Accelerating Deep Learning Inference in Relational Database Systems

by

Saif Masood

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2024 by the

Graduate Supervisory Committee:

Jia Zou, Chair
Xusheng Xiao
Yingzhen Yang

ARIZONA STATE UNIVERSITY
May 2024

ABSTRACT

Deep learning has become a potent method for drawing conclusions and forecasts from massive amounts of data. But when used in practical applications, conventional deep learning frameworks frequently run into problems, especially when data is stored in relational database systems. Thus, in recent years, a stream of research in integrating machine learning model inferences with a relational database to achieve benefits such as avoiding privacy issues and data transfer overheads is observed. The logic for performing the inference using the DNN model can be encapsulated in a user-defined function (UDF). These UDFs can then be integrated with the query interface of the DBMS and executed by the query execution engine. While it is relatively straightforward to leverage the User Defined Functions (UDFs) to implement machine learning algorithms using parallelism, it is observed that such implementations will not always be optimal and may incur issues in balancing the database threading and the threading of the libraries that the UDFs invoke. Since relational databases provide native support for relational operators, it is possible to leverage a cost model to make decisions for selectively transforming the UDFs based inference logic into a model-parallel implementation for optimal performance. Thus, this thesis will focus on the following: 1. Designing a domain-specific language for implementing the UDFs using Velox library, which can be lowered to a graph-based intermediate representation (IR); 2. Providing a cost model that aids in the decision-making of converting a UDF-centric implementation to a relation centric one.

ACKNOWLEDGMENTS

I extend my sincere gratitude to Dr. Jia Zou, Assistant Professor in the School of Computing and Augmented Intelligence, for her invaluable guidance and unwavering support throughout my thesis journey. Under Prof. Zou's mentorship, I delved deep into the realm of Machine Learning Systems during our collaborative work on the Unified Compilation project, which forms the essence of this thesis. Before joining Prof. Zou's lab, I had not authored any research papers, but her mentorship enabled me to contribute to the academic community with two publications. Additionally, I am indebted to Prof. Zou for broadening my academic horizons by allowing me to partake in SIGMOD'23, a significant research conference, where I gained invaluable insights and experiences. Furthermore, I extend my gratitude to Dr. Yingzhen Yang and Dr. Xusheng Xiao for generously dedicating their time and expertise as committee members for my Master's thesis defense.

I express my heartfelt appreciation to the members of the Cactus Data-Intensive Systems Lab at ASU, especially Lixi, Hong, and Qi for their assistance whenever I required it. Their support and collaboration greatly enriched my research journey.

Furthermore, I extend my gratitude to the Velox team at Meta and our esteemed collaborators at Amazon for their invaluable contributions and collaborative efforts, which significantly enhanced the scope and quality of my work.

I am also deeply thankful to my parents for their enduring support and encouragement throughout my Master's degree.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Deep learning in relational databases seeks to improve data processing, analysis, and decision-making capabilities by fusing deep learning methods with conventional relational database management systems (RDBMS). Relational databases have historically been used mostly for storing structured data and running SQL queries to manipulate and retrieve data. But with the development of deep learning, there's been an increasing interest in using neural networks in collaboration with relational databases (Zhou et al., 2024; Zhou et al., 2022; Zou et al., 2021; Jankov et al., 2020).

There emerges a class of AI/ML applications running on top of relational data including use cases from IBM, Amazon, and Meta, such as fraud card detection on credit-card transaction records, recommendation over customer orders and profiles, and conversational AI over customer activity and profiles. Decoupling the data preprocessing and AI/ML leads to performance gaps, development and management complexity, and privacy issues. Thus, there are practical uses for deep learning inference acceleration in RDBMS. Recommendation engines and fraud detection (Cheng et al., 2020) are two instances that will exemplify the speed up. The worldwide banking, card, and payment industries lost an estimated 385 billion dollars in 2021 as a result of fraud detection (Katkov, 2022). Deep learning model inferences are too costly and slow to be implemented in online processing systems. The real-time inference required for

data-intensive routines, such as credit card transactions, can add an additional 50 to 80 milliseconds to transaction processing time. This delay slows down approval times for transactions, leading to customer friction. As a result, the AI system processes very few (less than 10 per cent) high-risk transactions. The majority of transactions undergo post-transaction analysis after being scored using rules.

Thirty-five percent of Amazon's 2018 sales came from personalized recommendation models ("Amazon's Recommendation Algorithm Drives 35% of Its Sales," 2020). For any recommendation task, the latency requirements can range from tens to hundreds of milliseconds. There are thousands of features, and up to 10,000 inquiries could be in a batch. The primary bottleneck is the data transfer from the RDBMS to the ML runtime, which necessitates pre-loading, pre-partitioning, and pre-indexing of the data in memory before model serving.

Both the scenarios mentioned above could benefit from a speed-up in inference. Accelerating fraud detection using a hybrid approach could lead to an increase in the number of transactions that could be run through the system, thereby flagging more potentially fraudulent transactions and minimizing losses. On the other hand, the in-database inference could save the data transfer and pre-processing costs (Guan et al., 2023) in case of recommendation models, leading to a richer experience for the end user.

This thesis delves into the significance of expediting Deep Neural Network (DNN) inference within relational systems. It begins by exploring common integration patterns between Deep Learning (DL) and Relational Database Management Systems (RDBMS). Each integration pattern presents its own set of tradeoffs, suited to specific use cases. The

thesis emphasizes the adoption of User-Defined Functions (UDFs) as a central approach to accelerate DNN inference.

Meta's Velox execution engine emerges as an optimal choice for implementing UDFs or Machine Learning (ML) kernels for DNN inference. Leveraging its support for relational operators, runtime optimizations, and user-friendly components, Velox streamlines the DNN inference process effectively.

The thesis outlines the design and implementation specifics of the ML kernels. Various kernels essential for DNN forward propagation, such as matrix multiplication, matrix addition, and ReLU activation, are meticulously developed. External libraries like Torch and Eigen are harnessed to implement these UDFs.

To assess the performance of Velox kernels, two benchmarks are conducted. The first evaluates MNIST handwritten digits image classification, while the second scrutinizes a convolution benchmark known as Deep Bench Conv1. In most scenarios, Velox kernels surpass popular DL frameworks like TensorFlow and PyTorch in terms of performance.

Moreover, the thesis explores the impact of different hyperparameters on benchmark performance. It identifies a challenge related to the calibration of Velox and external library threads, which share a common runtime.

Furthermore, the thesis delves into optimizing the inference query plan by employing a cost model. It provides a high-level overview of the cost model's design and proposes a strategy to estimate the plan cost using learned coefficients. This approach aims to enhance the efficiency of DNN inference within relational systems.

The remaining sections of the thesis are structured as follows:

Chapter 2: This chapter delves into the existing literature and related work pertinent to the subject matter. It provides a comprehensive review of research, projects, and advancements in the field, offering valuable context for the thesis's contributions.

Chapter 3: Here, readers are presented with foundational knowledge and background information essential for understanding the concepts and systems central to this thesis. This chapter lays the groundwork by elucidating key terminology, methodologies, and technologies utilized throughout the research.

Chapter 4: Focusing on the heart of the thesis, this chapter delves into the intricate design and implementation intricacies of Machine Learning (ML) kernels within the Velox execution engine. It explores the development process, challenges encountered, and innovative solutions devised to overcome them, providing readers with a deep understanding of the technical aspects of the project.

Chapter 5: This chapter delves into the creation of a simple cost model tailored to optimize decision-making processes within the inference query plan. By dissecting the design rationale and methodology behind the cost model, readers gain insight into the strategic approach employed to enhance performance in real-world scenarios.

Chapter 6: Concluding the thesis journey, this chapter reflects on the challenges encountered throughout the research process. It also offers insights and suggestions for future research directions, highlighting areas for further exploration and potential advancements in the field. This reflective chapter serves as a springboard for ongoing discourse and innovation within the realm of DNN inference optimization in relational systems.

# CHAPTER 2

# LITERATURE SURVEY

In RDBMS, there are various methods for providing model serving. They fall into two general categories: white box and black box (Zhou et al., 2024). Machine learning models are implemented using the "black box" technique, which conceals the core mechanisms and architecture of the models. Rather, the model is viewed as a "black box," one that receives input data and produces predictions without allowing insight into the workings of the model itself. Black-box deployment techniques with modifiable parameters for optimization are used by well-known model serving systems including Microsoft's Azure ML (Barga et al., 2015), Amazon SageMaker (Liberty et al., 2020), and TensorFlow Serving (Olston et al., 2017). For example, these systems balance accuracy and latency by dynamically optimizing batch sizes.

The internal workings, parameters, or intermediate representations of the machine learning models being used, on the other hand, are exposed and leveraged in a white box model serving method. More control and transparency over the deployed models are provided by this method, which enables more precise customization and optimization. A bin-packing approach is introduced by Nexus (Shen et al., 2019) to optimize batch sizes for individual layers and distribute machine learning layers to devices. For improved performance, DeepSpeed (Yazdani Aminabadi et al., 2022) and FlexGen (Sheng et al., 2023) include offloading particular weights to the CPU or even the disc. Other strategies

used are operator sharing amongst models, early prediction based on cached inference findings, and utilizing approximation inference approaches for increased performance.

A few strategies have been used to speed up the RDBMS's DL query serving process. A recently popular method is to use a vector database for caching inference results. This method takes advantage of popular datasets like Pinecone (2023), Milvus (Wang et al., 2021), Faiss (Johnson et al., 2019), etc. They employ a variety of indexing strategies, including product quantization, hierarchical navigable small world (HNSW), and locality-sensitive hashing (LSH), to make search easier.

A challenge in executing deep learning inference within RDBMS is the efficient execution of complex neural network models. Several studies have explored query optimization techniques to improve the performance of DL inference. For example, Chen et al. (2023) proposed a query optimization framework for accelerating DL inference in PostgreSQL databases by optimizing the execution of neural network operations.

Some RDBMS systems offer native support for executing machine learning models directly within the database environment. Oracle Database provides in-database execution capabilities for TensorFlow (Abadi et al., 2015) models through its Oracle Machine Learning framework (Oracle, 2022). Similarly, IBM Db2 offers in-database inference capabilities through its Db2 Machine Learning Accelerator, enabling efficient execution of deep learning models within the database (IBM, 2019). In this context, model pipeline and partitioning strategies have also been introduced, particularly for huge models that are too large to fit in memory ("Systems for Parallel and Distributed Large-Model Deep Learning Training," 2023).

# CHAPTER 3

# BACKGROUND

## 3.1 Deep Neural Networks (DNN)

One kind of machine learning model that draws inspiration from the composition and operations of the human brain is the neural network. They are made up of layers of networked nodes, known as neurons. Every neuron takes in information, processes it via an activation function, and then generates an output.

A particular kind of neural network called Deep Neural Networks (DNNs) is distinguished by the presence of numerous hidden layers in between the input and output layers. The word "deep" describes these networks' depth, or the quantity of layers they have. DNN layers are logically divided into three types:

1. Input Layer: This is the layer to which the input is passed. Each neuron processes or represents an input feature.
2. Hidden Layer: Most computing takes place in these layers, which are situated between the input and output layers. Multiple neurons make up each buried layer, which transforms the incoming data. The training process teaches these transitions, which deepen in complexity as they go.
3. Output Layer: The neural network's ultimate output is generated by this layer. The type of problem being solved determines how many neurons are in the output layer.

For instance, two output neurons might indicate the odds of belonging to each class in a classification job with two classes.

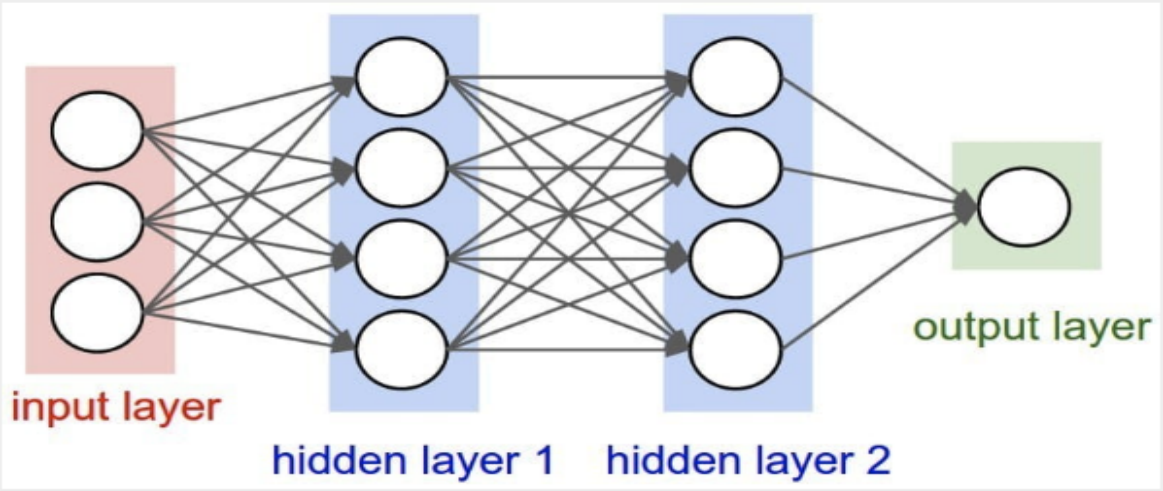


Figure 1: Johnson, J. (2020, July 27). Deep Neural Network. BMC. https://www.bmc.com/blogs/deep-neural-network/

DNNs are popular when it comes to image classification. The input image is flattened and fed to the neural network. When a DNN is trained, its weights and biases are updated to minimize a loss function that calculates the difference between the expected and actual outputs. Usually, an optimization approach like gradient descent is used for this. The training process involves forward propagation, calculation of loss, and backward propagation along with updation of weights. This process is repeated for multiple iterations till the loss converges.

Once a DNN has been trained, it can be used to perform inference. During inference, for each layer, a linear transformation is performed followed by an activation function like ReLU:

$$Z^{[l]} = W^{[l]} . A^{[l-1]} + b^{[l]}$$

In the formula, l is the layer number, W is the weight matrix, A is the activation function and b is the bias vector. Thus, inference at each layer can be written as a matrix multiplication, followed by a matrix addition, and finally an activation function.

## 3.2 DNN-RDBMS Integration Models

The integration of Deep Learning models with relational databases falls into three categories(Zhou et al., 2024) :

### 3.2.1 ML-Centric

Designing solutions that smoothly blend the advantages of deep learning and conventional relational database management systems (RDBMS) is essential to integrating deep neural networks (DNNs) into relational databases in a machine learning (ML) centered manner. Therefore, ML runtimes handle the ML operations in this approach. Relational databases are well-suited for storing structured data, and deep learning models perform well in tensor operations unique to machine learning. One disadvantage of ML-centric integration is that it necessitates procedures and tools for the relational database's training data extraction. To obtain pertinent data for model training,

this entails querying the database, carrying out any required preprocessing (such as feature scaling and normalization), and getting readily labeled datasets if supervised learning is going to be used. This leads to memory copy and data transfer overheads. Furthermore, co-optimization strategies for ML and SQL processes are difficult because they are separate systems. It should be possible to run SQL queries including both model inference and conventional relational operations within the integrated system. For the deployed deep learning models to seamlessly integrate with it, the query execution engine must be extended. For quicker inference, the system might, for instance, optimize queries to shift processing to the GPU.

### 3.2.2 User-Defined Function (UDF) Centric

User-defined functions (UDFs) enable users to design custom functions that may be called from within SQL queries, offering a powerful way to expand the capabilities of relational database management systems (RDBMS). Using this method, deep learning models or particular procedures associated with deep learning tasks are contained in UDFs. These consist of models that have already been trained or models that have been trained inside a database. The RDBMS-supported programming languages can be used to implement these UDFs. Additionally, these UDFs can call many low-level or high-level libraries, like Eigen (2020) or TensorFlow, without the need to include an external runtime. On the other hand, this can conflict with the host relational system's runtime. Moreover, co-optimization is challenging to accomplish because UDFs are typically

opaque to the system. This method also fails for huge DNNs, when the input tensor exceeds the memory capacity or the memory is not large enough to fit the complete model. The intermediate outcomes of machine learning algorithms not fitting into memory is another noteworthy problem.

### 3.2.3 Relation-Centric

In this approach, the DNN inference framework is tightly coupled with the database system. The inference process is modeled as relational algebra operations. The model parameters are treated as a dataset helping in distributed processing thereby achieving model parallelism.

This can even work for models larger than the available memory. The tensors can be split into tensor blocks that fit into memory and the intermediate results can be spilled to disk. The model weight tensor block size can be fine-tuned to ensure better cache locality i.e. fewer cache misses leading to lower latency. Thus, leveraging the strengths of databases - paging/swapping, caching, spilling, pipelining, etcetera makes the inference process feasible. A drawback of this technique is that not all ML operations can be efficiently transformed into relational processes, leading to higher latency, especially for smaller models for which dedicated ML systems like TensorFlow and PyTorch are optimized.

## 3.3 Query Plan Optimization

In the context of databases, a query plan is a blueprint that a database management system (DBMS) creates in order to effectively execute a given database query. The most effective technique to access and modify the data in order to satisfy your request is determined by the database management system (DBMS) when you submit a query to a database.

The process of choosing the most effective execution strategy to carry out a database query is known as query optimization. A query plan's objective is to reduce the amount of time and resources needed to complete the query, including disc I/O, memory utilization, and CPU usage. The following are a few aspects of query optimization:

1. Cost Optimization: This entails calculating the approximate costs of various execution strategies using variables like statistics about the available data, indexes, and system resources. The plan with the lowest estimated cost is selected by the optimizer.
2. Query Rewriting: The query is often rewritten by the optimizer into a more efficient equivalent form. This might involve changing the sequence of joins, tweaking the subqueries, and skipping obsolete actions.
3. Statistics Collection: The DBMS often collects statistics about the tables and indexes to aid in query optimization.
4. On-the-fly optimization: In a few cases, the optimizer dynamically changes the execution plan based on the workload or runtime statistics.

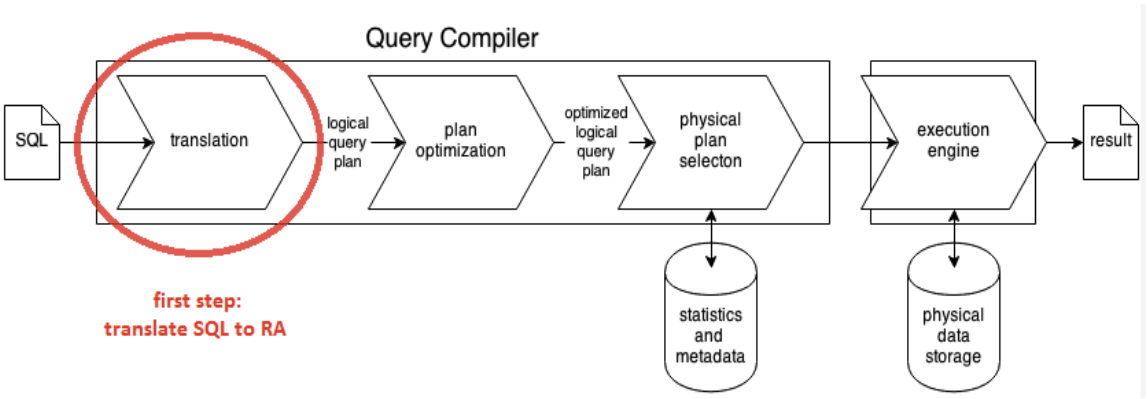


Figure 2: MLWiki. (n.d.). Logical Query Plan Optimization. Retrieved from http://mlwiki.org/index.php/Logical_Query_Plan_Optimization

### 3.4 The Hybrid Framework

Gauging the benefits and drawbacks of the techniques for integrating ML computation with relational databases, a middle ground should be found for supporting different use cases. Thus, to accelerate the inference of DNN models, it is possible to take a hybrid approach by combining the UDF and Relation-Centric approaches. Thus, the goal is to have a system to automatically optimize for deep learning inference queries - no data transfer overhead, co-optimization of Deep Learning computations, and Relational Processing - an adaptive IR that can dynamically choose Relation-Centric or UDF-Centric for any subgraph, and a Unified Resource Management model.

In the hybrid approach of performing DNN inference in a relational database, the UDF encapsulating the inference computation is represented as an intermediate representation (IR) that can be selectively transformed into relational algebra computations for large DNN models.

Here the IR has two levels:

1. Nested relational algebra with each node representing a relational operator like join, filter, projection, aggregation, etc which are customizable using UDFs
2. ML Kernel operators such as matrix multiplication, matrix addition, ReLU, softmax, etcetera

The optimizer can then traverse the two-level IR and transform the linear algebra operations into equivalent relational algebra operations using a cost model.

Consider the example of the matrix multiplication operation (encapsulated by a UDF) of a DNN inference model being transformed into join and aggregation (Yuan et al.,2021) operations by the optimizer.

```
PlanBuilder()
   .filter(" Id > 100")
   .project("model.predict(vector)")
   .planNode();

          ⇓

PlanBuilder()
   .filter(" Id > 100 ")
   .project("relu(matrixAdd(matrixMul(vector,w0) , b0)")
   .planNode();

          ⇓

PlanBuilder()
   .values({data})
   .filter(" Id > 100 ")
   .Join ("select Mul(vector, w0) as Pairs where vector.c = w0.r")
   .Aggregate ("Group by (vector.r, w0.c), sum(Pairs) as Result")
   .project("relu(matrixAdd(Result, b0))")
   .planNode();
```

Figure 3: UDF-Centric to Relation-Centric Transformation

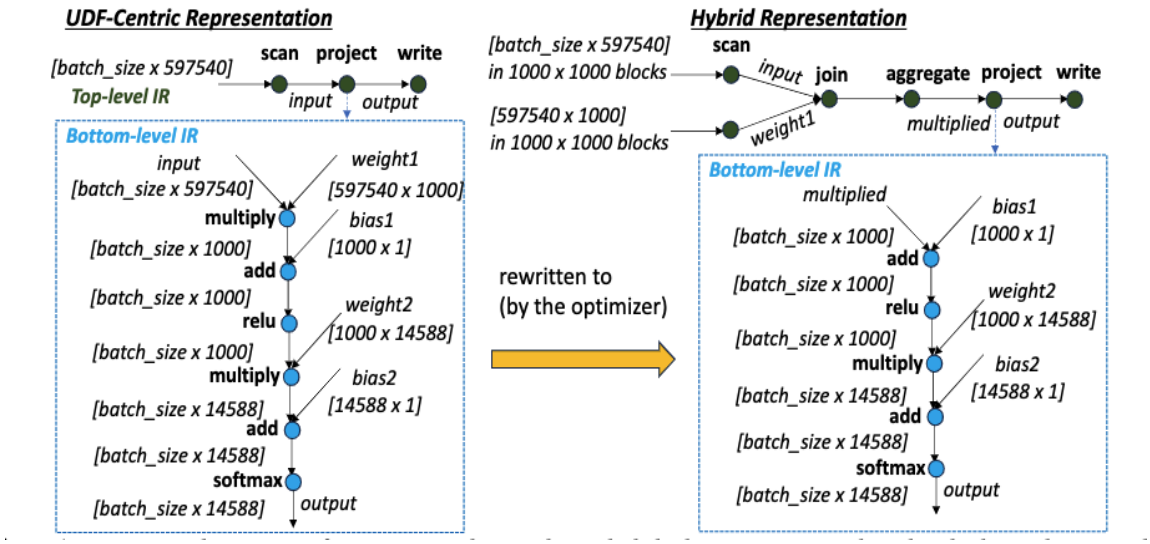


Figure 4: Optimizer Transforming Matrix Multiplication to Join & Aggregation

### 3.5 Velox Execution Engine

("Velox: Open-source execution engine," 2023) Despite their apparent differences at first glance, data computation engines are all made up of the same essential parts: an execution runtime, an optimizer, an intermediate representation (IR), a language interface, and an execution engine (Figure 2). Velox (Pedreira et al., 2022) provides the building blocks required to build execution engines. These building blocks cover a variety of data-intensive activities carried out on a single host, such as expression evaluation, aggregation, sorting, and joining—a.k.a. the data plane. As a result, Velox uses local host resources to effectively execute an optimized plan that it predicts as input. After the IR has been fully optimized by the optimizer, Velox receives it and builds pipeline stages for plan execution.

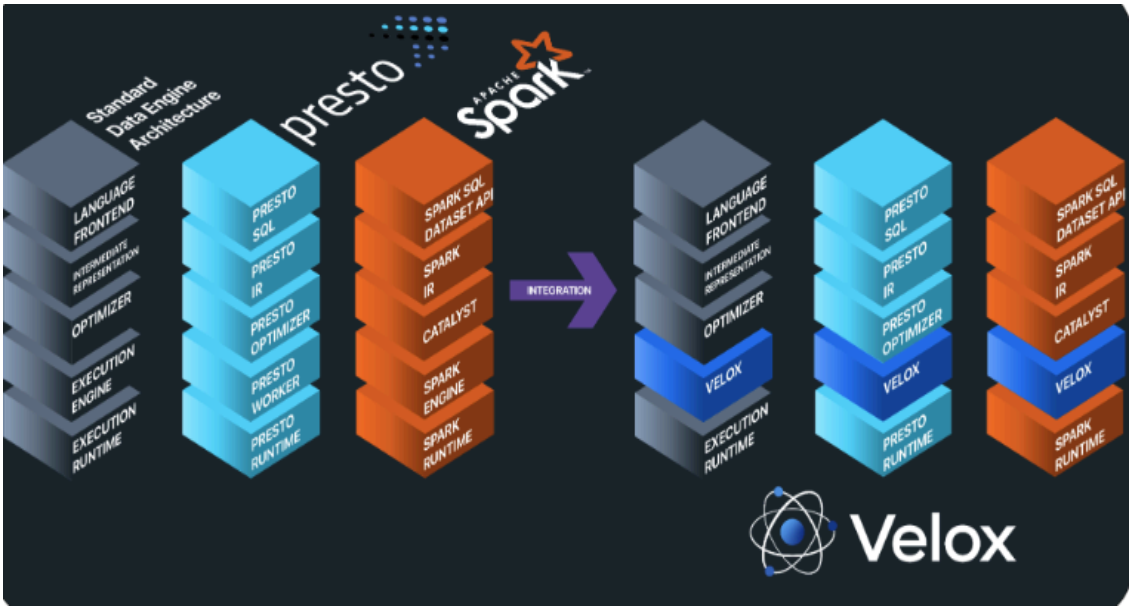


Figure 5. Velox as Execution Engine (Diagram by Philip Bell). Retrieved from Engineering @ Facebook: https://engineering.fb.com/2023/03/09/open-source/velox-open-source-execution-engine/

Velox leverages a multitude of runtime optimizations, including dynamic filter pushdown, adaptive column prefetching, key normalization for array and hash-based aggregations and joins, and filter and conjunct reordering. These optimizations use statistics and insights from incoming data batches to improve local efficiency. In addition, Velox is carefully designed to manage the kinds of complicated data that are common in contemporary workloads. It heavily depends on dictionary encoding for operations like joins and filtering that either raise or lower cardinality, while still providing quick routes for simple data types.

("Velox: Open-source execution engine," 2023) describes the following components

- Type: It may express scalar, complex, and nested data types, such as structs, maps, arrays, functions (lambdas), decimals, tensors, and more, using a generic type system.

- Vector: a columnar memory layout module compatible with Apache Arrow that supports a lazy materialization pattern, out-of-order buffer population, and different encodings, including flat, dictionary, constant, sequence/RLE, and frame of reference.
- Expression Eval: a cutting-edge vectorized expression evaluation engine that uses methods like dictionary peeling, constant folding, encoding-aware evaluation, fast null propagation, common subexpression removal, and memoization.
- Functions: APIs that offer a row-by-row and vectorized (batch by batch) interface for scalar functions and an API for aggregate functions, to create custom functions.
- Operators: application of widely used SQL operators, including OrderBy, TopN, HashJoin, MergeJoin, Unnest, Project, Filter, Aggregation, and Exchange/Merge.
- I/O: API's for integrating with other runtimes:
    - Connectors: permits data sources and sinks to be customized by developers specifically for TableScan and TableWrite operators.
    - DWIO: an expandable interface with support for encoding and decoding widely used file formats, including DWRF, Parquet, and ORC.
    - Storage adapters: a byte-based extendable interface that enables Velox to link to a variety of storage systems, including HDFS, Tectonic, and S3.
    - Serializers: a serialization interface that supports Spark's UnsafeRow format and PrestoPage, aimed at network communication and capable of implementing many wire protocols.
- Resource management: a group of primitives for managing computational resources,

including caching for SSD, spilling, and CPU and memory management.

# CHAPTER 4

# ML INFERENCE IN VELOX

Velox requires an optimized query plan as input for execution. The computations of an inference workload are represented as a Velox query plan by utilizing the various Velox components. The following are the steps for executing an inference workload:

## 4.1 Velox Logical Plan

According to Facebook Incubator (n.d.), Velox's logical plan is a tree of Plan Nodes. Each PlanNode has zero or more child PlanNodes. There are different types of plan nodes like Filter, Project, Aggregation, HashJoin, TableScan, etcetera. Plan Nodes can have user-defined expressions that are evaluated by the Velox execution engine during query execution. Thus, for any inference workload, the constituent ML kernels can be implemented as UDFs and passed to the Plan Node as expressions. The logical plan is then converted into a set of pipelines. Each pipeline comprises a linear sequence of operators corresponding to a linear sub-tree of the plan. The plan tree is broken down into a set of linear sub-trees by disconnecting all but one child node from each node that has two or more children as shown in Figure 6.

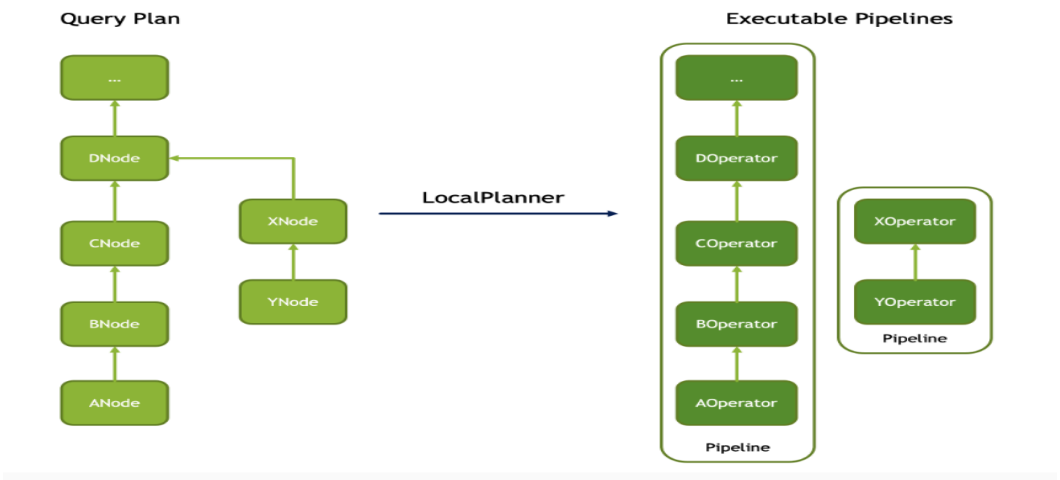


Figure 6. Illustration of Velox Operators Functionality. Adapted from "Velox Operators" by Facebook Incubator. Retrieved from https://facebookincubator.github.io/velox/develop/operators.html

## 4.2 Velox Tasks, Operators, Splits and Drivers

As per Facebook Incubator (n.d.), the Velox task is responsible for converting a query plan into a set of pipelines. Each pipeline is made up of operators stacked on top of each other, where each operator is obtained from one or more plan nodes. Each pipeline is executed by a single driver (Velox thread). The task receives data from the source/leaf nodes like TableScan in the form of splits. The splits are consumed by the operators in the pipeline for execution.

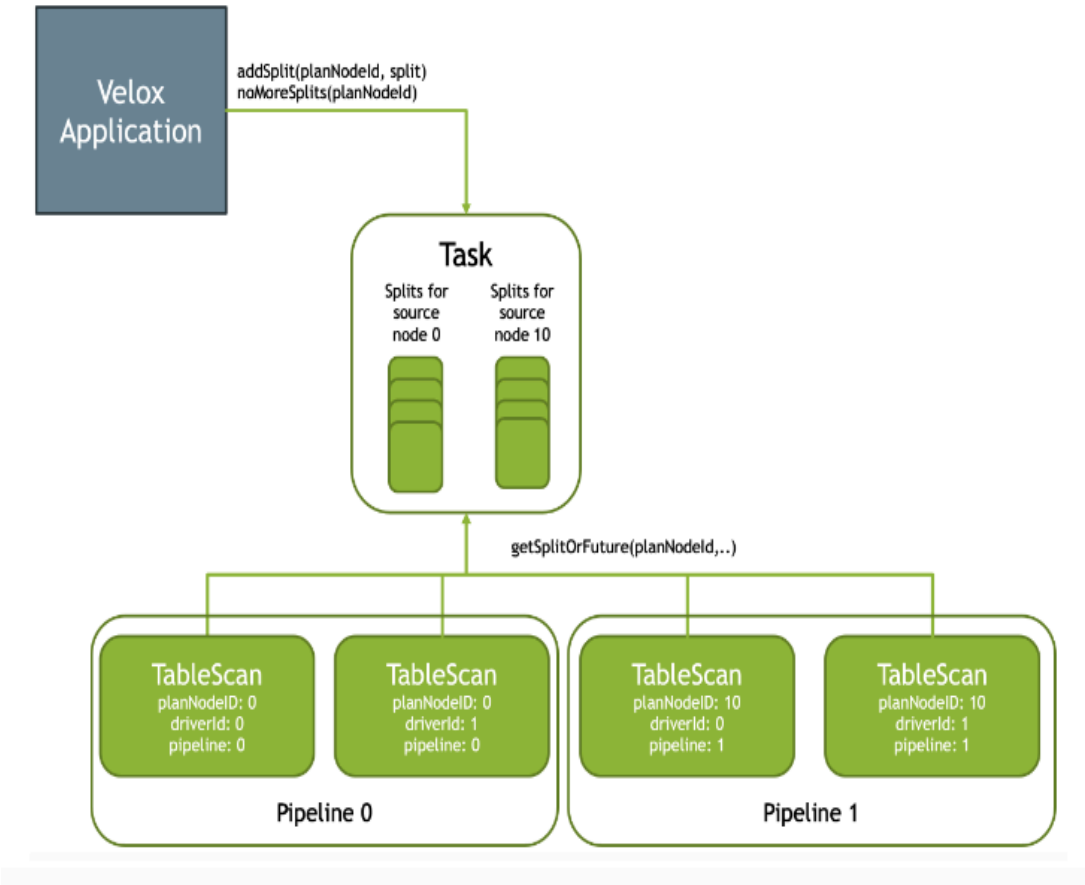


Figure 7. Velox Task execution. Adapted from "What's in the Task" by Facebook Incubator.Retrieved from https://facebookincubator.github.io/velox/develop/task.html

## 4.3 ML Kernels as UDF

### 4.3.1 Design

Velox provides Vector Functions that process a batch of rows and produce a vector of results. This makes it extremely convenient to implement ML kernels efficiently, similar to many ML-centric systems like Torch and TensorFlow. Building upon the design of the

Velox Vector function, a virtual class called MLFunction, which inherits from VectorFunction, is introduced. Subsequently, the ML core functions are implemented by overriding the apply function with custom logic. This logic involves invoking external libraries such as Eigen, Torch, XgBoost (Chen et al.,2016), etcetera.

To summarize, the UDFs-based expression aligns with the design of the Vector Function. The ML functions are encapsulated within the Vector Functions and the logical plan is constructed utilizing these functions.

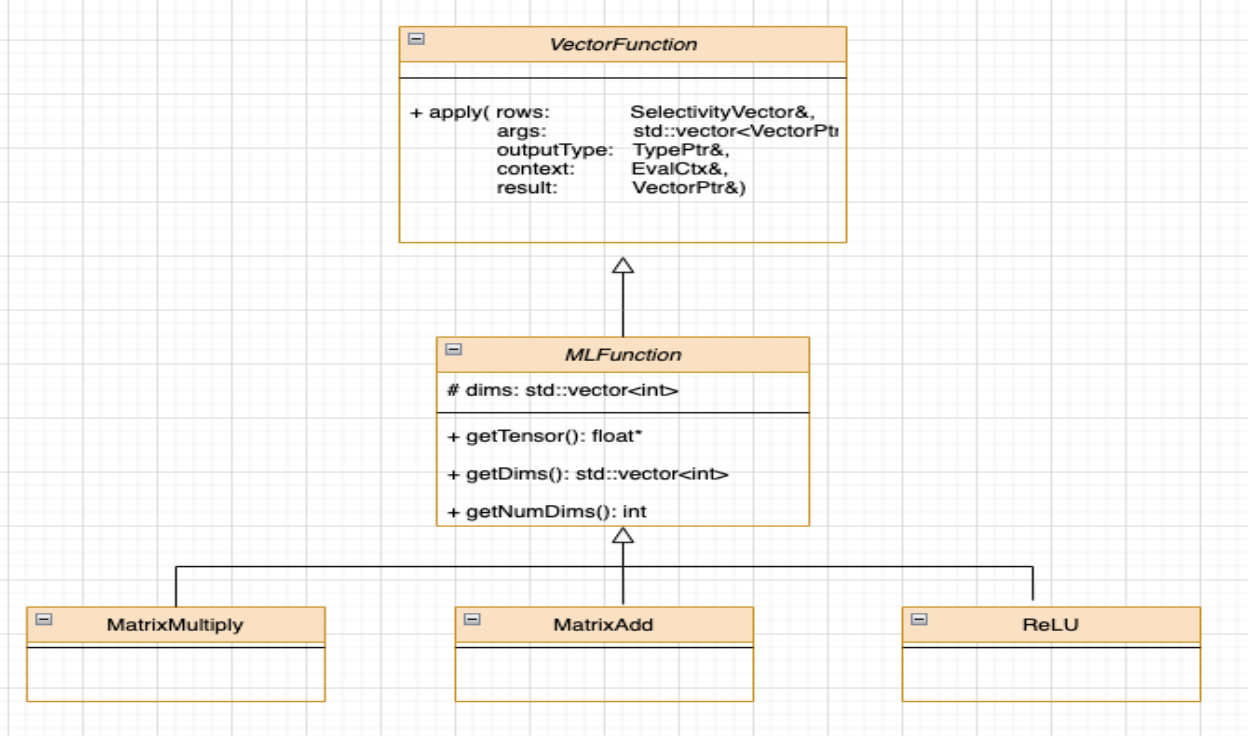


Figure 8: Class Diagram for MLFunction

### 4.3.2 Matrix Multiplication

The matrix multiply kernel is used to multiply two matrices. In the context of DNN inference, it is used to obtain the product of the input and the weight of a single layer of the DNN in forward propagation. It is implemented using the Eigen library. First, in the constructor, the weight of the DNN model is saved in the private member weights_ which is a pointer to a float array. The dimensions of the DNN layer are also passed to the constructor so that the one-dimensional weights_ array can be mapped to a two-dimensional matrix for multiplication.

```
MatrixMultiply(float* weights, int num_rows, int num_cols) {
    weights_ = weights;
    dims.push_back(num_rows);
    dims.push_back(num_cols);
}
```

In the overridden apply() method, the matrix multiplication of the inputs and weights is performed. First, the two-dimensional input array vector is flattened to a one-dimensional vector. Then the input vector and the weight vector are mapped to Eigen matrices m1 and m2. Here, the address space of the vectors is mapped so there is no additional copy of the underlying values of the vectors. Then both matrices are multiplied

and the result is mapped to an array vector to be returned to the caller. The code for matrix multiplication is provided in Appendix A.

```
int input_size = input_elements->size();
Eigen::Map<Eigen::Matrix<float,Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>>
m1(input_values, input_size/dims[0], dims[0]);
Eigen::Map<Eigen::Matrix<float,Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>>
m2(weights_, dims[0], dims[1]);
Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>m= m1 * m2;
```

### 4.3.3 Matrix Addition

As the name suggests, this kernel is used to add two matrics. Concerning DNN, it is used to add the bias values to the output of matrix multiplication from the previous step. The matrix addition kernel is similar to the matrix multiplication kernel with a couple of differences. First, we only pass the number of columns in the constructor of the MatrixAddition. This is because the weight and the input matrics have the same dimensions, and the number of rows can be inferred from the number of input samples.

```
MatrixAddition(float* weights, int num_cols) {
    weights_ = weights;
    dims.push_back(num_cols);
```

}

Second, we use the Eigen's matrix addition to get the final result. The code for matrix addition is provided in Appendix A.

```
Eigen::Map<Eigen::Matrix<float,Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>>
m1(input_values, rows.size(), dims[0]);
Eigen::Map<Eigen::Matrix<float,Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>>
m2(weights_, rows.size(), dims[0]);
Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>m= m1 + m2;
```

#### 4.3.4 Activation Kernels

The final step in the forward propagation step at each layer is passing the result obtained after matrix multiplication and matrix addition through an activation function. This is a non-linear function, and its output decides the neurons in the DNN that will be "activated". ReLU is a commonly used activation function for the input and hidden layers while softmax is a popular function used for the output layer.

##### 4.3.4.1 ReLU

This ML kernel transforms the input such that only positive numbers retain their value

while the negative ones are converted to 0.

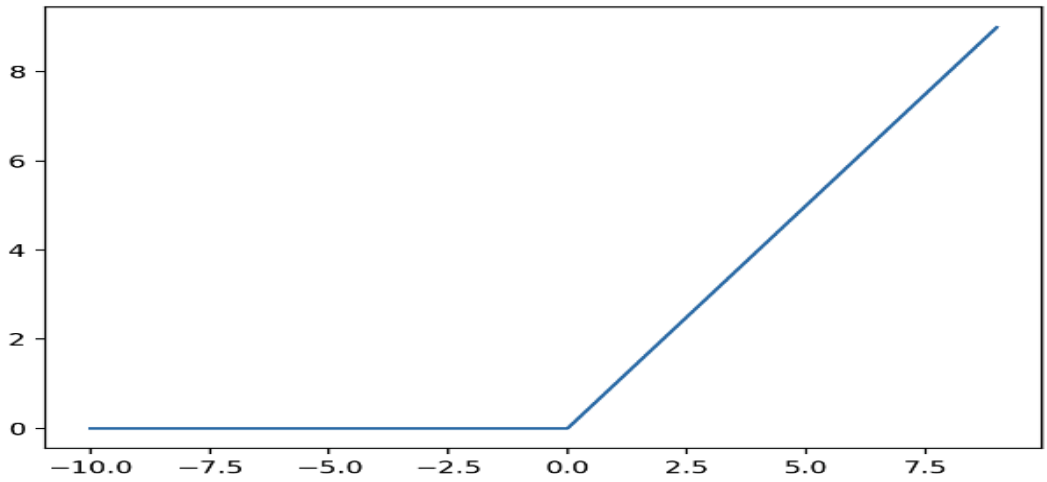


Figure 9: Brownlee, J. (2020, August 20). Line Plot of Rectified Linear Activation for Negative and Positive Inputs. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

The implementation of the kernel is straightforward. The input is transformed, one row at a time, with each negative value replaced by zero and a positive value retained as is. The comple implementation can be found in Appendix A.

```
for (int i = 0; i < num_rows; i++) {
        std::vector<float> rowResult(num_cols);
        std::transform(input_values + i*num_cols, input_values + (i+1)*num_cols,
        rowResult.data(), relu_function);
        result.push_back(rowResult);
}
```

#### 4.3.4.2 Softmax

This kernel is used to transform a vector of integers into a probability distribution. In the context of DNN, especially classification, it is used to estimate the likelihood of each class. Mathematically, softmax can be represented as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Here $z_i$ is the score for class i, K is the total number of classes and e is the base of the natural log.

The kernel is implemented using Eigen. First, the exponentiation for each value is calculated

```
Eigen::Map<Eigen::Matrix<float,Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>>
m(input_values, num_rows, num_cols);
Eigen::ArrayXXf exp = m.array().exp();
```

Next, the row-wise sum is calculated for the matrix. This corresponds to the denominator in the softmax formula.

```
Eigen::ArrayXXf sum = exp.rowwise().sum();
```

Finally, the probability for each class in each sample is calculated by dividing the exponentiation with the row-wise sum. Appendix A contains the complete implementation of this method.

```
for (int i = 0; i < exp.rows(); i++) {
    exp.row(i) /= sum(i);
}
```

### 4.3.5 TorchDNN

Each layer of a DNN is implemented by chaining the matrix multiplication, matrix addition, and an activation kernel. However, using the torch library, the entire DNN can be implemented using a single ML kernel.

First, the TorchDNN class is initialized by passing the weights and bias vector along with the input dimensions of each layer.

```
TorchDNN(float** weights, float** bias, std::vector<int> dimensions) {
    this->weights = weights;
    this->bias = bias;
    dims = dimensions;
}
```

The dimensions are used to create the layers while the weights and bias vectors are used to initialize them.

```
torch::nn::Linear dense1(dims[0], dims[1]);
torch::nn::Linear dense2(dims[1],dims[2]);
torch::nn::ReLU relu;
dense1->weight.set_data(weightTensor1);
dense2->weight.set_data(weightTensor2);
dense1->bias.set_data(bias1);
dense2->bias.set_data(bias2);
```

Once the layers are initialized, a forward pass is performed to obtain the classification results.

```
torch::Tensor layer1_output = dense1->forward(input);
torch::Tensor reluOutput = relu->forward(layer1_output);
torch::Tensor layer2_output = dense2->forward(reluOutput);
torch::Tensor softmax_output = torch::nn::functional::softmax(layer2_output, 1);
```

## 4.4 Hyperparameters

The performance of the ML kernels and the workload, in general, is determined by the hyperparameter setting. Having the optimal values for the following parameters leads to the best performance:

### 4.4.1 Splits

The input data is divided into chunks called splits before it is sent to the ML kernels for execution. Each split is executed by a Velox thread (driver) so increasing the number of splits can increase parallelism. However, having a higher number of splits isn't always optimal. For instance, creating 8 splits on a machine with 8 CPU cores may be more efficient than creating 16 splits on the same machine since only 8 splits can be executed concurrently. Further, executing a larger split can be more optimal than executing two smaller splits since the Eigen library in ML kernels can better exploit SIMD vectorization, and larger matrices can have better cache efficiency.

Velox provides support for creating hive splits from a DWRF file.

```
auto hiveSplits =  makeHiveConnectorSplits(file->path, num_splits,
dwio::common::FileFormat::DWRF);
```

These splits can then be added to the task for execution:

```
for(auto& split : hiveSplits) {
    task->addSplit(p0, exec::Split(std::move(split)));
}
```

### 4.4.2 Drivers (Velox Threads)

The number of Velox drivers or threads represents the total number of pipelines that can be executed concurrently. Each thread is responsible for the execution of a single split. Thus, the number of Velox drivers is bounded by the total number of CPU cores on the machine. Although increasing the number of threads can increase concurrency, the overall execution time may suffer in the case multithreaded libraries are used to implement ML kernels. For instance, both Eigen and Torch have multithreading support. As such, a higher number of Velox threads may interfere with the library threads due to increased context switching, leading to degradation in the total workload execution time.

### 4.4.3 Library Threads

These threads are used to execute the core operations of the ML kernels. For instance, the execution of matrix multiplication in the MatrixMultiply kernel can be optimized by using multiple Eigen threads. But similar to the Velox threads, more library threads don’t always translate to better performance. The optimal performance can be achieved by

striking a careful balance between the Velox threads and library threads.

For Eigen, the number of threads can be set using Eigen::setNbThreads(n). Alternatively, Eigen can use openBLAS if the preprocessor directive #define EIGEN_USE_BLAS is provided.

For Torch, the number of threads is set using torch::set_num_threads(n).

#### 4.4.4 Batch size

The batch size is controlled by max_output_batch_rows and preferred_output_batch_bytes is another important parameter that determines the execution time of the ML kernels. Each split added to the task can in turn be split into batches before being executed by the pipeline. This parameter also needs to be carefully tuned for optimal performance. A very small or large value may fail to leverage SIMD vectorization, cache efficiency, expression optimization, or parallelism.

### 4.5 Feed Forward Neural Network (FFNN) Inference

The Modified National Institute of Standards and Technology database (MNIST) dataset (Deng, 2012) consists of handwritten letters from 0 to 9 and is a popular dataset for image classification benchmarks. This dataset is used here as an example of running an FFNN (Bebis et al., 1994) workload on Velox. The FFNN model consists of an input layer of size 784, a hidden layer of size 1024, and an output layer of size 10 (since there

are 10 digits for prediction).

First, we identify and implement the ML kernels required for this workload. Recall that an FFNN consists of a matrix multiplication of the input vector and the model weights, followed by the addition of the result with the bias term ( represented as a matrix addition), and finally followed by an activation function like ReLU or softmax. Thus matrix multiplication, matrix addition, ReLU, and softmax are required to be implemented for this inference workload. Thus, the following three steps are required:

1）Register the ML functions

```
 exec::registerVectorFunction(
"mat_mul",                                          // name of function
 MatrixMultiplication::signatures(),                // types of inputs and outputs
 std::make_unique< MatrixMultiplication >()         // memory allocation
 );
```

2) Implement of function

```
class MatrixMultiplication: public exec::VectorFunction {
MatrixMultiplication(){}                            // constructor
void apply (){}                                     // main logic
signatures()                                        // define types of input/output
}
```

3) Build the Logical Plan

```
data = {x, vector}
Wo, b: model params
PlanBuilder()
        .values({data})
    .project("relu(mat_add(mat_mul(x, w0), b)")
    .planNode();
```

Here the ML kernels are chained together to form the expression of a Plan Node. This also represents the IR that is understood by the custom optimizer. The input to these kernels is an array of vectors where each vector is a data sample (in the case of the first layer) or an intermediate result.

The results for the benchmark are shown in table 1. This benchmark was performed on AWS EC2 r4.2xlarge (CPU:8, Mem: 61GB) instance. The running time is in seconds.

| Sample(K) | Velox (Eigen) | Velox (Torch) | Tensorflow | Torch |
|---|---|---|---|---|
| 1 | 0.04 | 0.05 | 0.51 | 0.11 |
| 10 | 0.29 | 0.26 | 1.1 | 0.21 |
| 60 | 1.40 | 0.76 | 4.36 | 0.79 |

Table 1: Benchmark results for MNIST Inference workload

# CHAPTER 5

# COST MODEL

As stated earlier, it is possible to selectively transform the UDF-centric implementation into a Relation-centric one. Once the Velox logical plan is built, it can be optimized by the custom optimizer by using some co-optimization rules. To decide whether the final query plan after applying these optimization rules is potentially better than the original one, a cost model is required.

## 5.1 Design

The cost model has the following components

1. CostEstimator: this is the class used by the optimizer to estimate the cost of the query plan. It uses the costModel to estimate the cost of the plan.
2. CostModel: it is used to estimate the cost of a plan node given its sources. It uses a catalog to get information about the sources to estimate the cost
3. Catalog: it is a store of all the data sources that will act as the input to the query plan
4. Source: it represents a data source. It can be of type Node (PlanNode), File, Vector, or Database.
5. Stats: it represents the statistics of a data source. It is used by the cost model to get information about the source so that a cost estimate can be made. For instance, the

cardinality for each column of a table could be stored in the stats which can then be used to estimate the cost of a HashJoin plan node. This class can be extended for different data sources.

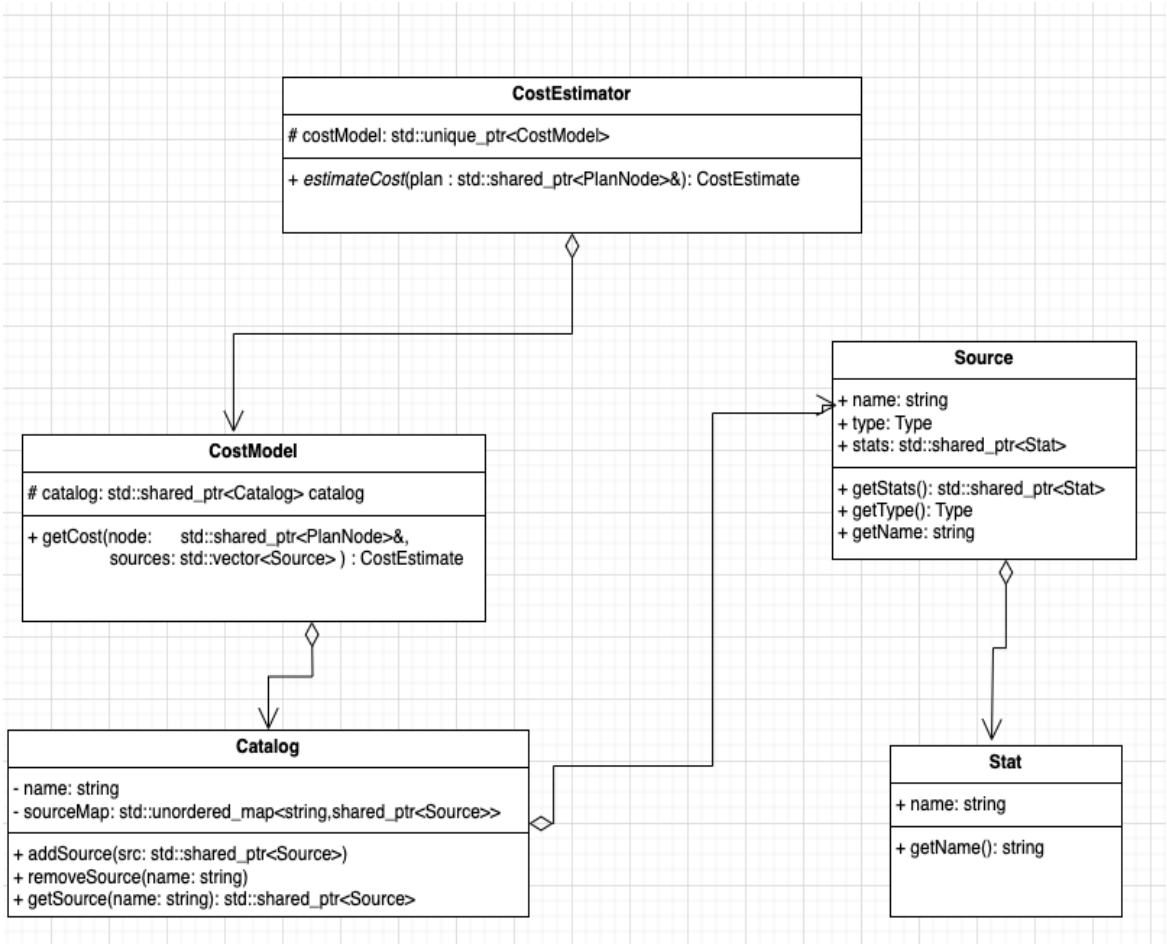


Figure 10: Class Diagram for Cost Model

## 5.2 Implementation

Once a query plan has been created, its input sources are added to the catalog. These sources have been initialized with their respective statistics (Stats).

```
std::shared_ptr<Catalog> catalog = std::make_shared<Catalog>(Catalog("test-catalog"));
catalog->addSource(std::make_shared<Source>(src1));
catalog->addSource(std::make_shared<Source>(src2));
```

Now, a CostModel can be created and passed to a CostEstimator which will estimate the entire cost of the plan. Here, CostEstimate object contains the estimated cost along with the estimated number of output rows and columns.

```
CostModel* cm = new SimpleCostModel(catalog);
CostEstimator* ce = new SimpleCostEstimator(std::unique_ptr<CostModel>(cm));
CostEstimate cost = ce->estimateCost(plan);
```

The cost model is similar to a database query optimizer. It traverses the Velox query plan and estimates the cost for each Plan Node. Since the query plan is a tree, a depth-first search is used to obtain the cost of all the children of a Plan Node before calculating and returning its own cost. For Plan Nodes like HashJoin and Filter nodes, the costModel uses the statistics like cardinality of the data source to estimate the cost. Each operator in the query plan also has a weight *w* configured. After getting an estimate of the cost of a Plan Node, this weight is multiplied with it to get a weighted cost. For instance, the value of w can be 1.0 for HashJoin and 0.5 for Filter operation. This value is user-configurable and is set by carrying out benchmarks.

The cost of the ML kernels is computed similarly. The cost model traverses the query plan and parses each expression in the Plan Nodes. If a node has UDFs in the expression, it gets the handle to the function pointer of the UDF from the Velox function registry and invokes the *getCost()* method of the respective UDF class to get an estimate of the UDF's cost. The cost function of the UDFs uses the input and model size to get a cost estimate. Similar to the other relational operations, UDFs also have an associated weight that helps to get the weighted cost. Finally, the cost model adds up the cost for each Plan Node and returns the cost estimate for the entire plan. Kindly refer to Appendix B for the implementation details.

# CHAPTER 6

# CONCLUSION & FUTURE WORK

This work discussed a hybrid approach to executing Deep Learning inference workloads for improved performance. By adaptively choosing between UDF-centric and Relation-centric approaches, one gets the best of both worlds. Thus, the system can support workloads of varying nature and provide performance comparable to specialized systems.

The work also proposes a couple of interesting optimization problems. First, finding the optimal configuration for the number of Velox threads and Eigen / Torch threads is non-trivial since they both are in the same runtime (share the same resources). Figures 4, 5, and 6 depict the variability in performance for different numbers of Velox and Torch threads over different samples for a CNN workload. The best running time is highlighted in red. Further, there is no heuristic for obtaining the optimal performance when it comes to the batch size and number of data splits in Velox. To add to the complexity, the hardware configuration also determines the performance of different configurations. Since the search space is large, a grid search approach for these hyperparameters will not be feasible. Thus, a hyperparameter optimization approach like Hyperband (Li et al., 2018) may be worth exploring.

Second, the cost model currently depends on the user-provided weight coefficient for estimating the cost of the query plan. Such a configuration is also non-trivial and requires

a large number of benchmarks to get it right. Perhaps exploring a probabilistic model will be more helpful in this direction.

Finally, inference result caching using vector databases is also a promising direction to explore for accelerating the inference workloads.


Figure 11: Velox Threads vs Torch Threads CNN Benchmark for 64 samples

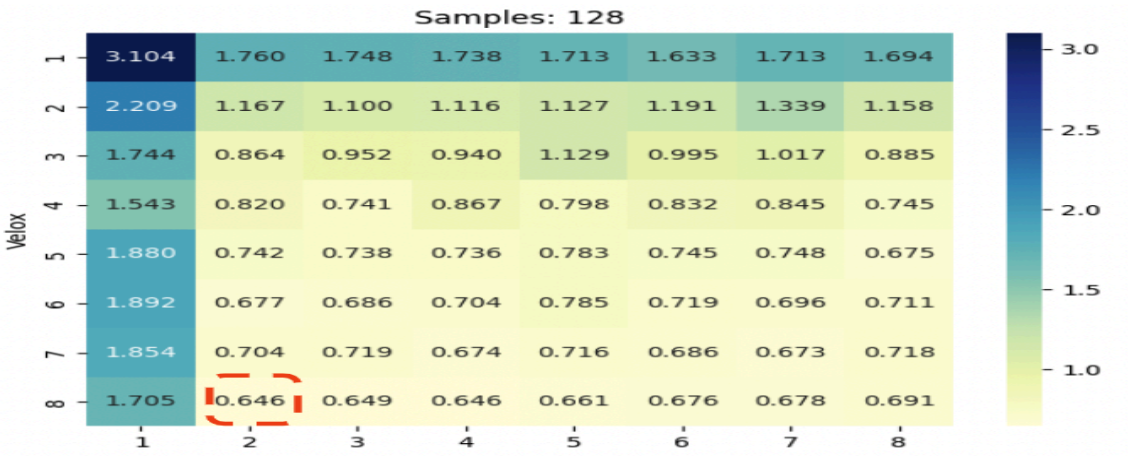


Figure 12: Velox Threads vs Torch Threads CNN Benchmark for 128 samples

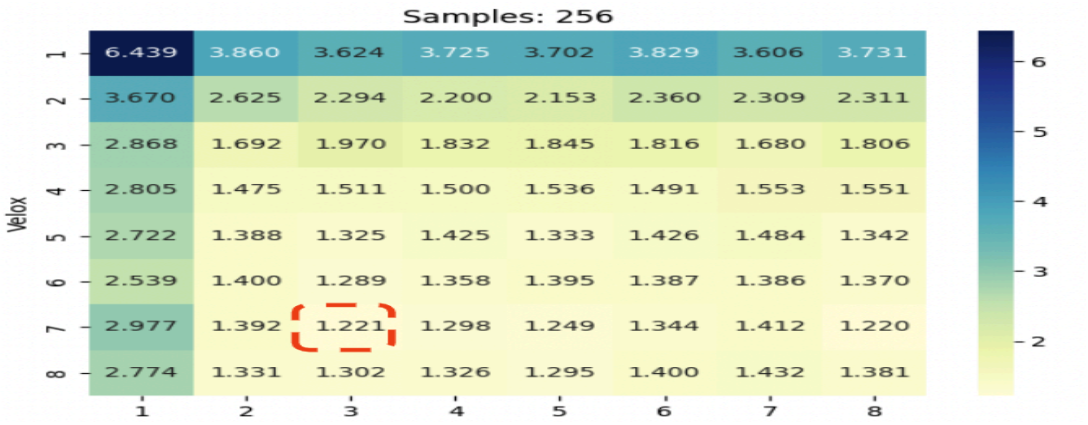


Figure 13: Velox Threads vs Torch Threads CNN Benchmark for 128 samples

## REFERENCE


**Lixi Zhou, Qi Lin, Kanchan Chowdhury, Saif Masood, Alexandre E. Eichenberger, Hong Min, Alexander Sim, Jie Wang, Yida Wang, Kesheng Wu, Binhang Yuan, Jia Zou:Serving Deep Learning Models from Relational Databases. EDBT 2024: 717-724**

Lixi Zhou, Jiaqing Chen, Amitabh Das, Hong Min, Lei Yu, Ming Zhao, Jia Zou: Serving Deep Learning Models with Deduplication from Relational Databases. Proc. VLDB Endow. 15(10): 2230-2243 (2022)

Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, Chris Jermaine: Lachesis: Automated Partitioning for UDF-Centric Analytics. Proc. VLDB Endow. 14(8): 1262-1275 (2021)

Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, Zekai J. Gao:Declarative Recursive Computation on an RDBMS: or, Why You Should Use a Database For Distributed Machine Learning. SIGMOD Rec. 49(1): 43-50 (2020)

Dawei Cheng, Sheng Xiang, Chencheng Shang, Yiyi Zhang, Fangzhou Yang, and Liqing Zhang.2020. Spatio-temporal attention-based neural network for credit card fraud detection. In Proceedings of the AAAI conference on artificial intelligence, Vol. 34. 362–369

**Guan, H., Masood, S., Dwarampudi, M., Gunda, V., Min, H., Yu, L., Nag, S. and Zou, J., 2023, October. A Comparison of End-to-End Decision Forest Inference Pipelines. In Proceedings of the 2023 ACM Symposium on Cloud Computing (pp. 200-215).**

Neil Katkov. 2022. OPERATIONALIZING FRAUD PREVENTION ON IBM Z16. https://www.ibm.com/downloads/cas/DOXY3Q94

Amazon’s recommendation algorithm drives 35% of its sales. Evdelo. (2020, June 28). https://evdelo.com/amazons-recommendation-algorithm-drives-35-of-its-sales/

Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflowserving: Flexible, high-performance ml serving. arXiv preprint arXiv:1712.06139 (2017).

Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. 2020. Elastic machine learning algorithms in amazon sagemaker. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 731–737

Roger Barga, Valentine Fontama, Wee Hyong Tok, and Luis Cabrera-Cordon. 2015. Predictive analytics with Microsoft Azure machine learning. Springer

Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. 322–337.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–15

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. Highthroughput generative inference of large language models with a single gpu. arXiv preprint arXiv:2303.06865 (2023).

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. IEEE Transactions on Big Data 7, 3 (2019), 535–547.

Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In Proceedings of the 2021 International Conference on Management of Data. 2614–2627

Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. (2023). LEON: A New Framework for ML-Aided Query Optimization. PVLDB, 16(9), 2261-2273. doi:10.14778/3598581.3598597

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org

2023. Pinecone: Vector Database for Vector Search. https://www.pinecone.io/

Oracle. (2022). Oracle Machine Learning Documentation. Retrieved from https://docs.oracle.com/en/database/oracle/machine-learning/

IBM. (2019). Db2. Retrieved from https://www.ibm.com/products/db2

arXiv. (2023, January 6). Systems for Parallel and Distributed Large-Model Deep Learning Training [Preprint]. arXiv:2301.02691v1.

2020. Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page

Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. Proc. VLDB Endow.14, 8 (2021), 1338–1350 https://doi.org/10.14778/3457390.3457399

Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, Biswapesh Chattopadhyay. Velox: Meta's Unified Execution Engine. PVLDB, 15(12): 3372 - 3384, 2022. doi:10.14778/3554821.3554829

Engineering @ Facebook. (2023, March 9). Velox: Open-source execution engine. Facebook Engineering. Retrieved from https://engineering.fb.com/2023/03/09/open-source/velox-open-source-execution-engine/

Facebook Incubator. (n.d.). Velox Operators. Facebook Incubator.https://facebookincubator.github.io/velox/develop/operators.html

Facebook Incubator. (n.d.). What's in the task. Facebook Incubator.https://facebookincubator.github.io/velox/develop/task.html

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 785–794

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), 141–142.

G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," in IEEE Potentials, vol. 13, no. 4, pp. 27-31, Oct.-Nov. 1994, doi: 10.1109/45.329294

Intel Corporation. (2023). Intel Math Kernel Library.[Software]. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In Proceedings of the 17th European Conference on Machine Learning (pp. 282–293). Springer-Verlag.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar.Hyperband: A novel bandit-based approach to hyperparameter optimization. Journal of Machine Learning Research, 18(185):1–52, 2018

# APPENDIX A

## ML KERNEL CODE

## A.1 Matrix Multiplication

```
class MatrixMultiply: public MLFunction {

public:
   MatrixMultiply(float* weights, int num_rows, int num_cols) {
      weights_ = weights;
      dims.push_back(num_rows);
      dims.push_back(num_cols);
   }

    MatrixMultiply(std::string weightsFile, int num_rows, int num_cols) {
      weightsFile_ = weightsFile;
      dims.push_back(num_rows);
      dims.push_back(num_cols);
   }


   void apply(
      const SelectivityVector& rows,
      std::vector<VectorPtr>& args,
      const TypePtr& type,
      exec::EvalCtx& context,
      VectorPtr& output) const override {

      BaseVector::ensureWritable(rows, type, context.pool(), output);
      auto input_elements = args[0]->as<ArrayVector>()->elements();
      float* input_values = input_elements->values()->asMutable<float>();
      int input_size = input_elements->size();

      Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic,
      Eigen::RowMajor>> m1(input_values, input_size/dims[0], dims[0]);

      Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic,
      Eigen::RowMajor>> m2(weights_, dims[0], dims[1]);

      Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
      m  =  m1 * m2;

      std::vector<std::vector<float>> result;
      for (int i = 0; i < m.rows(); i++) {
         std::vector<float> row(
         m.row(i).data(),
```

```
            m.row(i).data() + m.cols());
            result.push_back(row);
        }
        VectorMaker maker{context.pool()};
        output = maker.arrayVector<float>(result, REAL());
    }

    static std::vector<std::shared_ptr<exec::FunctionSignature>> signatures() {
        return {exec::FunctionSignatureBuilder()
                    .returnType("array(REAL)")
                    .argumentType("array(REAL)")
                    .build()};
    }

    float* getTensor() const override {
        return weights_;
    }

    static std::string getName() {
        return "mat_mul";
    };

    std::string getWeightsFile() {
        return weightsFile_;
    }

    void setWeights(float* weights){
        weights_ = weights;
    }

    CostEstimate getCost(std::vector<int> inputDims){
        float cost = getWeightedCost(getName(), inputDims[0] * inputDims[1] * dims[0] *
        dims[1]);
        return CostEstimate(cost, dims[0], inputDims[1]);
    }


private:
    float* weights_;
    std::string weightsFile_;

};
```

## A.2 Matrix Addition

```
class MatrixAddition: public MLFunction {
public:
   MatrixAddition(float* weights, int num_cols) {
      weights_ = weights;
      dims.push_back(num_cols);
   }

   MatrixAddition(std::string weightsFile, int num_cols) {
      weightsFile_ = weightsFile;
      dims.push_back(num_cols);
   }

   void apply(
      const SelectivityVector& rows,
      std::vector<VectorPtr>& args,
      const TypePtr& type,
      exec::EvalCtx& context,
      VectorPtr& output) const override {

      BaseVector::ensureWritable(rows, type, context.pool(), output);

      auto input_elements = args[0]->as<ArrayVector>()->elements();
      float* input_values = input_elements->values()->asMutable<float>();

      Eigen::Map<Eigen::Matrix<float,Eigen::Dynamic,Eigen::Dynamic,
      Eigen::RowMajor>> m1(input_values, rows.size(), dims[0]);

      Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic,
      Eigen::RowMajor>> m2(weights_, rows.size(), dims[0]);

      Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
      m  =  m1 + m2;

      int result_size = m.size();
      float* data = m.data();

      std::vector<std::vector<float>> result;
      for (int i = 0; i < m.rows(); i++) {
         std::vector<float> row(
         m.row(i).data(),
         m.row(i).data() + m.cols());
         result.push_back(row);
```

```
        }
        VectorMaker maker{context.pool()};
        output = maker.arrayVector<float>(result, REAL());
    }

    static std::vector<std::shared_ptr<exec::FunctionSignature>> signatures() {
        return {exec::FunctionSignatureBuilder()
                    .returnType("array(REAL)")
                    .argumentType("array(REAL)")
                    .build()};
    }

    float* getTensor() const override {
        return weights_;
    }

    static std::string getName() {
        return "mat_add";
    };

    std::string getWeightsFile() {
        return weightsFile_;
    }

    void setWeights(float* weights){
        weights_ = weights;
    }

    CostEstimate getCost(std::vector<int> inputDims){
        float cost = getWeightedCost(getName(),inputDims[0] * inputDims[1] +  dims[0] *
        dims[1]);
        return CostEstimate(cost , inputDims[0], inputDims[1]);
    }

private:
    float* weights_;
    std::string weightsFile_;

};
```

A.3 ReLU

```
class Relu: public MLFunction {
public:
   Relu() {}

   float static relu_function(float x) {
      return (x > 0.0f) ? x : 0.0f;
   }

   void apply(
      const SelectivityVector& rows,
      std::vector<VectorPtr>& args,
      const TypePtr& type,
      exec::EvalCtx& context,
      VectorPtr& output) const override {

      BaseVector::ensureWritable(rows, type, context.pool(), output);

      auto input_elements = args[0]->as<ArrayVector>()->elements();
      float* input_values = input_elements->values()->asMutable<float>();
      int input_size = input_elements->size();
      int num_rows = args[0]->size();
      int num_cols = input_size / num_rows;

      std::vector<std::vector<float>> result;
      for (int i = 0; i < num_rows; i++) {
         std::vector<float> rowResult(num_cols);
         std::transform(input_values + i*num_cols, input_values + (i+1)*num_cols,
         rowResult.data(), relu_function);
         result.push_back(rowResult);
      }
      VectorMaker maker{context.pool()};
      output = maker.arrayVector<float>(result, REAL());
   }

   static std::vector<std::shared_ptr<exec::FunctionSignature>> signatures() {
      return {exec::FunctionSignatureBuilder()
                  .returnType("array(REAL)")
                  .argumentType("array(REAL)")
                  .build()};
   }
```

```
  float* getTensor() const override {
    return new float[0];
  }

  static std::string getName() {
    return "relu";
  };

  CostEstimate getCost(std::vector<int> inputDims){
    float cost = getWeightedCost(getName(), inputDims[0] * inputDims[1]);
    return CostEstimate(cost, inputDims[0], inputDims[1]);
  }
};
```

## A.4 Softmax

```
class Softmax: public MLFunction {
public:
  Softmax() {}

  void apply(
    const SelectivityVector& rows,
    std::vector<VectorPtr>& args,
    const TypePtr& type,
    exec::EvalCtx& context,
    VectorPtr& output) const override {

    BaseVector::ensureWritable(rows, type, context.pool(), output);

    auto input_elements = args[0]->as<ArrayVector>()->elements();
    float* input_values = input_elements->values()->asMutable<float>();
    int input_size = input_elements->size();

    int num_rows = args[0]->size();
    int num_cols = input_size / num_rows;

    Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic,
    Eigen::RowMajor>> m(input_values, num_rows, num_cols);
    Eigen::ArrayXXf exp = m.array().exp();
    Eigen::ArrayXXf sum = exp.rowwise().sum();
    for (int i = 0; i < exp.rows(); i++) {
      exp.row(i) /= sum(i);
```

```
        }

        std::vector<std::vector<float>> result(num_rows, std::vector<float>(num_cols));
        for (int i = 0; i < num_rows; ++i) {
            for (int j = 0; j < num_cols; ++j) {
                result[i][j] = exp(i,j);
            }
        }

        VectorMaker maker{context.pool()};
        output = maker.arrayVector<float>(result, REAL());
    }

    static std::vector<std::shared_ptr<exec::FunctionSignature>> signatures() {
        return {exec::FunctionSignatureBuilder()
                    .returnType("array(REAL)")
                    .argumentType("array(REAL)")
                    .build()};
    }

    float* getTensor() const override {
        return new float[0];
    }

    static std::string getName() {
        return "softmax";
    };

    CostEstimate getCost(std::vector<int> inputDims){
        float cost = getWeightedCost(getName(), inputDims[0] * inputDims[1]);
        return CostEstimate(cost, inputDims[0], inputDims[1]);
    }
};
```

## A.5 TorchDNN

```
class TorchDNN: public MLFunction {
public:
    TorchDNN(float** weights, float** bias, std::vector<int> dimensions) {
        this->weights = weights;
        this->bias = bias;
        dims = dimensions;
```

```
}

void apply(
    const SelectivityVector& rows,
    std::vector<VectorPtr>& args,
    const TypePtr& type,
    exec::EvalCtx& context,
    VectorPtr& output) const override {

    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
    torch::nn::Linear dense1(dims[0], dims[1]);
    torch::nn::Linear dense2(dims[1],dims[2]);
    torch::nn::ReLU relu;

    torch::Tensor weightTensor1 = torch::from_blob(weights[0], {dims[0], dims[1]}).t();
    torch::Tensor weightTensor2 = torch::from_blob(weights[1], {dims[1], dims[2]}).t();
    torch::Tensor bias1 = torch::from_blob(bias[0], {dims[1]});
    torch::Tensor bias2 = torch::from_blob(bias[1], {dims[2]});

    dense1->weight.set_data(weightTensor1);
    dense2->weight.set_data(weightTensor2);
    dense1->bias.set_data(bias1);
    dense2->bias.set_data(bias2);

    auto input_elements = args[0]->as<ArrayVector>()->elements();
    float* input_values = input_elements->values()->asMutable<float>();
    int input_size = input_elements->size();

    torch::Tensor input = torch::from_blob(input_values, {rows.size(), dims[0]});

    torch::Tensor layer1_output = dense1->forward(input);
    torch::Tensor reluOutput = relu->forward(layer1_output);
    torch::Tensor layer2_output = dense2->forward(reluOutput);
    torch::Tensor softmax_output = torch::nn::functional::softmax(layer2_output, 1);
    float* data = softmax_output.data_ptr<float>();

    std::vector<std::vector<float>> results;
    for (int i = 0; i < rows.size(); ++i) {
            std::vector<float> result(data + i*dims[2], data+ (i+1)*dims[2]);
       results.push_back(result);
    }
    VectorMaker maker{context.pool()};
    output = maker.arrayVector<float>(results, REAL());
}
```

```
    static std::vector<std::shared_ptr<exec::FunctionSignature>> signatures() {
      return {exec::FunctionSignatureBuilder()
                .returnType("array(REAL)")
                .argumentType("array(REAL)")
                .build()};
    }

    float* getTensor() const override {
      return new float[0];
    }
    float** getWeights() const {
      return weights;
    }

    float** getBias() const {
      return bias;
    }

    static std::string getName() {
      return "torch_dnn";
    };

    CostEstimate getCost(std::vector<int> inputDims){
      float cost = getWeightedCost(getName(), inputDims[0] * inputDims[1] * dims[0] *
      dims[1]);
      return CostEstimate(cost, inputDims[0], inputDims[1]);
    }

    private:
      float** weights;
      float** bias;
};
```

# APPENDIX B

# COST MODEL

## B.1 Cost Estimator

```
class CostEstimator {

   protected:
      std::unique_ptr<CostModel> costModel;

   public:
      virtual ~CostEstimator() = default;

   CostEstimator(std::unique_ptr<CostModel> model): costModel(std::move(model)) {}

  virtual CostEstimate estimateCost(std::shared_ptr<const core::PlanNode>& plan) const
  = 0;
};


class SimpleCostEstimator : public CostEstimator {

   public:

      SimpleCostEstimator(std::unique_ptr<CostModel>model):
      CostEstimator(std::move(model)) {}

     CostEstimate estimateCost(std::shared_ptr<const core::PlanNode>& plan) const
     override {
         if(!plan)
            return CostEstimate(0,0,0);

         std::vector<Source> sources;
         // total cost of all the sources for the current node
         // this will be added to the cost of the current node
         // to get the total cost so far including the current node

         float srcCost = 0.0; // the current node
         for (auto source : plan->sources()) {

            CostEstimate estimate =  estimateCost(source);

            srcCost += estimate.cost;
            std::shared_ptr<OutputStat> stat = std::make_shared<OutputStat>
            (OutputStat(estimate.outputRows , estimate.outputCols));
            // inputs to the current node
            // output from previous step is input to the current node
```

```
            // hence the stat is the output stat of the sources
              sources.push_back(Source(std::string(plan->name()),
              Source::Type::NODE, stat));
          }

          CostEstimate estimate = costModel->getCost(plan, sources);
          // total cost so far
          estimate.cost += srcCost;
          return estimate;
      }
};
```

## B.2 Cost Model

```
class CostModel {

   public:

      enum PlanNodeType {
         Filter,
         Project,
         Aggregation,
         HashJoin,
         OrderBy,
         TableScan,
         NONE
      };

      virtual ~CostModel() = default;

      CostModel(std::shared_ptr<Catalog> catalog) : catalog(catalog) {}

      // node needs a stat (which isn't necessarilty a source stat)
      virtual CostEstimate getCost(std::shared_ptr<const core::PlanNode>& node,
      std::vector<Source> sources) = 0;
      // if else on node type and handle each type .. e.g projections, filters, etc

      PlanNodeType hashPlanNode(std::string_view name) {
         if (name == "Filter") return Filter;
         if (name == "Project") return Project;
         if (name == "TableScan") return TableScan;
         if (name == "HashJoin") return HashJoin;
```

```
        if (name == "OrderBy") return OrderBy;
        if (name == "Aggregation") return Aggregation;
        return NONE;
    }

protected:
    std::shared_ptr<Catalog> catalog;

};

class SimpleCostModel: public CostModel {

public:
    SimpleCostModel(std::shared_ptr<Catalog> catalog): CostModel(catalog) {}

    CostEstimate getCost(std::shared_ptr<const core::PlanNode>& node,
    std::vector<Source> sources) override {

        if(!node)
            return CostEstimate(0,0,0);

        // it is a leaf node
        if(sources.empty()){
            return handleLeafNode(node);
        } else {
            return handleInternalNode(node, sources);
        }
    }

private:

    CostEstimate handleLeafNode(std::shared_ptr<const core::PlanNode>& node){

        std::shared_ptr<Source> src = catalog->getSource(node->id());
        if(!src)
            throw std::runtime_error("Source not found for node: " + node->id() + ":" +
            std::string(node->name()));

        switch (src->getType()) {
            case Source::Type::FILE:
            case Source::Type::DATABASE:
            case Source::Type::NODE:
            case Source::Type::VECTOR: {
```

```
            // for now we're just using output stat but
            // we can use different stats here based on the src type

            std::shared_ptr<OutputStat> stat = std::static_pointer_cast<OutputStat>
(src->getStats());

            // the cost for source node is 0 or constant for a plan
            return CostEstimate(0.0, stat->getRows(), stat->getCols());
        } break;

        default:
            throw std::runtime_error("Source type not supported");
    }
}

CostEstimate handleInternalNode(std::shared_ptr<const core::PlanNode>& node,
std::vector<Source> sources){

    if(sources.empty())
        throw std::runtime_error("Source not found for node: " + node->id() + ":" +
        std::string(node->name()));

    switch (hashPlanNode(node->name())) {
        case Aggregation:
        case OrderBy:{

            std::shared_ptr<OutputStat> stats = std::static_pointer_cast<OutputStat>
             (sources[0].getStats());

             return CostEstimate(stats->getCols() + stats->getRows(), stats->getRows(),
             stats->getCols());

        } break;
        case Filter: {
            float lambda = 0.5;
            std::shared_ptr<OutputStat> stats =  std::static_pointer_cast<OutputStat>
            (sources[0].getStats());

           return CostEstimate(lambda * stats->getRows() , lambda * stats->getRows(),
           stats->getCols());

        } break;
        case HashJoin: {
            float lambda = 0.7;
```

```
            std::shared_ptr<OutputStat> stats1 =
            std::static_pointer_cast<OutputStat>(sources[0].getStats());
            std::shared_ptr<OutputStat> stats2 =
            std::static_pointer_cast<OutputStat>(sources[1].getStats());
            return CostEstimate(lambda * stats1->getRows() * stats2->getRows(),
            lambda * stats1->getRows() * stats2->getRows(), stats1->getCols() +
            stats2->getCols());
        } break;
        case Project: {

            std::shared_ptr<const ProjectNode> projectNode =
            std::dynamic_pointer_cast<const ProjectNode> (node);

            const std::vector<TypedExprPtr> & projections =
            projectNode->projections();

           std::cout << "There are " << projections.size() << " projections." << std::endl;

            for(auto projection : projections){
               handleProjection(projection, sources);
            }


            return CostEstimate(0,1,1);
        } break;

        default:
            throw std::runtime_error("Node type not supported");
    }
}


CostEstimate getUDFCost(const std::vector<std::string> udfs, std::vector<Source>
sources) {

    // get the stat for the source of this node
    std::shared_ptr<OutputStat> stat =
    std::static_pointer_cast<OutputStat>(sources[0].getStats());

    CostEstimate finalEstimate(0, stat->getRows(), stat->getCols());

    vectorFunctionFactories().withRLock([&](auto& functionMap) {

        for(std::string udf : udfs){
```

```
        auto it = functionMap.find(udf);

        if (it != functionMap.end()) {

            std::cout << udf << std::endl;
            core::QueryConfig config({});
            std::shared_ptr<VectorFunction> func = getVectorFunction(udf,
            {ARRAY(REAL())}, {}, config);
            std::shared_ptr<MLFunction> mlFunc =
            std::dynamic_pointer_cast<MLFunction>(func);
            CostEstimate curCost = mlFunc->getCost({finalEstimate.outputRows,
            finalEstimate.outputCols});

            finalEstimate.cost += curCost.cost;
            std::cout << finalEstimate.cost << std:: endl;
            finalEstimate.outputCols = curCost.outputRows;
            finalEstimate.outputCols = curCost.outputCols;
        }
      }
   });
   std::cout<< finalEstimate.cost;
   return finalEstimate;
}


void handleProjection(const core::TypedExprPtr expression,  std::vector<Source>
sources) {

   if (!expression) return;
   // entire expression for udf
   std::string exp = expression->toString();
   std::cout << exp << std::endl;
   // relu(mat_add(mat_mul())
   // (mat_mul -> mat_add -> relu) ()
   std::vector<std::string> udfs = getUDFs(exp);

   // to estimate entire expression cost
   // we have to start with the innermost udf
   // because the input size applies to the
   // innermost udf
   // hence we have to reverse the exp
   // relu(mat_add(mat_mul())) -> mat_mul(mad_add(relu))
   // now we can lookup the udf cost map
```

```
        std::reverse(udfs.begin(), udfs.end());
        getUDFCost(udfs, sources);


    }

    std::vector<std::string> getUDFs(std::string expression) {
        std::istringstream stream(expression);
        std::string token;
        std::vector<std::string> udfs;
        char delimiter = '(';

        while (std::getline(stream, token, delimiter)) {
            udfs.push_back(token);
        }
        // last element is the input
        // hence we have to delete it
        // relu(mat_add(mat_mul(x)))
        if(!udfs.empty())
            udfs.pop_back();

        return udfs;
    }
};
```