Learning State Representations for Query Optimization with Deep Reinforcement Learning

Jennifer Ortiz†, Magdalena Balazinska†, Johannes Gehrke‡, S. Sathiya Keerthi†

University of Washington†, Microsoft‡, Criteo Research†

ABSTRACT

We explore the idea of training a deep learning model to predict query cardinalities. Instead of relying entirely on basic statistics and formulas to estimate cardinalities, we train a model to automatically learn important properties of the data to more accurately infer these estimates. Importantly, we use that model to learn subquery representations that can serve to derive the representation, and cardinality, of more complex queries and that can serve to build query plans bottom-up using deep reinforcement learning. As of today, there are few studies that have used deep learning techniques to solve database problems, although some have started to raise awareness for the potential of this method in our field [14]. Now is the time to explore this space, since we have the computational capabilities to run these models. In particular, we make two contributions in this paper: A succinct way of representing the state of a table (or subquery) using deep learning, and a principled way to enumerate plans for a given query using those states together with reinforcement learning.

State Representation. A key challenge of this approach is how to represent queries and data. As a first contribution, we develop an approach that learns to incrementally generate a succinct representation of each subquery’s intermediate results: The model takes as input a subquery and a new operation to predict the resulting subquery’s representation. This representation can serve to derive the subquery’s cardinality (Section 3).

Query Plan Enumeration. We also present an initial approach to using these representations to enumerate query plans through reinforcement learning. Reinforcement learning is a general purpose framework used for decision-making in contexts where a system learns by trial and error from rewards and punishments. We propose to use this approach to incrementally build a query plan by modeling it as a Markov process, where each decision is based on the properties of each state. Figure 1 illustrates our approach to query optimization. Given a query and a database, the model incrementally builds a query plan through a series of state transitions. In the initial state $t$ in the figure, the system begins with a representation of the entire database. Given an action selected using reinforcement learning, the model transitions to a new state at $t + 1$, having now constructed a larger subquery. Each action represents a query operation and each state captures a representation of the subquery’s intermediate results. We train a state transition function (a neural network), $NN_{ST}$, to generate this representation. $NN_{ST}$ is a recursive function that takes as input a previous subquery representation as well as an action at time $t$, to produce the subquery representation for time $t + 1$.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:


1 INTRODUCTION

Query optimization is not a solved problem, and existing database management systems (DBMSs) still choose poor execution plans for some queries [5]. Because query optimization must be efficient in time and resources, existing DBMSs implement a key step of cardinality estimation by making simplifying assumptions about the data (e.g., inclusion principle, uniformity or independence assumptions). When these assumptions do not hold, cardinality estimation errors occur, leading to sub-optimal plan selections [5].

Recently, thanks to dropping hardware costs and growing datasets available for training, deep learning has successfully been applied to solving computationally intensive learning tasks in other domains. The advantage of these types of models comes from their ability to learn unique patterns and features of the data that are difficult to manually find or design [3].

In this paper, we explore the idea of training a deep learning model to predict query cardinalities. Instead of relying entirely on basic statistics and formulas to estimate cardinalities, we train a model to automatically learn important properties of the data to more accurately infer these estimates. Importantly, we use that model to learn subquery representations that can serve to derive the representation, and cardinality, of more complex queries and that can serve to build query plans bottom-up using deep reinforcement learning. As of
2 BACKGROUND

Deep Learning Deep learning models, also known as feedforward neural networks, are able to approximate a non-linear function, \( f \) [3]. These models define a mapping from an input \( x \) to an output \( y \), through a set of learned parameters across several layers, \( \theta \). During training, the behavior of the inner layers are not defined by the input data, instead these models must learn how to use the layers to produce the correct output. Since there is no direct interaction between the layers and the input training data, these layers are called hidden layers [3].

These feedforward networks are critical in the context of representation learning. While training to meet some objective function, a neural network’s hidden layers can indirectly learn a representation, which could then be used for other tasks [3]. There is a trade-off between preserving as much information as possible and learning useful properties about the data. Depending on the output of the network, the context of these representations can vary [3].

Reinforcement Learning Reinforcement learning models are able to map scenarios to appropriate actions, with the goal of maximizing a cumulative reward. Unlike supervised learning, the learner (the agent) is not explicitly shown which action is best. Instead, the agent must discover the best action through trial and error by either exploiting current knowledge or exploring unknown states [11]. At each timestep, \( t \), the agent will observe a state of the environment, \( s_t \), and will select an action, \( a_t \). The action selected depends on the policy, \( \pi \). This policy can reenact several types of behaviors. As an example, it can either act greedily or balance exploration and exploitation through an \( \epsilon \)-greedy (or better) approach. The policy is driven by the expected rewards of each state, which the model must learn. Given the action selected, the model will arrive at a new state, \( s_{t+1} \). The environment then sends the agent a reward, \( r_{t+1} \), signaling the “goodness” of the action selected. The agent’s goal is to maximize this total reward [11]. One approach is to use a value-based iteration technique, where the model records state-action values, \( Q(s, a) \). These values specify the long-term desirability of the state by taking into account the rewards for the states that are likely to follow [11].

3 LEARNING A QUERY REPRESENTATION

Given as input a database \( D \) and a query \( Q \), the first component of our approach is to apply deep learning to derive compact, yet informative representations of queries and the relations they produce. To ensure that these representations are informative, we focus on training these representations to predict subquery cardinalities.

3.1 Approach

There are two approaches that we could take. In the first approach, we could transform \((Q, D)\) into a feature vector and train a deep network to take such vectors as input and output a cardinality value. As discussed in the introduction, the problem with this approach is that the size of the feature vector would have to grow with the complexity of databases and queries. This would result in very long, sparse vectors, which would require large training datasets.

Instead, we take a different approach, a recursive approach: We train a model to predict the cardinality of a query consisting of a single relational operation applied to a subquery as illustrated in Figure 2. This model takes as input a pair \((h_t, a_t)\), where \( h_t \) is a vector representation of a subquery, while \( a_t \) is a single relational operation on \( h_t \). Importantly, \( h_t \) is not a manually specified feature vector, but it is the latent representation that the model learns itself. The \( NN_{ST} \) function generates these representations by adjusting the weights based on feedback from the \( NN_{Observed} \) function. This \( NN_{Observed} \) function learns to map a subquery representation to predict a set of observed variables. As we train this model, we use back propagation to adjust the weights for both functions. In this
work, we only focus on predicting cardinalities, but we could extend the model to learn representations that enable us to capture additional properties such as more detailed value distributions or features of query execution plans, such as their memory footprint or runtime.

Before using the recursive \( NN_{ST} \) model, we must learn an additional function, \( NN_{init} \), as shown in Figure 3. \( NN_{init} \) takes as input \((x_0, a_0)\), where \( x_0 \) is a vector that captures the properties of the database \( D \) and \( a_0 \) is a single relational operator. The model outputs the cardinality of the subquery that executes the operation encoded in \( a_0 \) on \( D \). We define the vector, \( x_0 \) to represent simple properties of the database, \( D \). The list of properties we provide next is not definitive and more features can certainly be added. Currently, for each attribute in the dataset \( D \), we use the following features to define \( x_0 \): the \text{min} value, the \text{max} value, the number of distinct values, and a representation of a 1-D equi-width histogram.

As shown in the figure, we then include the recursive model, \( NN_{ST} \), that takes \((h_t, a_t)\) as input and predicts the observed variables of the subqueries as well as the representation, \( h_{t+1} \) of the new subquery. We combine these models to train them together. During training, the weights are adjusted based on the combined loss from observed variable predictions. We want to learn an \( h_t \) representation that captures not only enough information to predict the cardinality of that subquery but of other subqueries built by extending it.

3.2 Preliminary Results

We use the publicly available Internet Movie Data Base (IMDB) data set from the Join Order Benchmark (JOB) [5]. Unlike TPC-H and TPC-DS, the IMDB data set is real and includes skew and correlations across columns [5]. In our experiments, we use Python Tensorflow to implement our approach [1].

Training \( NN_{init} \): As a first experiment, we initialize \( x_0 \) with properties of the IMDB dataset and train \( NN_{init} \) to learn \( h_1 \). \( a_0 \) represents a conjunctive selection operation over \( m \) attributes from the \textit{aka\_title} relation. We generate 20k unique queries, where 15k are used for training the model and the rest are used for testing. \( NN_{init} \) contains 50 hidden nodes in the hidden layer. We update the model via stochastic gradient descent with a loss based on relative error and a learning rate of .01.

In Figure 4a, we show the cardinality estimation results for selection queries where \( m = 3 \). On the x-axis, we show the number of epochs used during training and on the y-axis we show the relative error with the error bars representing the standard deviation. We compare our approach \( NN_{model} \) to estimates from a popular commercial DBMS. We use a commercial engine to ensure a strong baseline. With fewer epochs (less training) the \( NN_{init} \)’s cardinality predictions result in significant errors, but at the 6th epoch, the model performs similarly to the commercial system and then it starts to outperform it. We have also observed that increasing the number of training examples reduces the error variance.

In Figure 4b, we increase the number of columns in the selection to \( m = 5 \). In general, we observe that \( NN_{init} \) takes longer to converge once more columns are introduced. This is expected, as \( NN_{init} \) must learn about more joint distributions across more columns. Nevertheless, the model still manages to improve on the commercial engine’s estimates by the 9th epoch.

Training \( NN_{init} \) and \( NN_{ST} \): In the previous experiment, we only trained the \( NN_{init} \) model for selection queries over base data. For this next experiment, we predict the cardinality of a query containing both a selection and join operation by using the combined model. Here, \( a_0 \) represents the selection, while the subsequent action \( a_1 \) represents the join. Through this combined model, we can ensure that \( h_1 \) (the hidden state for the selection) captures enough information to be able to predict the cardinality after the join. In Figure 5, we show the cardinality prediction for \( h_1 \) and \( h_2 \). In these scatter plots, the x-axis shows the real cardinality, while the y-axis shows the predicted cardinality from the model. Although there is some variance, \( h_1 \) was able to hold enough information about the underlying data to make reasonable predictions for \( h_2 \).

Training this model takes ~535 seconds for 100 epochs. Cardinality predictions for each query takes ~0.004 seconds on average.

4 QUERY PLAN ENUMERATION WITH REINFORCEMENT LEARNING

In this section, we present and discuss our design to leverage the subquery representations from the section above, not only to estimate cardinalities, but to build query plans. Given a query, \( Q \), we seek to identify a good query plan by combining our query representations from \( NN_{ST} \) with reinforcement learning.

We assume a model-free environment, where transition probabilities between states are not known. At \( s_0 \), the model only knows
about $D$ and $Q$. No query operations have yet taken place. The agent transitions to a new state by selecting an operation from query $Q$. At each state, we encode an additional contextual vector, $s_t$, which expresses the operations that remain to be done for $Q$. We now describe how to initialize the vector $s_0$ at time 0:

Given database $D$, we have a set of $n$ relations $\mathcal{R} = \{\text{rel}_1, ..., \text{rel}_n\}$, where each $\text{rel}$ contains a set of $m$ attributes $\{\text{att}_1, ..., \text{att}_m\}$. The vector $u_t$ represents a fixed set of equi-join predicates and one-dimensional selection predicates, $\mathcal{C} = \{c_1, ..., c_p\}$. We set the $i$-th coordinate in $C$ accordingly if the corresponding predicate exists in the query $Q$. For example, $c_1$ could represent the following equi-join predicate, $\text{rel}_1.\text{att}_1 = \text{rel}_2.\text{att}_2$. If this predicate exists in $Q$ we encode it in $u_t$ by updating the value of $c_1$ to 1, otherwise we set it to 0. For selections, we track one-dimensional ranged selections of the following form: $\text{rel}_i.\text{att}_j <= v$. For now, we allow each attribute to have at most one ranged filter in $Q$. If the selection predicate exists in $Q$, we place the value $v$ in the corresponding element in $C$. Once the agent selects an action (query operation), $a_t$, we can update $u_t$ by setting the corresponding element to 0.

To select good query plans, we need to provide the model with a reward. This reward must either be given at each state or once the entire query plan has been constructed. Currently, we are using the negative of our system’s cardinality estimates at each step. The limitation is that this approach only optimizes logical query plans. We plan to extend the reward function in future work to also capture QL by setting the corresponding element to 0.

Initially, all state-action pairs are random values. At each timestep, the agent selects an action and observes the reward, $r_{t+1}$ at state $s_{t+1}$. As the agent explores, these state-action pairs will converge to represent the expected reward of the states in future timesteps. At each state transition, each $QL(s, a)$ is updated as follows:

$$QL(s_t, a_t) \leftarrow QL(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_{a'} QL(s_{t+1}, a') - QL(s_t, a_t)]$$

Where the $\max_{a'} QL(s_{t+1}, a')$ represents the maximum value from $s_{t+1}$ given the target policy. We compute the subsequent state given the state transition function, $NS_T$.

**Open Problems:** Many open problems remain for the above design. As we indicated above, the first open problem is the choice of reward function and its impact on query plan selection. Another open problem is that the state-space is large even when we only consider selections and join operators as possible actions. Thus, the Q-learning algorithm as initially described is impractical as the state-action values are estimated separately for each unique subquery. In other words, for each query that we train, it is unlikely that we will run into the same exact series of states for a separate query. Thus, a better approach is to consider approximate solutions to find values for $QL(s, a)$. We can learn a function, $\hat{QL}(s, a, w)$ to approximate $QL(s, a)$ given parameter weights $w$. This allows the model to generalize the value of a state-action pairs given previous experience with different (but similar) states.

### 5 RELATED WORK

Eddies [2] gets rid of the optimizer altogether and instead of building query plans, uses an eddy to determine the sequence of operators based on a policy. Tzoumas et al. [13] took this a step further and transformed it into a reinforcement learning problem where each state represents a tuple along with metadata about which operators still need to be applied and each action represents which operator to run next. Since the eddies framework does not build traditional query plans, their work does not consider learning representations of intermediate relations. Leo [10], was one of the first approaches to automatically adjust an optimizer’s estimates based on past mistakes. This requires successive runs of similar queries to make adjustments. Liu et al. [6] use neural networks to solve the cardinality estimation problem, but focus on selection queries only. Work by Kraska et al. [4] uses a mixture of neural networks to learn the distribution of an attribute to build fast indexes. Our goal is to learn the correlation across several columns and to build query plans. Work by Marcus et al. [7] also uses a deep reinforcement learning technique to determine join order for a fixed database. Each state also represents a subquery, but our approach models each state as a latent vector that is learned through a neural network and is propagated to other subsequent states. Their approach uses a policy gradient to determine the best action, while our technique proposes to use a value-based iteration approach.

### 6 CONCLUSION

In this work, we described a model that uses deep reinforcement learning for query optimization. By encoding basic information about the data, we use deep neural networks to incrementally learn state representations of subqueries. As future work, we propose to use these state representations in conjunction with a reinforcement learning model to learn optimal plans.

**Acknowledgements** This project was supported in part by NSF grant IIS-1247469 and Teradata.

**REFERENCES**