Plan-Structured Deep Neural Network Models for Query Performance Prediction

Ryan Marcus
Brandeis University
ryan@cs.brandeis.edu

Olga Papaemmanouil
Brandeis University
olga@cs.brandeis.edu

ABSTRACT

Query performance prediction, the task of predicting a query's latency prior to execution, is a challenging problem in database management systems. Existing approaches rely on features and performance models engineered by human experts, but often fail to capture the complex interactions between query operators and input relations, and generally do not adapt naturally to workload characteristics and patterns in query execution plans. In this paper, we argue that *deep learning* can be applied to the query performance prediction problem, and we introduce a novel neural network architecture for the task: a *plan-structured neural network*. Our neural network architecture matches the structure of any optimizer-selected query execution plan and predict its latency with high accuracy, while eliminating the need for human-crafted input features. A number of optimizations are also proposed to reduce training overhead without sacrificing effectiveness. We evaluated our techniques on various workloads and we demonstrate that our approach can outperform the state-of-the-art in query performance prediction.

PVLDB Reference Format:
Ryan Marcus, Olga Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*, 12(11): 1733-1746, 2019.
DOI: https://doi.org/10.14778/3342263.3342646

1. INTRODUCTION

Query performance prediction (QPP), the task of predicting the latency of a query, is an important primitive for a wide variety of data management tasks, including admission control [67], resource management [64], and maintaining SLAs [13, 40]. QPP is also notoriously difficult, as the latency of a query is highly dependent on a number of factors, such as the execution plan chosen, the underlying data distribution, and the resource availability. As database management systems and their supported applications become increasingly complex, query performance prediction only gets more difficult: the wide diversity of applications often leads to variable and complex query plans. Additionally, new query operators (and their new physical implementations) can introduce novel and complex interactions with the rest of the operators in a plan.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342646

Prior approaches have focused on hand-designing new predictive measures (e.g., [14, 15]), manually deriving statistically useful combinations of query plan properties (e.g., [70, 73]), proposing mathematical models of relational operators (e.g., [33]), or combining plan-level and operator-level information in ad-hoc ways (e.g., [7]). All of these methods heavily rely on humans to analyze available query properties and combine them into more complex *predictive metrics* that correlate with query latency and are useful to a statistical prediction model. This manual process generally requires significant effort from human experts, but, more importantly, scales poorly with the increasing complexity of the DBMS, as humans (even the most experienced database engineers) can rarely manually derive combined features that effectively capture the complex interactions between query operators/plans and query performance.

In this paper, we present a class of deep neural networks (DNNs) capable of performing query performance prediction while significantly reducing human effort. DNNs have shown tremendous performance on many machine learning tasks [61]. Unlike most traditional machine learning techniques, DNNs learn in an *end-to-end* fashion, requiring no human feature engineering beyond the architecture of the network and the selection of its inputs (e.g., query plan properties) [30]. During training, neural networks *automatically* derive and invent complex combinations of their inputs that are useful for their specific prediction task. Thus, in the context of query performance prediction, DNNs have the potential to alleviate the need for ad-hoc and human-derived models of relational operators and their combinatorial interactions.

Despite the deep learning's track record and massive growth, applying deep learning to query performance prediction is not a straight-forward task. DNNs, like many other machine learning algorithms, are designed to map input vectors to output vectors. However, to act as input to a DNN, query plans need to be carefully vectorized to allow the DNN to capture performance-related properties. Most importantly, this vectorization process needs to account for the tree-based structure of execution plans, as the plan's structure contains valuable information, including the processing flow of operators and interesting data properties being passed from downstream to upstream operators (e.g., sorting or materialization of an operator's output). Specifically, we note that query execution plans have *heterogeneous tree nodes*: different operators have different properties (e.g., number of children, predicates). This makes execution plans a unique structure and hence existing works applying neural networks to homogeneous tree structured data [48, 65] are ill-suited for the query performance prediction task.

We thus propose a novel deep neural network architecture, a *plan-structured neural network*, specifically crafted for predicting the latency of query execution plans in relational DBMSs. Critically, our structure uses a unique *neural unit*, a small neural net-

work, for each logical operator supported by a DBMS' execution engine. These neural units can model the latency of an operator while emitting "interesting" features to any subsequent operators in the plan. These neural units can be combined together into a tree shape isomorphic to the structure of a given execution plan, creating a single neural network which maps a query execution plan directly to a latency. By exploiting *weight sharing* [29], the property where the same neural unit is used for any instance of the same operator across and within plans, these neural units are capable of learning complex interactions between operators.

While performance prediction has been studied extensively [7, 15, 33, 70], our approach does not depend on *human-engineered* predictive metrics: instead of manually deriving statistically-useful combinations from query plans properties, our approach relies on DNNs to *automatically* identify complex combinations of these properties that can act as effective performance predictors. We argue that this is critical for a query performance prediction system to "keep up" with increasingly complex DBMSs. While this work aims to address this challenge, we do assume that (1) query operator latencies are measurable and (2) the query execution plan is not altered at runtime, which excludes approaches such as runtime query compilation [51], JIT compilation [24], and adaptive query processing [8]. However, we *do* account for multiple queries running in parallel, and for operator-level parallelism, such as the PostgreSQL "gather" operator, parallel hash joins, and parallel aggregates. Common database management systems, such as MySQL [1], PostgreSQL [2], and SQLite [3], meet these requirements.

This paper makes the following contributions:

- We introduce the notion of an *operator-level neural unit,* a deep neural network that models the behavior of a logical relational operator (e.g., join, selection, aggregate). Neural units are designed to produce (a) latency estimates and (b) performance related features that might be useful for the latency prediction of their parent operator within the query execution plan.
- We introduce *plan-structured deep neural networks*, a neural network model specifically designed to predict the latency of query execution plans by dynamically assembling neural units in a network isomorphic to a given query plan.
- We present experimental results demonstrating that our approach outperforms state-of-the-art techniques.

In the next section, we provide background information about neural networks. Section 3 outlines unique properties of query execution plans that motivate our new neural network architecture. Section 4 describes our plan-structured neural network model and how it can be applied to query execution plans. In Section 5, we describe how well-known training optimizations can be applied to plan-structured neural networks. We present experimental results in Section 6, describe related work in Section 7, and offer concluding remarks in Section 9.

2. NEURAL NETWORKS BACKGROUND

This section reviews DNNs and gradient descent, a method for training neural networks. See [37] for a more detailed discussion.

A DNN model consists of *layers*. The first layer takes in a vector representing the input data, and subsequent layers apply an activated affine transformation to the previous layer's output. Given an input vector $\vec{x}$ of size $n \times 1$, the i-th layer of a network $t_i(\vec{x})$ provides an output vector of size m:

$$t_i(\vec{x}) = S\left(W_i \times \vec{x} + \vec{b_i}\right) \tag{1}$$

where S is the activation function (a non-linearity such as a sigmoid function or a rectified linear function [19]) and W_i are the *weights* for the i-th layer, represented by a matrix of size $m \times n$. The bias, $\vec{b_i}$, is an $m \times 1$ vector representing the constant shift of an affine transform. Together, the weights and the biases represent the *parameters* of the network, controlling the transformation performed at each layer. DNNs end in an *output layer* responsible for mapping the output of the penultimate layer to a prediction.

Neural network layers are composed together by feeding the output of one layer into the input of the next. For a neural network with n layers, where $\circ$ represents the function composition operator, a neural network can be defined as: $N(\vec{x}) = t_n \circ t_{n-1} \circ \ldots t_1$.

DNNs are trained on a *dataset*, pairs of *inputs* and *targets*, and aim to learn an accurate mapping from a given input to the correct target. The quality of this mapping is measured by a *loss function*, which quantifies the difference between the neural network's prediction (output) and the ground truth (target). For example, let X be a set of vectors representing query plans, and $l(\vec{x})$ be the latency of the query plan represented by the vector $\vec{x} \in X$. The neural network N can be trained to produce the target $l(\vec{x})$ when fed $\vec{x}$ by minimizing a loss function [59]. One popular loss function is L_2 loss, or root mean squared error, which can be defined as:

$$err(X) = \sqrt{\frac{1}{|X|}\sum_{\vec{x} \in X}\left(N(\vec{x}) - l(\vec{x})\right)^2} \tag{2}$$

DNNs learn via a process called *gradient descent*, a method that incrementally adjusts the transformation performed by each layer (i.e., the weights) to minimize the loss function. Gradient descent works by adjusting each weight of the neural network independently. First, we compute the derivative of a given neural network parameter, w, with respect to the loss function. If the derivative is positive, meaning that an increase in this parameter would (locally) lead to a increase in the loss function, the weight w is decreased. If the derivative is negative, meaning that an increase in this parameter would (locally) lead to a decrease in the loss function, the weight w is increased. After adjusting the weight, the algorithm then repeats this procedure for each parameter in the network. This simple procedure is iterated until gradients of all parameters (weight and biases) are relatively flat (i.e., convergence). Training a DNN can be seen as a corrective feedback loop, rewarding weights that support correct guesses, punishing weights that lead to errors, and slowly pushing the loss function towards smaller and smaller values. In the process, the network takes advantage of correlations and patterns in the underlying data, creating new, transformed representations of the data [30].

3. CHALLENGES

Despite their advantages, there are numerous challenges in applying deep neural networks to query performance prediction. A straightforward application of deep learning would be to model the whole query as a single neural network and use information from the query plan as the input vector. However, this naive approach ignores the fact that the query plan structure, features of intermediate results, and non-leaf operators often impact query execution times and hence can be useful in any predictive analysis task.

Furthermore, query plans are diverse structures – the type and number of operators varies per plan, operators have different correlations with query performance, and operators have different sets of properties and hence different sets of predictive features. Traditional DNNs have static network architectures and deal with input vectors of fixed size. Hence, "one-size-fits-all" neural network

architectures do not fit the query performance prediction task. Finally, while previous work in the field of machine learning has examined applying deep neural networks to sequential [20] or tree-structured [58, 65] data, none of these approaches are ideal for query performance prediction, as we describe next.

Heterogeneous tree nodes: Traditional neural networks operate on input vectors of a fixed structure. However, in a query execution plan, each type of operator has fundamentally different properties. A join operator may be described by the join type (e.g., nested loop join, hash join), the estimated required storage (e.g., for an external sort), or other properties specific to the underlying execution engine. A filter operation, however, will have an entirely different set of properties, such as selectivity estimation or parallelism flags. Since feature vectors of different operators are likely of different sizes and different semantic meaning, simply feeding them into the *same* neural network is not possible.

A naive solution to this problem might be to concatenate vectors together for each relational operator. For example, if a join operator has 9 properties and a filter operator has 7 properties, one could represent either a join or a filter operator with a vector of size 16 properties. If the operator is a filter, the first 9 entries of the vector are simply 0, and the last 7 entries of the vector are populated. If the operator is a join, the first 9 entries of the vector are populated and last 7 entries are empty. The problem with this solution is *sparsity*: if one has many different operator types, the concatenated vectors will have an increasingly larger proportion of zeros. Generally speaking, such sparsity represents a major problem for statistical techniques [32], and transforming sparse input into usable, dense input is still an active area of research [68, 69]. In other words, using sparse vectors to overcome heterogeneous tree nodes replaces one problem with a potentially harder problem.

Position-independent operator behavior: As pointed out by previous work [33, 73], two instances of the same operator (e.g., join, selection), will share similar performance characteristics, even when appearing within different plans or multiple times in the same plan. For example, in the case of a hash join, latency is strongly correlated with the size of the probe and build relations, and this correlation holds *regardless of the operator's position in the query execution plan.* Since other join algorithms exhibit similar correlations, one could potentially train a neural network model to predict the performance of a join operator, and that same model can be used any time the join operator appears in a plan.

4. PLAN-STRUCTURED DNNS

This paper proposes a new tree-structured neural network architecture, in which the structure of the network matches the structure of a given query plan. This *plan-structured neural network* consists of operator-level neural networks (called *neural units*), and the entire query plan is modeled as a tree of neural units. On its own, each neural unit is expected to predict (1) the performance of an individual operator type – for example, the neural unit corresponding to a join predicts the latency of joins – as well as (2) "interesting" data regarding the operator that could be useful to the parent of the neural unit. The plan-level neural network is expected to predict the execution time of a given query plan. We begin by discussing the operator-level neural units, and then move on to the plan-structured neural network architecture.

4.1 Operator-level neural units

We model each logic operator type supported by a DBMS' execution engine with a unique neural unit, responsible for learning the performance of that particular operator type (e.g., a unique unit for joins, a unique unit for aggregates). These neural units aim to represent sufficiently complex functions to model the performance of relational operators in a variety of contexts. For example, while a simple polynomial model of a join operator makes predictions based only on estimated input cardinalities, our neural units will automatically identify the most relevant features out of a wide number of candidate inputs (e.g., underlying structure of the table, statistics about the data distribution, uncertainty in selectivity estimates, available buffer space), all without any hand-tuning.

Input feature vectors: We define as $\vec{x} = F(x)$ a vector representation describing an instance of a relational operator x. This vector will act as an input to the neural unit for that particular operator. These vectors could be extracted from the output of the query optimizer, and contain information such as the type of operator (e.g., hash join or nested loop join for join operators), the estimated number of rows to be produced, or the estimated number of I/Os required. Many DBMSs expose this information through convenient APIs, such as `EXPLAIN` queries. For example, the appendix lists the features used in our experimental study for several operators. Note that the size of the input vector may vary based on the corresponding operator: input vectors for join operators have a different set of properties and thus different sizes than the input vectors for selection operators. We assume every *instance* of a relational operator of a given type will have the same size input vector (e.g., all join operators have the same size input vectors).

Output vector: Performance information for an operator instance x is often relevant to the performance of x's parent. To capture this, and allow for flow of information between neural units, each neural unit outputs both a latency prediction and a *data vector.* While the latency output predicts the operator's latency, the output data vector represents "interesting" features that are relevant to the performance of the parent operator. For example, a neural unit for a scan operator may produce a data vector that contains information about the expected distribution of produced rows. Much like the internal representation of an auto-encoder [37], these data vectors are learned *automatically* during training, without any human interference or selection of the features that appear in the output vector.

Neural units: Next, we define a *neural unit* as a neural network N_A, with A representing a type of relational operator (e.g., $N_{\bowtie}$ is the neural unit for join operators). For each instance a of the operator type A in a given query plan, the neural unit N_A takes as input the vector representation of the operator instance a, $\vec{x_a}$.

This input is fed through a number of hidden layers, with each hidden layer generating features by applying an activated affine transformation (Equation 1). These complex transformations can be learned *automatically* using gradient descent methods, which gradually adjusts the weights and bias of the neural unit N_A in order to minimize its loss function. The last layer transforms the internal representation learned by the hidden layers into a latency prediction and an output data vector.

Formally, the output of a neural unit N_A is defined as:

$$\vec{p_a} = N_A\left(\vec{a}\right), \text{ when } a \text{ is a leaf} \tag{3}$$

where a is the instance of the operator type A. The output vector has a size of $d+1$. The first element of the output vector represents the neural unit's estimation of the operator's latency, denoted as $\vec{p_a}[l]$. The remaining d elements represent the data vector, denoted as $\vec{p_a}[d]$. We note that since the input vectors to different neural units will not have the same size, each neural unit may have different sizes of weight and bias vectors that define the neural unit, but their fundamental structure will be similar.

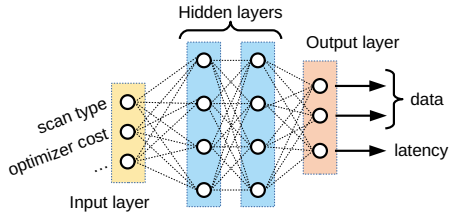


Figure 1: Neural unit corresponding to a scan operator, N_s.

4.1.1 *Leaf neural units (scan)*

The simplest neural units are those representing leaf nodes of the query plan tree and are responsible for accessing data from the database. Following PostgreSQL terminology, we refer to these as *scan* operators to distinguish them from the selection operator that filters out intermediate data.

For a given instance s of the scan operator type S, the neural unit N_S takes as input the raw vector representation of the operator instance s, $\vec{x_s}$, and produces an output vector $\vec{p_s}$. Since these operators access stored data, their corresponding vectors include (among others) information about the input relation of the scan operator. This information is collected through the optimizer (or various system calls). We refer to the function collecting this information for an instance a of the operator A as $F(a)$.

Figure 1 shows an illustration of a neural unit for a scan operator N_S. The unit takes information from the query plan (e.g., index/table scan, optimizer's cost, cardinality estimates, estimated I/Os) as input. By running the raw vector representation of a scan operator through many successive hidden layers, the neural unit can model complex interactions between the inputs. For example, a series of activated affine transformations might "trust" the optimizer's cost model more for certain types of scans, or for scans over particular relations. The neural unit transforms the input vector into a latency prediction and a output data vector. This output data vector could, for example, contain features related to the distribution of the rows emitted by the scan.

4.1.2 *Internal neural units*

Having constructed neural units for each leaf operator type, we next explain how the internal operators (i.e., operators with children in a query execution plan) can be modeled using neural units. Like leaf operators, the neural units for the internal operator instance x of the query execution plan will take an operator-specific input vector, provided by the function $F(x)$. However, the performance of the internal operators depends also on the behavior of its children. Hence, each internal neural unit receives also as input the latency prediction and the output data vector of its children.

A neural unit for an internal operator type A is a neural network N_A. Given a query execution plan where an operator instance a of type A receives input from operators $x_i, i \in \{1, ...n\}$ (i.e., a is the parent operator of each such x_i), the input vector of N_A will contain the operator-related information produced by $F(a)$ in addition to *the output vectors produced by the operator's children*:

$$\vec{p_a} = N_A(F(a) \frown \vec{p_{x_i}} \frown \cdots \frown \vec{p_{x_n}}) \quad (4)$$

where $\frown$ represents the vector concatenation operator.

Figure 2 shows an example of an internal neural unit, corresponding to a join operator, $N_{\bowtie}$. The unit takes information about the join operator itself (e.g., the type of join, the optimizer's predicted cost, cardinality estimates) as well as information from the join operator's children. Specifically, the join neural unit will receive both the data vector and latency output of its left and right

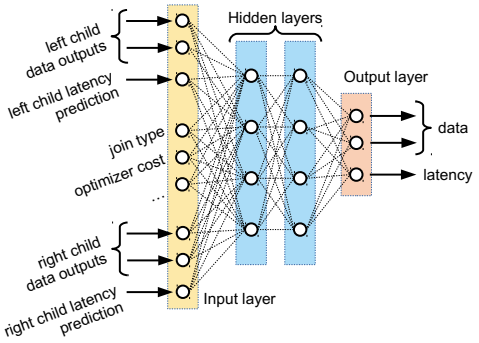


Figure 2: Neural unit corresponding to a join operator, $N_{\bowtie}$. The neural unit takes input from its two children (top and bottom), as well as information from the query execution plan (middle).

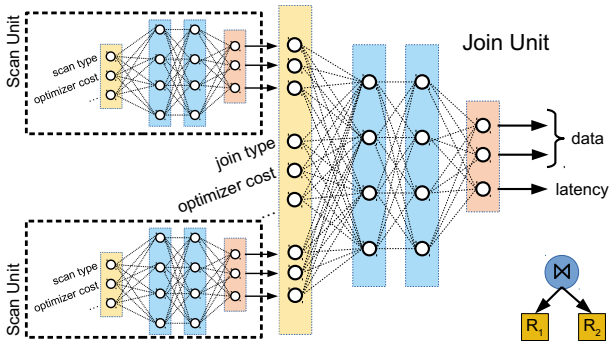


Figure 3: A neural network for a simple join query

child. These inputs are fed through a number of hidden layers and transformed into a final output vector, where the first element of the output vector represents the predicted latency and the remaining elements represent the data output features. This allows $N_{\bowtie}$ to be further composed with other neural units.

4.2 Trees of neural units

Next, we show how neural units can be composed into tree structures isomorphic to any particular query execution plan. Intuitively, each operator in a query execution plan is replaced with its corresponding neural unit (e.g., join operators are replaced with $N_{\bowtie}$), and the output of each neural unit is fed into the parent. The latency of the query execution plan is the first element of the output vector $\vec{p_r}$, where r is the instance operator on the root of the query execution plan. Note that the recursive definition (Equation 4) of $\vec{p_r}$ will "replace" each relational operator with its corresponding neural unit in a top-down fashion.

Figure 3 shows an example of this construction. For the query execution plan in the bottom-right of the figure (two scans and a join), two instances of the scan neural unit and one instance of the join neural unit are used. The outputs of the scan units are concatenated together with information about the join operator to make the input for the join unit, which produces the final latency prediction.

Figure 4 shows a more general example, with a query plan (top left) and the corresponding neural network tree (right). Each neural unit, represented as trapezoids, takes in a number of inputs. For the leaf units (orange, corresponding to the table scans in the query plan), the inputs are information from the query plan (black arrows). The internal, non-leaf units take information from the query plan as well, but additionally take in the latency output (green arrows) and the data outputs (red arrows) of their children. The latency outputs represent the model's estimate of the latency of each

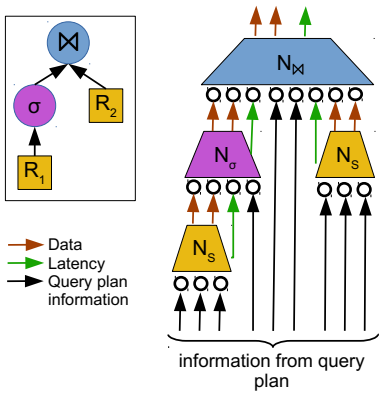


Figure 4: General neural network for latency prediction

operator, and the data outputs contain information about each operator that may be useful to the parent operator (for example, the table scan neural unit may encode data about which relation is being read). The two orange trapezoids correspond to the scan operators of R_1 and R_2 in the query plan, and thus use the same neural unit.

4.3 Model discussion

Next, we discuss the advantages and disadvantages of our plan-structured neural network model and we analyze the differences between our model and plan level models, which transform the *entire* query plan into a feature vector, and operator level models, which model each query operator independently.

Operator interactions: Complex interactions between query operators is a central challenge for query performance prediction systems. While some interactions are relatively simple (e.g., the latency of a join changing depends on the selectivity of its child filter operator), other interactions are extremely complex to analyze (e.g., the latency of pipelined overlapping query operators and their shared memory bandwidth). While every operator in a query plan tree *may* interact with any other operator, QPPNet leverages the fact that any relational operator in a query execution plan *does* affect the performance of *all* of its ancestors. Plan-level models [70, 73] ignore this structural information entirely, and operator-level models [7, 33] make a much stronger assumption: that an operator *only* interacts with its *direct* parent. Conversely, the way we assemble neural units into trees respects this property: each neural unit passes information upwards, and the model automatically *learns* which features should be passed up the tree, both to each operator's direct ancestors and beyond. Intuitively, this upwards-only communication policy directly encodes knowledge about the structure of the query execution plan into the network architecture.

Of course, query operators can interact with operators other than their ancestors. For example, two table scans may execute in parallel, competing for disk I/Os. A database engine may implement advanced techniques such as sideways information passing [22,62]. These types of interactions are unlikely to be captured by either our proposed neural network model *or* existing plan-level or operator-level models [7, 33, 70, 73]. However, our tree-structured neural network *does* capture interactions between children and their (non-immediate) ancestors, whereas plan-level and operator-level models cannot capture even these interactions.

Finally, our model can model plan parallelism. To use PostgreSQL as an example, multiple parallel workers are represented with a "gather" operator, which we model using a neural unit just like any other PostgreSQL operator. The neural unit representing the "gather" operator could learn an admittedly-simplistic interference model (e.g., if the underlying parallel tasks will compete for a resource) through the data output vectors. Intuitively, such a neural unit could minimally learn to report the maximum of its children's estimated latency.

Heterogeneous tree nodes: Operator-level neural unit accept input vectors of different size depending on the operator they model, while producing a fixed-sized output vector. This enables the structure of the plan-structured neural network to dynamically match any given query plan, thus making our model suitable to handle arbitrary plans. For example, regardless of if the child of a join operator is a filter (selection) or a scan, its child neural unit will produce a vector of a fixed size, allowing this output vector to be connected to the neural unit for a join operator.

Position-independent operator behavior: Since we expect a particular operator to have some common performance characteristics regardless of its position in the query execution plan, the same neural unit is used for every instance of a particular operator. Because the same query execution plan can contain multiple instances of the same operator type (e.g., multiple joins), our architecture can be considered a *recurrent neural network* [35], and as such benefits from *weight sharing* [29]: since instances of the same operators share similar properties, representing them with a single neural unit (and thus a a single set of weights and bias) is both efficient (fewer total weights) and effective (enforces a constraint). However, since *distinct* operator types are represented by different neural units (and do *not* share the same weights and bias), our approach can handle the heterogenous nature of the query execution plan operators.

5. MODEL TRAINING

So far, we have discussed how to assemble neural units into trees matching the structure of a given query execution plan. In this section, we describe how these plan-structured neural networks are trained. Training is the process of progressively refining the network's weights and bias (in our case, the weight and bias of the neural units included in a plan-structured neural network) to minimize the loss function using gradient descent (see Section 2).

Initially, each hidden layer, and the final latency and data vector output for each neural operator, will simply be a random activated affine transformation. In other words, the weights and bias that define the transformations (Equation 1) are initially picked randomly. Through repeated applications of stochastic gradient descent, these transformations are slowly tweaked to map their inputs slightly closer to the desired target outputs.

This training process is performed using a large corpus of executed query plans. Formally, for a dataset of executed query execution plans, let D be the set of all query operator instances within those plans. Then, for each query operator $o \in D$, let $l(o)$ be the latency of the operator. The neural units are trained by minimizing the following loss function:

$$L_2(D) = \sqrt{\frac{1}{|D|} \sum_{o \in D} \left(\vec{p_o}[l] - l(o)\right)^2} \tag{5}$$

where $\vec{p_o}[l]$ represents the latency output of the operator o's neural unit (the neural unit's prediction).

Note that if a particular query operator instance o is not a leaf in its query execution plan, the evaluation of its output $\vec{p_o}$ will involve multiple neural units, based on the recursive definition given by Equation 4. The loss function $L_2(D)$ thus represents the prediction accuracy of leaf operators, internal operators, *and* the root operator of a query execution plan. Minimizing this loss function thus minimizes the prediction error for *all* operators.

Intuitively, Equation 5 is simply the combined differences between the latency the model predicted for each operator and the

observed, "ground truth" latency. The loss function explicitly compares the predicted latency of a particular operator $\vec{p_o}[l]$ with the "ground truth" latency $l(op)$.

However, it is important to note that the loss function does *not* explicitly compare the data vector of the output $\vec{p_o}$ to any particular value. The stochastic gradient descent algorithm is thus free to tweak the transformations creating the output data vector to produce useful information for the parent neural unit consuming the data vector output. To exemplify this, consider evaluating the output $\vec{p_j}$ for a simple query plan involving the join j of two relation scans, s_1 and s_2, as illustrated in Figure 3. Following Equation 4, and defining as $N_{\bowtie}$ the neural unit for joins and N_S the neural unit for scans, we can expand the output vector $\vec{p_j}$ as follows:

$$\begin{aligned}\vec{p_j} &= N_{\bowtie}\Big(F(j) \frown \vec{p_{s_1}} \frown \vec{p_{s_2}}\Big) \\ &= N_{\bowtie}\Big(F(j) \frown N_S(F(s_1)) \frown N_S(F(s_2))\Big) \\ &= N_{\bowtie}\Big(F(j) \frown [\vec{p_{s_1}}[l] \frown \vec{p_{s_1}}[d]] \frown [\vec{p_{s_2}}[l] \frown \vec{p_{s_2}}[d]]\Big)\end{aligned}$$

where $\vec{p_s}[d]$ represents the output data vector for the neural unit corresponding to the operator s in the query execution plan.

Thus, the transformations producing the output data vector for each neural unit are adjusted by the gradient descent algorithm to minimize the latency prediction error of their *parents*. In this way, each neural unit can learn what information about its represented operator type is relevant to the performance of the parent operator *automatically*, without expert human analysis. Because the training process does not push output data vectors to represent any prespecified values, we refer to these values as *opaque*, as the exact semantics of the output data vector will vary significantly based on context, and may be difficult to interpret directly, as is generally the case with recurrent neural networks [35, 37].

5.1 Training optimizations

Training neural networks can be time consuming [37]. Thus, we use two optimizations which aim to improve the *performance* of computing the loss function of a plan-structured neural network (Equation 5) in the context of gradient descent. Section 5.1.1 explains how the loss function can be computed efficiently in a vectorized way using *bucketing*. Section 5.1.2 shows how computing the loss function can be accelerated by memoizing the gradient computation. We note that modern deep learning libraries [5, 54] have built-in infrastructure for these optimizations.

5.1.1 Batch training

Gradient descent minimizes a neural network's loss function by tweaking each weight by a small amount based on the gradient of that weight, but requires the entire dataset to fit in memory and are thus often *space* prohibitive. Thus, modern differentiable programming frameworks [5, 54] preform training in *batches*: the gradient is *estimated* using simple random samples (called batches or mini-batches) drawn from the data. This widely-adopted technique is called *stochastic gradient descent* [59]. Since each sample is selected at random, the estimation of the gradient is unbiased [12].

Modern neural network libraries take advantage of vectorization (applying mathematical operators to entire vectors simultaneously) to speed up computation. To do so, neural networks are assumed to have a fixed architecture: an architecture that does not change based on the particular input. By assuming a fixed architecture, libraries can assume that the symbolic gradient of each weight will be identical for each item in the batch (i.e., the derivative of any weight can be computed using the same sequence of mathematical operations), and thus their computation can be vectorized.

Stochastic gradient descent and vectorization work for neural networks where the structure of the network does not change based on the inputs. However, these optimization pose a challenge for our plan-structured neural network model: if two samples (i.e., query plans) in a batch have different tree structures, the symbolic derivative for a given weight will vary depending on the input sample, and thus the sequence of mathematical operations needed to compute the derivative of a given weight can differ.

One solution might be to group the training set into query execution plans with identical structure, and then use each group as a batch. However, most of the effectiveness of stochastic gradient descent depends on the batch being a true simple random sample [12]. By only creating training batches with identical query plans, each batch, and thus each estimation of the gradient, will become biased.

Bucketing: Luckily, we note that a standard bucketing technique [5] can be applied to allow our approach to benefit from vectorization. With bucketing, we construct large batches of randomly sampled query plans. Within each large batch B, we group together sets of query plans with identical structure into buckets. Formally, we partition B into equivalence classes $c_1, c_2, \ldots, c_n$ based on plan's tree structure, such that $\bigcup_{i=1}^{n} c_i = B$. Then, we compute:

$$\nabla_w(L_2, B) = \frac{1}{\sum_{i=1}^{n} |c_i|} \times \sum_{i=1}^{n} \left(\sum_{p \in c_i} \frac{\partial L_2(p)}{\partial w} \right)$$

The gradient is then efficiently estimated within each class, and the results are summed and normalized. This optimization works best when the number of equivalence classes is relatively small (e.g., many queries share similar plan structures). In the extreme case where *no* queries share the same plan structure, this optimization is equivalent to not performing any bucketing at all.

5.1.2 Information sharing in subtrees

Next, we discuss how to efficiently compute the loss function by exploiting the tree-structure architecture of our neural network. Let us assume that r of type R is the root operator of a query execution plan, and c is the sole child of that operator. When computing the loss function of the query execution plan's neural network, estimating the latency prediction error of the root, $(\vec{p_r}[l] - l(r))$ in Equation 5, requires us to compute the output vector of its child c, $\vec{p_c}$, as an intermediate value. This follows from the definition of $\vec{p_r}$ in Equation 4, based on which $\vec{p_r} = N_R(F(r) \frown \vec{p_c})$. Since computing the loss function will *also* require computing $\vec{p_c}$ (in the term $(\vec{p_c} - l(c))$), we can avoid a significant redundant computation by caching the value of $\vec{p_c}$, thus only computing it once.

More generally, this effectively amounts to memoization while traversing the tree in post-order. For an arbitrary root $r \in D$ of a query execution plan, we can compute the error of each neural unit in the plan's neural network rooted at r in a bottom-up fashion: first, for each leaf node $leaf$ in the tree rooted at r, compute and store the output the neural units corresponding to the leaf nodes, $p_{\vec{leaf}}$. Then, compute and sum the $(p_{\vec{leaf}}[l] - l(leaf))^2$ values, storing the result into a global accumulator variable. Once all the leaf nodes have been resolved in this way, repeat the process moving one level up the tree. When the root of the tree has been reached, the global accumulator value will contain the $(\vec{p_x}[l] - l(x))^2$ values for every operator x in the tree rooted at r. The global accumulator contains the sum of the squared differences between the predicted latency and the actual latency for every node in the tree. Applying this technique over every plan-structured neural network in D can greatly accelerate the computation of our loss function as defined in Equation 5.

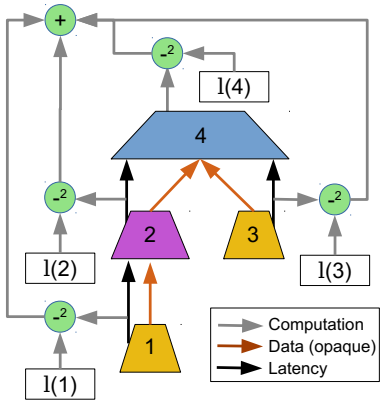


Figure 5: Efficiently computing the sum of losses for each neural unit in a tree. The "$-^2$" nodes represent computing the difference and then squaring the result.

We note that deep learning libraries support this optimization naively. For example, in PyTorch [54], the latency outputs of intermediary nodes can be effectively queried using `detach` function and the `requires_grad` parameter.

Example: Figure 5 shows a graphical representation of this process. First, we compute and cache the output of the neural unit labeled "1". We then add the squared difference between the outputted latency and the true latency, $l(1)$ into the global accumulator variable g. Next, we compute and cache the output of the neural unit labeled "2". Note that to compute the output of "2", we reuse the cached output value of "1". Then, we compute the squared difference between the latency predicted by "2" and $l(2)$, adding the result into the global accumulator. The same process is repeated for neural unit "3", which has no children. Finally, we compute the output of neural unit "4", which requires reusing the cached outputs from neural units "2" and "3". We then compute the squared difference between the latency predicted by neural unit "4" and $l(4)$, and add it to the global accumulator variable.

6. EXPERIMENTAL RESULTS

In this section, we describe the experimental study we conducted for our proposed plan-based neural network model. In all our experiments, our queries were executed with PostgreSQL 11.2 [2] (a row store) on a single node with an Intel Xeon CPU E5-2640 v4 processor, 32GB of RAM, and a solid-state drive. PostgreSQL was configured to use a maximum of 4 parallel workers.

Workload: We conducted experiments using TPC-H [57], a decision support benchmark, and TPC-DS [50], a decision support benchmark with a focus on more complex queries. All 22 TPC-H query templates were used, but only 69 TPC-DS query templates are compatible with PostgreSQL (without significant modification), hence we use only these templates for TPC-DS. For both benchmarks, 22K queries were executed with a scale factor of 100GB. Execution times and execution plans were recorded using PostgreSQL's `EXPLAIN ANALYZE` capability. The input features used for each neural unit are those that PostgreSQL makes available through the `EXPLAIN` command before a query is executed. See the appendix in the extended version of this paper [42] for a listing.

Training data: Queries are split into a training and testing sets in two different ways. For the TPC-DS queries, all of the instances of 10 randomly selected query templates are "held out" of the training set (the neural network trains on 59 query templates, and the performance of the network is measured on instances of the unseen 10 query templates). For the TPC-H queries, since there are not enough query templates to use the same strategy, 10% of the queries, selected randomly, are "held out" of the training set (the neural network trains on 90% of all query instances, and the performance of the network is measured on the other 10%). The dataset contained instances of 21 distinct query operators, with query plans containing between 6 and 91 operator instances.

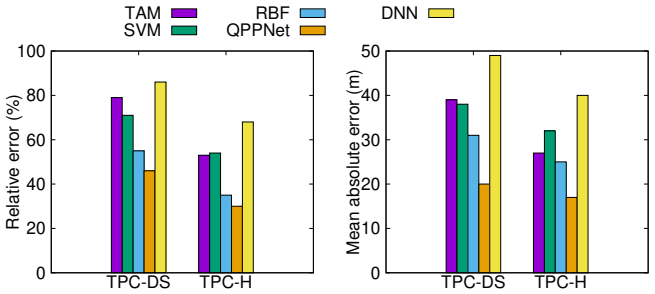
Neural networks: Unless otherwise stated, each neural unit had 5 hidden layers, each with 128 neurons . The data vector size was set to $d = 32$. Rectified linear units (ReLUs [19]) were used as activation functions. Training was conducted over 1000 epochs (full passes over the training queries). We used PyTorch [54] to implement our plan-structured neural networks.

Evaluation techniques: We compare our plan-based neural network model (*QPPNet*) with four other latency prediction approaches.

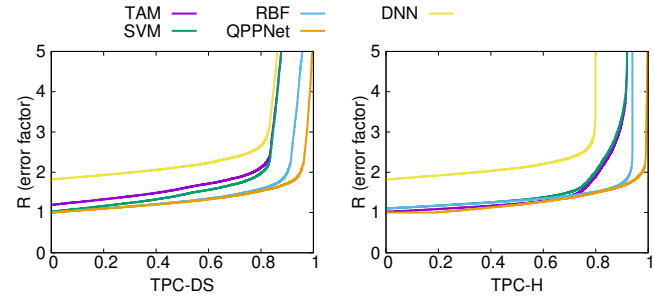
1. *SVM-based models (SVM)*: We implemented the learning-based approach proposed in [7], the state-of-the art in latency prediction for relational queries with no explicit human modeling. We use a regression variant of SVM (Support Vector Machine) that uses radial basis function kernel to built models for each operator. Selective applications of plan-level models are used in situations where the operator-level models are likely to be inaccurate. In contrast to our approach, the set of input vectors for both the operator and plan level models are hand-picked through extensive experimentation.
2. *Resource-based features (RBF)*: We also implemented a predictive model that take as input the features proposed by [33]. Although these features are picked for predicting resource utilization of query operators and by extension query plans, resource usage can be good indicator of query performance in non-concurrent query executions. Hence we modified the MART regression trees used in [33] to predict query latency. Similarly to the SVM approach, the input features of this model are hand-picked and not automatically engineered as in QPPNet. However, unlike the SVM approach, the RBF approach uses human-derived models for capturing operator interactions.
3. *Tuned analytic model (TAM)*: We also implemented a version of the tuned optimizer cost model model proposed in [73]. This approach uses the optimizer cost model estimate to predict query latency. First, some "calibration queries" are ran to determine the coefficients for the "calibrated cost model." These calibration queries are generally simple plans designed to measure the relationship between the optimizer's estimated cost and an operator's runtime. Then, this calibrated cost model is used to predict the query latency using the optimizer's cardinality estimates as inputs.[1] The TAM approach is thus entirely human-engineered, except for a sparse number of tuned parameters that are adjusted using the special calibration queries.
4. *Non-tree structured deep neural network (DNN)*: To validate our tree-structured architecture, we also compare against a fully-connected deep neural network with 8 hidden layers with 256 neurons each.[2] The inputs to this network is the concatenation of the sums of the vectors representing each operator type.

Evaluation metrics: To evaluate the prediction accuracy of these techniques, we use two metrics: *relative prediction error* and *mean absolute prediction error*. The relative prediction error has been used in [7, 33] and can be defined as follows. Letting Q be the set

[1]Our version of [73] uses optimizer estimates of cardinalities as inputs without the proposed "data sampling" optimization.
[2]Selected via grid search of 2^n for $2 <= n <= 10$.



(a) Comparison of relative and mean absolute error for different workloads



(b) Comparison of cumulative error factor $R(q)$ for different workloads. The x-axis signifies the proportion of the test set for which each approach achieved an error factor $R(q)$ below the value on the y-axis

Figure 6: Comparison of prediction accuracy

of test queries, letting $predicted(q \in Q)$ be the predicted latency of q, and letting $actual(q \in Q)$ be the actual latency, the relative prediction error is: $\frac{1}{|Q|}\sum_{q\in Q}\frac{|actual(q)-predicted(q)|}{actual(q)}$.

However, the "relative error" metric has several known flaws [66]. Specifically, relative error asymmetrically favors underestimates. No matter how bad an under-prediction is, the worst value the relative error can take on is 0. However, for over-predictions, the relative error is unbounded, hence the asymmetry. Thus, we also report the mean absolute error, a standard metric [56], which symmetrically penalizes under and over estimations:

$$\frac{1}{|Q|}\sum_{q\in Q}|actual(q) - predicted(q)|$$

A useful property of mean absolute error is that it shares the same units as the regression target; since we are predicting a quantity of time, the units of the mean absolute error are also time units.

We also report $R(q)$, the maximum of the ratio between the actual and the predicted and the ratio between the predicted and the actual (not to be confused with the coefficient of determination):

$$R(q) = \max\left(\frac{actual(q)}{predicted(q)}, \frac{predicted(q)}{actual(q)}\right)$$

Intuitively, the $R(q)$ value represents the "factor" by which a particular estimate was off. For example, if a model estimates a query's q latency to be 2 minutes, but the latency of the query is actually 1 minute, the $R(q)$ value would be 2, as the model was off by a factor of two. Similarly, if the model estimates a query's latency to be 2 minutes, but the latency of the query is actually 4 minutes, the $R(q)$ value would also be 2, as the model was again off by a factor of two.

6.1 Prediction Accuracy

The accuracy of each method at estimating the latency of queries in the TPC-H and TPC-DS workloads are shown in Figure 6a. The results reveal that our neural network approach outperforms the other baselines. The relative error improved by 9% (TPC-DS) and 5% (TPC-H) over RBF, by 25% (TPC-DS) and 24% (TPC-H) over SVM and by 28% (TPC-DS) and 21% (TPC-H) over TAM. In terms of absolute error, the average error decreased by 11 minutes (TPC-DS) and 7 minutes (TPC-H) from RBF, and by 18 minutes (TPC-DS) and 15 minutes (TPC-H) from SVM and 21 minutes (TPC-DS) and 13 minutes (TPC-H) from TAM.

We suggest two possible explanations for the larger improvement seen in TPC-DS as compared to TPC-H. First, the fact that the average TPC-DS query plan has more operators than the average TPC-H query plan (28 operators vs. 18 operators), resulting in QPPNet being able to take advantage of a larger amount of training data. Second, TPC-DS was designed to have more complex queries than TPC-H [50], and QPPNet's tree-based architecture captures these complexities better than hand-derived models, resulting in a large improvement over previous techniques. Overall, QPPNet is able to learn predictive features that are able to match and exceed the predictive power of the techniques we tested that rely on hand-engineered features, with more significant gains when the query workload is more complex (in our case, TPC-DS vs. TPC-H).

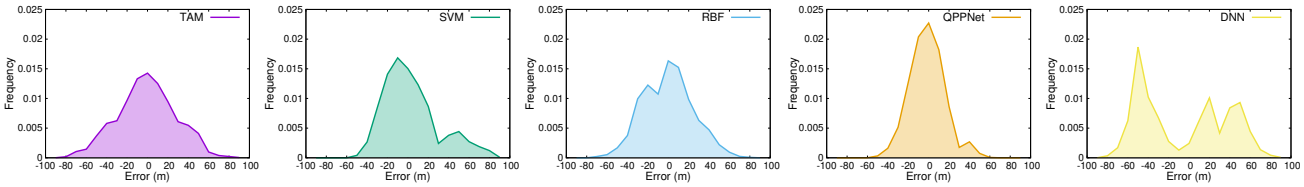
The notably poor performance of the non-tree structured neural network (DNN) provides evidence that the tree structure of QPPNet provides a significant advantage. Intuitively, the gap between DNN and QPPNet can be explained by *inductive bias* [44]: while deep neural networks can approximate any function [21], it is important to constrain the learned function to avoid overfitting (i.e., finding a function that performs well on the training data but generalizes poorly). By structuring QPPNet in a way that mirrors execution plans, we add model constraints and force QPPNet to fit the training data with a function that is intuitively related to the structure of the execution plans themselves (e.g., the latency of a parent operator is a function of its children). More broadly, this experiment provides evidence that out-of-the box application of deep learning models to the performace prediction problem is unlikely to yield good results. Instead, carefully encoding knowledge about query plans and their structure into deep learning architectures is critical to achieving acceptable prediction accuracy [10].

6.1.1 Prediction distribution

We analyzed how frequently each model's prediction are within a certain relative factor of the correct latency. We plot the distribution of $R(q)$ values in Figure 6b, in the style of cumulative density function. Each plot shows the largest $R(q)$ value achieved for a given percentage of the test set. For example, on the left hand graph for the QPPNet line, at 0.93 on the x-axis, the y-axis value is "1.5". This signifies that QPPNet's prediction was within at least a factor of 1.5 of the correct prediction for 93% of the testing data. For both datasets, QPPNet's curve has a smaller slope, and does not spike until it is much closer to 1 than the other curves. This means that QPPNet's estimates are within a lower error factor for a larger portion of the testing queries compared with the other techniques.

For both workloads, QPPNet has the highest proportion of the test set with an error factor less than 1.5. A high percentage of its predictions (89% for TPC-DS and 93% for TCP-H) are only within a factor of 1.5 of the actual latency, outperforming TAM by 38% (TPC-DS) and 15% (TPC-H), SVM by 21% (both TPC-H and TPC-DS) and RBF by 4% (TPC-DS) and 5% (TPC-H). The results indicate that our approach offers predictions closer to the real latency for a significantly higher number of queries.

Figure 7 shows error distributions for each technique on the TPC-DS dataset. Each plot has been normalized to sum to one. Figure 7 shows that all of the tested techniques gave unbiased estimations: no technique consistently over or under estimated query execution



(a) TAM (b) SVM (c) RBF (d) QPPNet (e) DNN

Figure 7: Prediction error distributions for TPC-DS

times (each has a mean around zero). Figure 7d shows that QPPNet achieves a tighter (lower variance) error distribution than the other techniques. Interestingly, Figure 7e shows that the non-tree structured neural network has a multi-modal error distribution. We hypothesize that the DNN technique is learning a model that essentially classifies TPC-DS queries into "long running" or "short running", and then assigns the mean latency of either group. This is a common occurrence in neural networks using L2 loss functions that are not well-structured to their problem domain [23].

.

6.1.2 *Prediction analysis per query type*

We evaluated how different types of queries affect the performance of QPPNet. We group queries from the TPC-H dataset into five different groups based on the "Choke Point" analysis presented by Boncz et al. in [11]: (1) *Agg*, queries that depend strongly on aggregation performance, (2) *Join*, queries that have voluminous joins, (3) *Locality*, queries with correlated filters requiring non-full-scans over data tables, (4) *Expr*, queries that evaluate complex logical or mathematical expressions, and (5) *Corr*, queries with correlated or dependent subqueries. Some query templates appear in multiple groupings, and some queries are not present in any group.[3]

Figure 8a shows the mean absolute error of the TPC-H queries grouped by their "Choke Point" categorization. We first observe that QPPNet's mean absolute error is relatively consistent across each category (16-19m) compared to other approaches (e.g., SVM ranges from 19m to 33m). Second, we observe that QPPNet's biggest gains, relative to the other techniques, are in the Join and Corr categories. Both of these categories exhibit more complex plan structures and operator interactions than the other categories: for example, the Join category contains join operators that write intermediate data into disk, drastically altering the latency of these joins and the joins further up in the query plan (which may read their inputs from disk). The Corr category contains join operators with bushy sub-trees (as opposed to a left-deep trees), which increases the complexity of the interactions among operators. We believe that the relatively improved behavior of QPPNet on these categories is indicative of QPPNet's ability to effectively capture interactions between operators and their impact on query latency.

The number of operator instances and the depth of a query plan directly affect the network structured used by QPPNet. The Corr category contained both the highest average number of operator instances (39 operators) and the highest average plan depth (average depth of 11), whereas the Expr category contained the lowest (on average, 20 operators and a depth of 5). While QPPNet achieved a lower predictive error on queries in Expr category, QPPNet had a lower predictive error on the largest and deepest query in the Expr category (29 operators, depth 10) than on the smallest query in the Corr category (23 operators, depth 7). It is difficult to attribute this behavior to either the type of the query or the size/depth of the plan alone, and we leave such investigations to future work.

[3]These groups correspond to the "strong" categorization in Table 1 of [11].

6.1.3 *Impact of warm cache*

In line with prior work [7, 33, 73], previous experiments have tested QPPNet's ability to predict query performance in a "cold cache" scenario. However, since caching is almost always used in practice and can affect query performance, we studied the impact of caching on the performance of each technique on the TPC-H workload. Specifically, we train each model using query execution data from both a cold cache and a warm cache execution of the training set. We note that queries are executed in a random order.

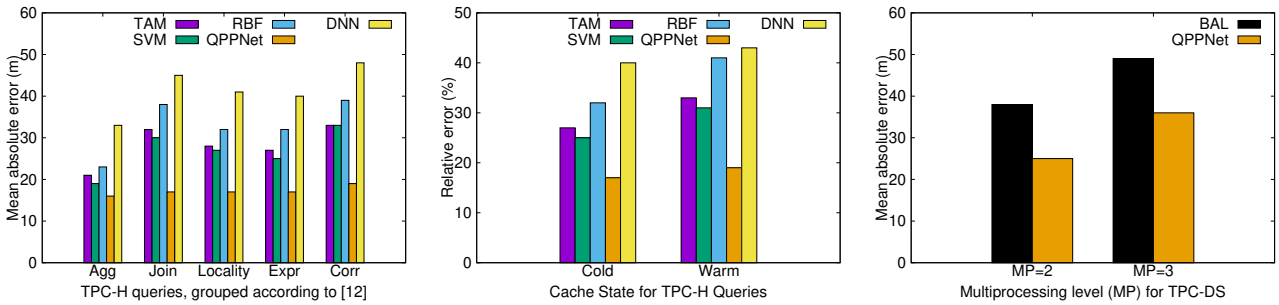
Figure 8b compares each technique's performance in a cold cache and warm cache scenario. For each technique, the relative error[4] is slightly higher when the cache is warm compared to when the cache is cold (e.g., QPPNet relative error is 17% with a cold cache and 19% with a warm cache). One potential explanation for this consistent decrease in performance is that query execution with a warm cache has slightly higher variance that execution with a cold cache (about 3% higher in our data). We hypothesize that QPPNet's accuracy in warm cache scenarios could be improved by adding information about the state of the cache to the inputs of the neural units, and we leave such investigations to future work.

6.1.4 *Concurrent query executions*

Since queries can exhibit complex interactions and performance characteristics when ran concurrently [15, 72], we also evaluated QPPNet's ability to predict the performance of queries at various multiprocessing levels. To model concurrent queries at multiprocessing level MP (i.e., running MP queries concurrently), we create a special neural unit which takes as input the MP outputs from the root nodes of MP concurrently executing queries. We evaluate this approach on the TPC-DS dataset at multiprocessing levels $MP = 2$ and $MP = 3$. In each case, we executed every possible grouping of queries (2346 pairs at $MP = 2$ and 52394 triples at $MP = 3$), and then randomly split them into a training set (90% of the data) and a test set (10% of the data).

We compare our approach with [14], a state-of-the-art approach based on modeling and predicting buffer access latency (BAL). While [14] focuses on concurrent queries that arrive in a queue, we focus on a simpler problem in which concurrently executing queries arrive at the same time. Additionally, [14] makes continuous predictions about a query's latency as the query executes (with increasing accuracy), whereas here we focus only on predicting the latency strictly prior to execution. Lifting these restrictions is not trivial, and we leave it to future work. We train the BAL model using the procedure outlined in [14]. Figure 8c shows the results. For $MP = 2$, QPPNet achieves a mean absolute error of 24 minutes (only a 20% increase from the non-concurrent case), whereas the BAL approach achieves a mean absolute error of 38 minutes (QPPNet's performance is nearly 36% better in this case). Similar results are observed at $MP = 3$.

[4]Comparing the absolute error between warm and cold cache scenarios would be meaningless, as the query times differ drastically between the two: the average query latency decreased by 34% in the warm cache execution.



(a) Prediction error per query type (b) Prediction error for warm vs cold cache (c) Prediction error per concurrency levels

Figure 8: Analysis of prediction accuracy

6.1.5 *Database variations*

Database size: Until now, all experiments were performed using databases containing about 100GB of data. However, different database sizes will have different performance characteristics: a small database may fit entirely in memory, whereas a large dataset may span several disk platters. Of course, databases change in size during normal operation as well, growing or shrinking to match the user's needs. In this section, we measure (1) QPPNet's ability to predict query latency on databases of different sizes, and (2) QPPNet's ability to predict latency on a significantly larger or smaller database than the database used to train QPPNet.

We constructed 6 TPC-H databases, ranging in size from 50GB to 1600GB. We evaluated QPPNet's accuracy when trained and tested on each pair of databases. For example, we trained QPPNet on a 50GB database and then tested performance on a 800GB database. The resulting average R values are plotted in Figure 9a. The lower-left to upper-right diagonal shows that when QPPNet is trained and evaluated on databases of equal size (e.g., trained on a 800GB database and evaluated on the same), the average R value is relatively constant, around 1.17. Due to space constraints, we omit plots for the other techniques, although we note that each technique also had consistent average R values when trained and evaluated on databases of the same size (TAM 1.28, SVM 1.32, RBF 1.26).

Figure 9a shows how QPPNet behaves when trained on a database of an initial size which is then grown or shrunk. First, we observe that QPPNet can generalize "down" (to smaller databases) much better than it can generalize "up" (to larger databases). For example, a model trained on a 50GB database does a poor job predicting query latency over a 1600GB database. But, a model trained on a 1600GB *can* achieve reasonable performance on a 50GB database. Second, we observe that models trained on larger databases generalize to even larger databases better than small databases generalize to slightly larger databases. For example, a model trained on a 50GB database gives poor performance on a 100GB database, but a model trained on a 800GB database has relatively better performance on a 1600GB data. A potential explanation for this behavior is that larger databases cover a wider gamut of DBMS phenomena (e.g., spill joins, cache misses) than smaller databases.

Figure 9b compares how each technique generalizes to other database sizes when initially trained on a 200GB database, and then tested on a database of larger or smaller size. Compared with other approaches, QPPNet achieves the best error factor $R(q)$ value score for each database size. When the size of the database was increased beyond 200GB, the gap between the QPPNet and the TAM approach gets significantly smaller. A potential explanation for this behavior is that the TAM approach uses a model that captures the asymptotic behavior of each query operator, whereas other approaches lack this *a priori* knowledge.

While these experiments demonstrate that QPPNet is somewhat capable of adapting to different database sizes, real world applications would likely need to retrain QPPNet periodically as the underlying database grows, shrinks, or shifts its distribution. We leave efficiently handling these challenges to future work.

Database skew: We evaluate the performance of QPPNet on skewed databases. We used the Zipfian TPC-H generator [4] to generate 100GB TPC-H databases as skew levels 1 (low skew), 2 (moderate skew), and 4 (high skew). We then evaluated each technique as before. The results are plotted in Figure 9c.

All the tested approaches except for DNN had consistent performance even at high skew levels. QPPNet achieves the best prediction accuracy for all skewness levels. While these results indicate that most query performance prediction techniques, including QPPNet, are robust to skew, they do not indicate how well a model trained at one skewness level would perform on data at another skewness level. QPPNet, along with most statistical learning techniques, assume high correspondence between the training data and the test data. Similarly, QPPNet assumes that the training database tuning corresponds to the test database tuning (e.g., same shared buffer, underlying hardware). We leave evaluating cases where these correspondences do not hold to future work.

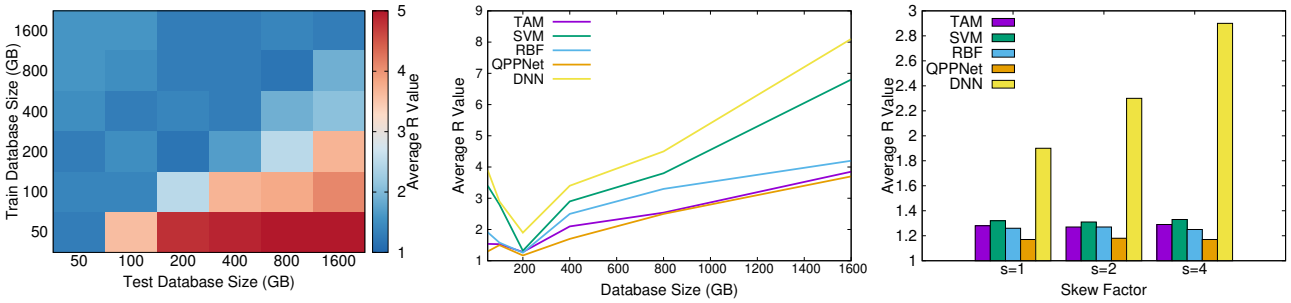
6.2 Training Overhead

Next, we evaluate model training, including analyzing the effectiveness of various optimizations.

Optimizations: We evaluated the training optimizations discussed in Section 5: *information sharing*, caching computations that are shared, and *bucketing*, grouping trees with similar structures into batches in order to take advantage of vector processing.

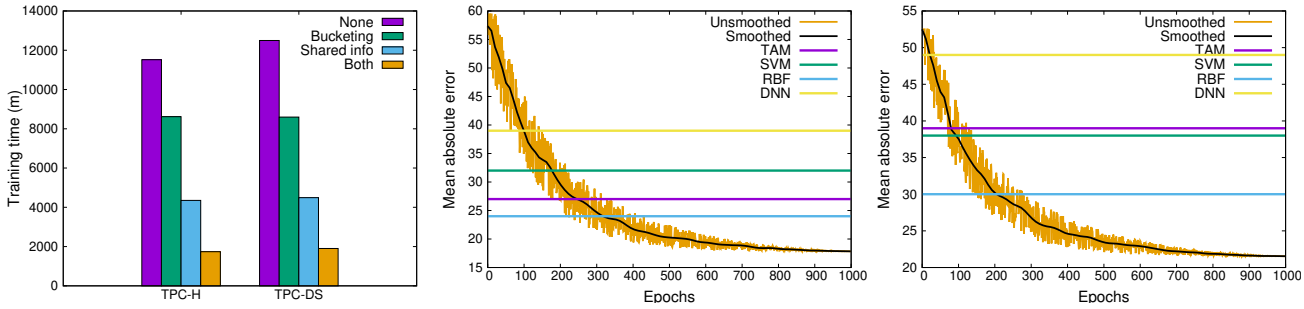
We evaluated these optimizations by training until the network converged to the best-observed accuracy using no optimizations (*None*), the bucketing optimization alone (*Bucketing*), the information sharing optimization alone (*Shared Info*), and both optimizations (*Both*). For the bucketing optimization, there were 131 (TPC-DS) and 45 (TPC-H) equivalence classes, ranging in size from thousands of queries to only 6 queries. Figure 10a shows the results. Without optimizations, training takes well over a week. Of the new optimizations, information sharing is the more significant in these experiments, bringing the training time down from over a week to a little under 3 days. Both optimizations combined bring the training down to only slightly over 24 hours.

We also measured the memory usage of the information sharing approach, which requires additional space to cache results. The size of the cache was minuscule in comparison to the size of the neural network's weights, with the cache size never exceeding 20MB. We conclude that both the information sharing and batch sampling optimizations reduce the time needed to train the neural network.



(a) R, varying train and test database sizes (b) R for various database sizes (trained at 200GB) (c) R for different data skew levels

Figure 9: Database size and skew (TPC-H)



(a) Impact of training optimizations (b) Training convergence (TPC-H) (c) Training convergence (TPC-DS)

Figure 10: Training overhead

Training convergence: After each training epoch (a full pass over the training queries), we recorded the mean absolute error across the test set. The results are shown in Figure 10b and 10c. While the neural network model did not converge until epoch 1000 ($\approx$ 28 hours), the performance of the neural network begins to exceed the performance of SVM at around epoch 250 (7 hours) with TPC-H, or after 150 epochs (4.5 hours) for TPC-DS. The neural network begins to exceed the performance of RBF after around 350 epochs (10 hours) for TPC-H, or after 250 epochs for TPC-DS (7 hours).

For some applications, these training overheads may be acceptable. The other tested approaches had significantly lower training time: TAM (20 seconds), SVM (11 minutes), RBF (1 minute). We advise users to be conscious of training time / accuracy tradeoffs. While new techniques [17] promise to drastically decrease neural network training time, we leave investigating them to future work.

6.3 Inference Time

We next evaluate inference time. Figure 11a plots inference time against mean absolute error for TPC-H and TPC-DS. TAM has exceptionally low inference time, as TAM only requires evaluating a polynomial model. SVM and RBF have higher inference time, as they compute dot products with support vectors (SVM) or descend a MART tree (RBF). While QPPNet has the highest inference time, it has significantly lower error and can still perform inference in under 100ms (TPC-H/DS queries can run for several minutes).

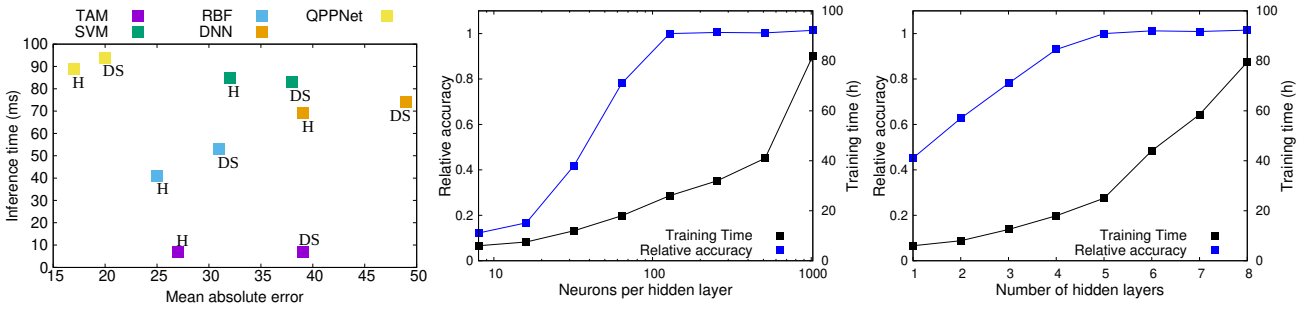
We also compare QPPNet's inference time for the TPC-DS dataset, where the average query has 28 operators, and the TPC-H dataset, where the average query has around 18 operators. Despite the increased number of operators (and thus an increased number of neural units), QPPNet's inference time is only 8ms higher for TPC-DS than for TPC-H. Based on profiling, we determined that QPPNet's inference procedure spent nearly 80% of its time in PyTorch [54] initialization routines, as opposed to performing neural network computations. We leave further optimizing QPPNet's inference time to avoid repetitive re-initialization to future work.

6.4 Network Architecture

So far, we have used 5 hidden layers with 128 neurons for each neural unit. Such networks may be considered small by modern standards. However, when assembled together into a tree, the network is much larger (one to two orders of magnitude). While there is no theoretically-best number of hidden layers or number of neurons per layer, good values can be found automatically with no manual intervention: the number of neurons and layers can be increased until accuracy no longer improves significantly. Intuitively, "deeper" (more hidden layers) architectures enable more feature engineering, as additional layers add additional transformations of the inputs. On the other hand, "taller" (larger hidden layers) architectures allow each feature transformation to be richer, as they have more weights and thus carry more data [30, 61].

We analyze four of the variables at play when trying to find the correct network configuration: the number of hidden layers, the number of neurons, the maximum accuracy the network can reach, and the time it takes to train the network. Generally, increasing either the number of hidden layers or the number or neurons results in an increase in training and inference time due to an increase in the number of weights. But, if the number of neurons or hidden layers is set too low, the network might not have enough weights to learn the underlying data distribution well enough. We thus seek the number of neurons and hidden layers that will minimize training time while still giving near-peak accuracy.

Number of neurons: Figure 11b shows the training time and relative accuracy when varying the number of neurons inside each of the five hidden layers. With an extremely small number of neurons (8 neurons), training time is low (6 hours), but accuracy is extremely poor: QPPNet achieves less than 15% of the accuracy that the 128-neuron network does. On the other hand, using an extremely large number of neurons causes the training time to skyrocket: with 1024 neurons per hidden layer, training time is nearly four times what is required for the 128 neuron network, with only a tiny increase in accuracy (less than 1%).



(a) Mean absolute error vs. inference time (b) Neurons vs. accuracy and training time (c) Layers vs. accuracy and training time

Figure 11: Performance properties of QPPNet

One may notice that the training time seems to grow with the log of the number of neurons at first, but then eventually becomes linear. This is because neural networks are trained on GPUs equipped with highly-parallel vector processing units. There is thus sublinear increases in training time until there is approximately one weight per vector processing core, after which the training time changes as expected. When the number of neurons greatly exceeds the capacity of the GPU, the slowdown will become worse than linear.

Number of hidden layers: Figure 11c shows a similar experiment, varying the number of hidden layers and keeping the number of neurons fixed at 128. Note that connecting two layers with 128 neurons to each other requires a matrix of size 128×128, so each additional hidden layer adds on the order of 2^{14} additional weights.

Adding hidden layers has a similar behavior to increasing the number of neurons: initially, each addition brings about a small increase in training time but a large jump in accuracy. Eventually, adding another hidden layer produces a much larger jump in training time and a much smaller jump in accuracy. Figure 11c shows that adding more than 5 hidden layers, at least when the size of each hidden layer is 128 neurons, does not bring about much benefit.

7. RELATED WORK

Query performance prediction: A number of approaches leveraging machine learning and statistical analysis to address the problem of query performance prediction have been proposed. We discuss and compare with [7, 33, 73] in our experimental study. [18] focuses on predicting multiple query resource usage metrics simultaneously (but not execution times). Both [53, 75] predict statistics about queries in XML databases. [72] demonstrating that optimizer cost models can be used to predict query performance if one is willing to sample a percentage of the underlying data.

All these techniques suffer from similar drawbacks: first, they require human experts to analyze the properties of an operator or query execution plan and determine how they should be transformed into features for a machine learning algorithm, whereas our deep-learning approach requires no such feature engineering. Second, while some of these approaches model plans, operators, or a combination thereof, none of them learn the *interactions* between various combinations of operators, as the approach presented here does.

A number of techniques [14, 15, 70, 72] extend to concurrent query performance prediction for analytical queries. These techniques assume a-priori knowledge of query templates [14], query structure [70] and/or require extensive offline training on representative queries [15, 72]. Furthermore, their proposed input features, metrics and models are hand-tuned to handle only analytical tasks, which make them less applicable to diverse workloads.

Cardinality estimation: Cardinality estimation is fundamentally related to performance prediction, as an operator's cardinality often correlates with its latency. Cardinality estimation techniques include robust statistical techniques [9,28,46], adaptive histograms [6, 63], and deep learning [25, 36]. While cardinalities are certainly an indicator of latency, translating accurate cardinality estimates to a total plan latency is not a trivial task. However, cardinality estimation techniques could be easily integrated into QPPNet by inserting the estimates into its neural units input vector. The neural network could then learn the relationship between these estimates and the latency of the entire query execution plan.

Progress estimators: Work on query progress indicators [31, 34, 38, 47, 74] essentially amounts to frequently updating a prediction of a query's latency. These approaches estimate the latency of a query *as it is running*, and the estimate that these techniques make at the very start of the query's execution may be quite inaccurate, but are quickly refined and corrected during the early stages of a query's progress. This greatly limits their applicability for ahead-of-time query performance prediction, and thus we do not compare against any of these techniques directly.

Deep learning: We are not the first to apply deep learning to problems in data management. Deep learning [30] has driven a recent groundswell of activity in the systems community [71], including several works applying deep reinforcement learning [45] to query optimization [27, 41, 43, 52] and job scheduling [39], learned embeddings to entity matching [49] and data exploration [16], supervised predictive models to index selection [55, 60], indexes themselves [26], and cardinality estimation [25, 36].

8. ACKNOWLEDGMENTS

This research is funded by NSF IIS 1815701, NSF IIS Career Award 1253196, and an Amazon Research Award.

9. CONCLUSIONS AND FUTURE WORK

We have introduce QPPNet, a novel neural network architecture designed to address the challenges of query performance prediction. The architecture allows for plan-structured neural networks to be constructed by assembling operator-level neural units (neural networks that predict the latency of a given operator) to form a tree-based structure that matches the structure of the query plan generated by the optimizer. We motivated the need for this novel model, described its architecture and have shown how the model can be effectively trained. Experimental results demonstrate that QPPNet outperforms state-of-the-art solutions.

Future work could advance in a number of directions. For example, the neural network architecture presented here could be adapted to handle a dynamic number of concurrent queries. Doing so would require modeling the resource contention among queries. Additionally, more complex operator interactions, such as sideways information passing [22,62] or parallel operator execution, could be incorporated into the model.

10. REFERENCES

[1] MySQL database, https://www.mysql.com/.

[2] PostgreSQL database, http://www.postgresql.org/.

[3] SQLite database, https://www.sqlite.org.

[4] TPC-H skewed, https://www.microsoft.com/en-us/download/details.aspx?id=52430.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467 [cs]*, Mar. 2016.

[6] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 181–192, New York, NY, USA, 1999. ACM.

[7] M. Akdere and U. ̧Cetintemel. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 390–401. IEEE, 2012.

[8] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM.

[9] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 119–130, New York, NY, USA, 2005. ACM.

[10] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. June 2018.

[11] P. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*, TPC '14, pages 61–76, Berlin, Heidelberg, 2014. Springer-Verlag.

[12] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of COMPSTAT'2010*, COMPSTAT '10, pages 177–186. Physica-Verlag HD, 2010.

[13] Y. Chi, H. J. Moon, H. Hacigumus, and J. Tatemura. SLA-tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '11, pages 129–140, Uppsala, Sweden, 2011. ACM.

[14] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 337–348, Athens, Greece, 2011. ACM.

[15] J. Duggan, O. Papaemmanouil, U. Cetintemel, and E. Upfal. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '14, pages 109–120, 2014.

[16] R. C. Fernandez and S. Madden. Termite: A System for Tunneling Through Heterogeneous Data. In *AIDM @ SIGMOD 2019*, aiDM '19, 2019.

[17] J. Frankle and M. Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. *International Conference on Learning Representations*, 2019.

[18] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *2009 IEEE 25th International Conference on Data Engineering*, ICDE '09, pages 592–603, Mar. 2009.

[19] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *PMLR '11*, pages 315–323, Fort Lauderdale, FL, USA, Apr. 2011. PMLR.

[20] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997.

[21] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, Jan. 1989.

[22] Z. G. Ives and N. E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 774–783, Apr. 2008.

[23] K. Janocha and W. M. Czarnecki. On Loss Functions for Deep Neural Networks in Classification. *Theoretical Foundations of Machine Learning*, Feb. 2017.

[24] Jun Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled Query Execution Engine using JVM. In *22nd International Conference on Data Engineering (ICDE'06)*, ICDE '06, pages 23–23, Apr. 2006.

[25] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[26] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[27] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv:1808.03196 [cs]*, Aug. 2018.

[28] P.-A. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality Estimation Using Sample Views with Quality Assurance. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 175–186, New York, NY, USA, 2007. ACM.

[29] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks*, pages 255–258, 1998.

[30] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[31] K. Lee, A. C. König, V. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash, R. Nehme, J. Li, and J. Naughton. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1753–1764, New York, NY, USA, 2016. ACM.

[32] H. Leeb and B. M. Pötscher. Sparse estimators and the oracle property, or the return of Hodges' estimator. *Journal of Econometrics*, 142(1):201–211, Jan. 2008.

[33] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[34] J. Li, R. V. Nehme, and J. Naughton. GSLPI: A Cost-Based Query Progress Indicator. In *2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 678–689, Apr. 2012.

[35] Z. C. Lipton, J. Berkowitz, and C. Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv:1506.00019 [cs]*, May 2015.

[36] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[37] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, Apr. 2017.

[38] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a

Progress Indicator for Database Queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 791–802, New York, NY, USA, 2004. ACM.

[39] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]*, Oct. 2018.

[40] R. Marcus and O. Papaemmanouil. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *PVLDB*, 9(10):780–791, 2016.

[41] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '18, Houston, TX, 2018.

[42] R. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *arXiv:1902.00132 [cs]*, Jan. 2019.

[43] R. Marcus and O. Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[44] T. M. Mitchell. The Need for Biases in Learning Generalizations. Technical report, 1980.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[46] G. Moerkotte, T. Neumann, and G. Steidl. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB*, 2(1):982–993, 2009.

[47] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[48] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI '16, pages 1287–1293, Phoenix, Arizona, 2016. AAAI Press.

[49] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep Learning for Entity Matching: A Design Space Exploration. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 19–34, New York, NY, USA, 2018. ACM.

[50] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *VLDB*, VLDB '06, pages 1049–1058, Seoul, Korea, 2006. VLDB Endowment.

[51] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[52] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning*, DEEM '18, 2018.

[53] M. T. Ozsu, N. Zhang, A. Aboulnaga, and I. F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *22nd International Conference on Data Engineering*, ICDE '06, page 61, Apr. 2006.

[54] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *Neural Information Processing Workshops*, NIPS-W '17, 2017.

[55] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, 2017.

[56] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[57] M. Poess and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Records*, 29(4):64–71, Dec. 2000.

[58] J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–105, Nov. 1990.

[59] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, Sept. 2016.

[60] M. Schaarschmidt, A. Kuhnle, B. Ellis, K. Fricke, F. Gessert, and E. Yoneki. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *arXiv:1808.07903 [cs, stat]*, Aug. 2018.

[61] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan. 2015.

[62] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the Vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, ICDE '13, pages 1196–1207, Apr. 2013.

[63] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, VLDB '01, pages 19–28, 2001.

[64] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. DeWitt. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 388–400, New York, NY, USA, 2016. ACM.

[65] K. S. Tai, R. Socher, and C. D. Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *arXiv:1503.00075 [cs]*, Feb. 2015.

[66] C. Tofallis. A Better Measure of Relative Prediction Accuracy for Model Selection and Model Estimation. *Journal of the Operational Research Society*, 2015(66):1352–1362, July 2014.

[67] S. Tozer, T. Brecht, and A. Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference On*, ICDE '10, pages 397–408, Mar. 2010.

[68] A. van den Oord, O. Vinyals, and K. Kavukcuoglu. Neural Discrete Representation Learning. *Neural Information Processing*, Nov. 2017.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *Neural Information Processing*, June 2017.

[70] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 363–378, 2016.

[71] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Rec.*, 45(2):17–22, Sept. 2016.

[72] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *PVLDB*, 6(10):925–936, 2013.

[73] W. Wu, H. Hacigumus, Y. Chi, S. Zhu, J. Tatemura, and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1081–1092, Washington, DC, USA, 2013. IEEE Computer Society.

[74] X. Xie, Z. Fan, B. Choi, P. Yi, S. S. Bhowmick, and S. Zhou. PIGEON: Progress indicator for subgraph queries. In *2015 IEEE 31st International Conference on Data Engineering*, ICDE '15, pages 1492–1495, Apr. 2015.

[75] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *VLDB*, VLDB '05, pages 289–300, 2005.