.626

the excellent generalization and superiority of our proposed approach on the datasets with different data distribution.

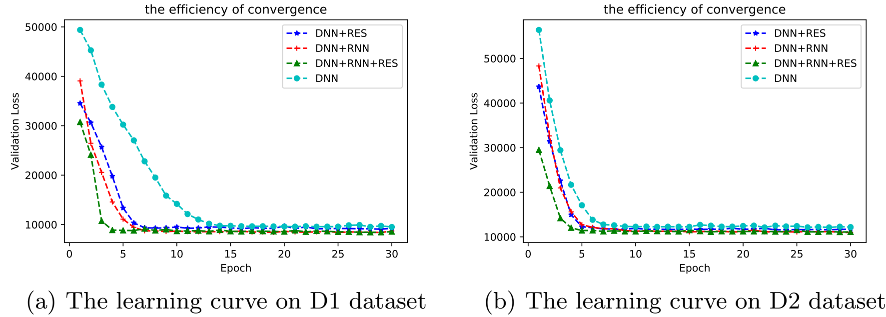
Since the execution time of most queries in datasets is between 0 and 100 seconds (shown in Figure 6), and the number of samples is large enough, the minor improvement of RMSE, 0.1 or 0.4, would represent a significant improvement in efficiency. Specifically, for a small number of long-term tasks in datasets, it would be an error difference of one minute or two minutes. Hence, it is essential for downstream tasks (e.g., Task Scheduling [6]).

4.5 Efficiency Analysis

In this section, we conduct experiments on two datasets to evaluate the efficiency of all the DeepQT variants by comparing their convergence rates. To ensure a fair comparison, we maintain the same hyperparameters for different variants and train the variants using Adam method [16] with batch size of 1024 and learning rate 0.001. The weights in our framework are initialized with Xavier initialization [9]. Learning curves are presented in Figure 7.

It is noted that the variant with the sequence component (marked as DNN+RNN) and the variant with the RES connection (marked as DNN+RES) converge much faster than the variants with a single component (marked as DNN), which means the proposed model can be trained by using less epochs. Furthermore, the sequence component and the RES connection not only increase the prediction accuracy but also improve the convergence rate. Not surprisingly, the combination of both sequence component and RES connection (marked as DNN+RNN+RES) can achieve the best performance in convergence rate.

To sum up, it can be found from the experiments that the proposed framework can well learn the sequential context to increase the prediction accuracy and reduce the iterative time for convergence under different data distributions.



(a) The learning curve on D1 dataset

(b) The learning curve on D2 dataset

Fig. 7. Efficiency of convergence on datasets

5 Related Work

5.1 Query Execution Time Prediction

As a significant problem in the database management research, the query execution time prediction has received great attention over the last decades. [8] proposes an approach to address this problem based on a Kernel Canonical Correlation Analysis (KCCA) modeling technique. After then many machine learning techniques are applied to query time prediction, such as Support Vector Regression (SVR) [24], Multiple Additive Regression-Tree (MART) [17] and Multiple Linear Regression (MLR) [2]. Unlike these work based on machine learning technique, [27] proposes a method based on calibrating cost models of the query optimizer. However, the above works always assume the workload of the database is static, which is unrealistic. To relieve this problem, some works are proposed by considering the workload is concurrent and dynamic for generality, such as [26], [1] and [7]. However, these works are not general enough. The query data they use for the experiments is still baseline queries rather than real-world queries. Moreover, the database management they use for the experiments is still centralized rather than distributed.

5.2 Deep Learning

Deep learning techniques have been applied to many difficult problems in various domains and achieved excellent success [5, 12, 21]. In this paper, a recurrent neural network and a deep feed-forward network are combined into a model for joint training to solve the difficulties in predicting query execution time. The idea of jointly training is inspired by previous researches such as Wide & Deep [5], which explores the joint training of a linear model and a feed-forward neural network for the CTR prediction. The DeepFM proposed by [11] is an extension of Wide & Deep model to share the input embedding for both the wide part and the deep part. In computer vision, the joint training of the convolution network and a graphical model have been applied to the human pose

estimation from images [22]. Additionally, in language models, a joint training of a maximum entropy model and a recurrent neural network are proposed to reduce the computational complexity [18]. Different from previous researches, our proposed DeepQT is jointly trained for learning the pattern from both sequential features and non-sequential features.

6 Conclusion

In this paper, we study the problem of query execution time prediction. We propose a learning framework named DeepQT, which jointly trains a recurrent neural network and a deep feed-forward neural network. Our approach has the ability to learn sequential context to improve prediction accuracy and reduce the number of convergent iterations. Extensive experiments are conducted on real-world datasets, whose results show that our model outperforms the existing methods. Moreover, our framework has excellent generality and can achieve great performance on datasets with different data distributions. One of our future work is to incorporate our framework into the database management system so that it can be used in more applications.

Acknowledgement

This work is partially supported by Natural Science Foundation of China (No. 61972069, 61836007, 61832017, 61532018, 61802054) and Alibaba Innovation Research (AIR).

References

1. Ahmad, M., Duan, S., Aboulnaga, A., Babu, S.: Predicting completion times of batch query workloads using interaction-aware models and simulation. In: EDBT. pp. 449–460 (2011)
2. Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., Zdonik, S.B.: Learning-based query performance modeling and prediction. In: ICDE. pp. 390–401 (2012)
3. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks **5**(2), 157–166 (1994)
4. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: SIGKDD. pp. 785–794 (2016)
5. Cheng, H.T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., et al.: Wide & deep learning for recommender systems. In: DLRS. pp. 7–10 (2016)
6. Chi, Y., Moon, H.J., Hacigümüş, H.: icbs: incremental cost-based scheduling under piecewise linear slas. PVLDB **4**(9), 563–574 (2011)
7. Duggan, J., Cetintemel, U., Papaemmanouil, O., Upfal, E.: Performance prediction for concurrent database workloads. In: SIGMOD. pp. 337–348 (2011)
8. Ganapathi, A., Kuno, H., Dayal, U., Wiener, J.L., Fox, A., Jordan, M., Patterson, D.: Predicting multiple metrics for queries: Better decisions enabled by machine learning. In: ICDE. pp. 592–603 (2009)

9. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: AISTATS. pp. 249–256 (2010)
10. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: AIStats. pp. 315–323 (2011)
11. Guo, H., TANG, R., Ye, Y., Li, Z., He, X.: Deepfm: A factorization-machine based neural network for ctr prediction. In: IJCAI. pp. 1725–1731 (2017)
12. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR. pp. 770–778 (2016)
13. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: CVPR. pp. 2261–2269 (2017)
14. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: ICML. pp. 448–456 (2015)
15. Karpathy, A., Johnson, J., Fei-Fei, L.: Visualizing and understanding recurrent networks. arXiv preprint arXiv:1506.02078 (2015)
16. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
17. Li, J., König, A.C., Narasayya, V., Chaudhuri, S.: Robust estimation of resource consumption for sql queries using statistical techniques. PVLDB **5**(11), 1555–1566 (2012)
18. Mikolov, T., Deoras, A., Povey, D., Burget, L., Černockỳ, J.: Strategies for training large scale neural network language models. In: ASRU Workshop. pp. 196–201 (2011)
19. Mishra, C., Koudas, N.: The design of a query monitoring system. TODS **34**(1), 1 (2009)
20. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: ICML. pp. 807–814 (2010)
21. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: NIPS. pp. 3104–3112 (2014)
22. Tompson, J.J., Jain, A., LeCun, Y., Bregler, C.: Joint training of a convolutional network and a graphical model for human pose estimation. In: NIPS. pp. 1799–1807 (2014)
23. Tozer, S., Brecht, T., Aboulnaga, A.: Q-cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In: ICDE. pp. 397–408 (2010)
24. Van Wouw, S.: Performance evaluation of distributed sql query engines and query time predictors (2014)
25. Wasserman, T.J., Martin, P., Skillicorn, D.B., Rizvi, H.: Developing a characterization of business intelligence workloads for sizing new database systems. In: DOLAP. pp. 7–13 (2004)
26. Wu, W., Chi, Y., Hacígümüş, H., Naughton, J.F.: Towards predicting query execution time for concurrent and dynamic database workloads. PVLDB **6**(10), 925–936 (2013)
27. Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., Naughton, J.F.: Predicting query execution time: Are optimizer cost models really unusable? In: ICDE. pp. 1081–1092 (2013)