Relational Database as a Machine Learning Tool

John Nagel
jknage01@louisville.edu

April 15, 2020

Contents

1 Introduction

Trends in modern computing have led to the accumulation of massive amounts of information in data storage facilities. Fields such as data mining and machine learning seek to make sense of this real world information by identifying trends or patterns. While there are many libraries and applications that can already handle these tasks, they usually are applied separate from the where the data is originally stored. This project will attempt to combine the benefits of relational databases and machine learning into a single resource. The result of this implementation will be called the Machine Learning Database (MLDB).

2 Product Description

2.1 Needs Assessment

Many business applications can benefit from applying machine learning to their data stores. Data analysis can inform decision making processes by both executives and clients. MLDB as described in this paper will focus on two algorithms commonly seen in machine learning, regression and classification. They both can analyze data that can be stored in a relational database. Regression can be used to extrapolate numeric values, while classification can map sets of data to distinct classes. Both of these tasks are supervised, meaning the data the algorithms require, called the input features, must already be matched with a solution, called a label.

The objective of the MLDB is to implement a neural network for machine learning completely within a relational database. There are many theoretical benefits MLDB can achieve when applied to existing software. Most significantly is that exporting data to an external library will not be necessary. MLDB seeks to minimize complex dependencies for client side or backend applications with existing machine learning libraries. With this integration, deploying a model to production is instant. Developers would only need to work in a single backend environment of the existing relational database. The interaction between client and server will use a standard connection to the database, as opposed to having to create a separate API to use the machine learning model.

Other benefits to using a relational database are inherent. Databases were built to scale with large amounts of data and to handle computation across its stores. Applying machine learning within a relational database can leverage exiting I/O capabilities to effectively bring data from disk into RAM. Other libraries can be limited by the size of RAM in terms of how much data can be used to update a model in one cycle. The distributed nature of some database systems can be used to parallelize computaions to speed up training [1].

This product will attempt to balance the complexity and fine control of directly programming a neural network with a framework such as Torch or Keras with the speed and usability of a modular approach. Users will interact with MLDB with an interface that can control most all functionality to manufacture a neural network with an architecture that fits their needs. The creation and use of the machine learning tools will be parameterized for use by a front end application. The limitations of having a user friendly front-end to manipulate the model will naturally have drawbacks, including the last of fine tuning and training time. This initial version of MLDB will primarily be a test to determine the feasibility of integrating machine learning algorithms with a relational database.

2.2 Background and Algorithms

The machine learning principles applied for MLDB will use deep learning elements of an artificial neural network. A neural network consists of a set of individual neurons that accept inputs and compute a non-linear output. A multiplayer feed-forward network has the neurons organized into layers, where the outputs of a layer connect to the inputs of the neurons in the next layer. If every output connects to every neuron in the next layer, then the network is described as fully connected or dense. The layer that accepts external values is called the input layer while the last layer is called the output layer. Any layer in between these two is called the hidden layer. The complexity of an artificial neuron network is meant to simulate a human brain, where sensory inputs cause electrical stimuli in the real neurons that communicate with each other to inform some action by the body [2].

The strength of the pathways between neurons control how a neuron reacts to inputs, denoted by the vector x. The numeric value assigned to the path between neurons is called the weight w, and the output of a neuron is partially determined by the input vector times the weight vector $w \times x$. The $\times$ denotes the dot product of w and x, or $\sum_i x_i y_i$. If this linear output is used, then the final value of a network is simply a linear function of the inputs which intersects at the origin of the coordinate system [3]. This is insufficient to form a hyperplane that can either follow a regression line or separate the input values of multiple classes. A term called the bias b is added to shift the output of each neuron, resulting in the equation $z = w \times x + b$.

Continuing with the limitations of a linear function, the result of each neuron is passed through a function whose output is non-linear. These functions are called the activation function, which mimics whether a neuron in the brain should fire an electrical impulse or not. Examples of these kinds of functions are the logistic sigmoid or hyperbolic tangent. With a function f, the output of a neuron is denoted as $y = f(w \times x + b)$. Computing the output of all neurons for a given input until the final output is called a forward pass or forward propagation.

With activation functions included in the network, the result can approximately match the output of any function in the real number space, assuming the network has the optimal values for the weights and biases. The Universal Approximation Theorem proves this relationship for networks with at least one hidden layer [3].

When first creating a network, the values of the weights and biases should be initialized to some small number, perhaps less that 1. In order to achieve the optimal values for the network, it must learn how to adjust the values given certain inputs and the expected outputs. The inputs can be the values of certain features of a real item and the output is dependent on the inputs in some way. The difference between the actual y and expected y output of the network is the error, and a function E that describes this error can be called the loss.

For a regression problem, a common loss function called the Mean-Squared Error (MSE) is used to more heavily penalize larger absolute differences between the actual and expected values as described in Equation 7 [3]. For classification, the Cross Entropy Loss (CEL) function in Equation 12 uses multiple output values of a network corresponding to each possible class. Each value is adjusted to form a probability distribution function using softmax in Equation 9, where all values sum to 1 and the value of each output is the probability that the neuron is the correct class. For an actual output number matched to the expected output number, there will be a non-zero loss for any output less than 1. The CEL function is $-ln(y)$ for single label classification [4].

The loss is used to adjust the weight and bias values in the direction that produces smaller loss. This method works directly for the output layer, and must be approximated for the hidden layers. This method of updating the weights and biases starting from the output layer and moving towards the input layer is called back propagation.

The direction towards better values can be calculated using the derivative of the loss function. By changing the weights in the direction opposite of the derivative of the loss function with respect to the weights, called the gradient, the loss can be minimized. This method of updating the weights is called gradient descent. We use a small constant L to change how much we update the weight. The process of changing the weights based on some input data is called training, which makes it possible to output correct responses while testing unseen data.

The simple approach to gradient descent is to update the weights for every training sample. This would be computationally expensive and thus slow to train. Samples are then randomly collected into groups called batches where the error is averaged across each of them and the weights updated once for the group. The batch size is typically small, though larger batches can afford to use a larger learning rate to speed up convergence. Because of the random organization, this method of training is called stochastic gradient descent using mini batches [4].

Backpropagation

$$z_{ij} = \sum_j w_{ij}x_j + b_i \tag{1}$$

Result of node j of layer i.

$$y_{ij} = f(z_{ij}) \tag{2}$$

Activation of node j layer i.

$$\delta_i = \frac{\partial E}{\partial y_i} = \frac{\partial E}{\partial z_i}\frac{\partial z}{\partial y_i} \tag{3}$$

Definition of error term δ of output neuron j.

$$\delta_i = \frac{\partial E}{\partial y_i} = \frac{\partial E}{\partial z_i}\frac{\partial z}{\partial y_i} = (\delta_{(i+1)j} \times w_{ij})\frac{\partial z}{\partial y_i} \tag{4}$$

Definition of error term δ of non output neuron j.

$$\Delta w_{ij} = -L\frac{\partial E}{\partial w_{ij}} = -L\delta_{ij}y_{ij} \tag{5}$$

Weight Update for layer i, weight j, learning rate L.

$$\frac{\partial E}{\partial w_{ij}} = \delta_i y_i \tag{6}$$

Error wrt. the weights at node j layer i.

Mean Squared Error Loss Function

$$E(y, \bar{y}) = \frac{1}{2}(y - \bar{y})^2 \tag{7}$$

MSE Loss Function.

$$\frac{\partial E}{\partial y_i} = -(y - \bar{y}) \tag{8}$$

Output node delta using MSE.

Cross Entropy Loss Function

$$S(y_i) = \frac{e^{y_i} - \mu}{\sum_k e^{y_k} - \mu}, \mu = max(y) \tag{9}$$

Softmax, where μ is for numerical stability.

$$\frac{\partial S_i}{\partial y_j} = \frac{e^{y_i}(\sum_k e^{y_k}) - e^{y_i}e^{y_j}}{(\sum_k e^{y_k})^2} = S_i(1 - S_j), i = j \tag{10}$$

Derivative of Softmax where $i = j$. j = output node num, i = label value.

$$\frac{\partial S_i}{\partial y_j} = \frac{(0)(\sum_k e^{y_k}) - e^{y_i}e^{y_j}}{(\sum_k e^{y_k})^2} = -S_iS_j, i \neq j$$
$$, S_i = \frac{e^{y_i}}{\sum_k e^{y_k}}, S_j = \frac{e^{y_j}}{\sum_k e^{y_k}} \tag{11}$$

Derivative of Softmax where $i \neq j$.

$$E(y, \bar{y}) = -\sum_{i=1}^{n} \bar{y}_i(x)log(y_i(x)) \tag{12}$$

CE Loss function for vectors.

$$E(y, \bar{y}) = -log(\bar{y}(x)), i = \bar{y} \tag{13}$$

CE Loss function simplified for single label classification.

$$\delta_i = \frac{\partial E}{\partial z_i}\frac{\partial z_i}{\partial y_i} = \frac{-1}{S_i}\frac{\partial S_i}{\partial y_i} = \frac{-1}{S_i}\begin{cases} S_i(1 - S_j) & : i = j \\ -S_iS_j & : i \neq j \end{cases}$$
$$= \begin{cases} S_j - 1 & : i = j \\ S_j & : i \neq j \end{cases} \tag{14}$$

CE Loss function simplified for single label classification.

Normalization

$$ZScore(X) = \frac{x - mean(X)}{std(X)}, \forall x \in X \tag{15}$$

Z-Score Normalization Function.

$$MinMax(X) = \frac{x - min(X)}{max(X) - min(X)}, \forall x \in X \tag{16}$$

Min-Max Normalization Function.

Activation Functions

$$identity(x) = x \tag{17}$$

Identity Activation Function.

$$linear'(x) = x \tag{18}$$

Derivative of Linear Activation Function.

$$sigmoid(x) = \frac{1}{1 + exp(-x)} \tag{19}$$

Sigmoid Activation Function.

$$sigmoid'(x) = \frac{1}{1 + exp(-x)}(1 - \frac{1}{1 + exp(-x)}) \tag{20}$$

Derivative of Sigmoid Activation Function.

$$tanh(x) = \frac{1 - exp(-2x)}{1 + exp(-2x)} \tag{21}$$

Hyperbolic Tangent Activation Function.

$$tanh'(x) = 1 - (\frac{1 - exp(-2x)}{1 + exp(-2x)})^2 \tag{22}$$

Derivative of Hyperbolic Tangent Activation Function.

$$relu(x) = \begin{cases} x & : x \geq 0 \\ 0 & : x < 0 \end{cases} \tag{23}$$

ReLU Activation Function.

$$relu'(x) = \begin{cases} 1 & : x \geq 0 \\ 0 & : x < 0 \end{cases} \tag{24}$$

Derivative of ReLU Activation Function.

2.3 Required Functionality

MLDB is meant to supplement an existing database system with the tools needed to apply selected machine learning tasks to its data. This means that the procedures used by MLDB will need read access to the production data. Columns from production tables will need to be designated as features or labels meant for training. The data will need to be copied into a dedicated table for use by MLDB to minimize interference with the existing system. The MLDB table will abstract the source column names by using indices so that source tables with different column structures can be collected into a single table. On this table there will be a field that tells whether the data sample will be used for training. The data values in this table will need to be transformed to numeric values and standardized by a selection of methods before being used for training, which can be either stored persistently or standardized at the time of training.

Once the datasets have been established, the next step is to create neural network models. Each model should have an identifying name and be tied to a single dataset. The number of input nodes should match the number of features of the dataset, while the size of the hidden layer and number of output nodes is determined by the creator. Three activation functions can be chosen for the input, hidden, and output layers. Finally, the loss function that the network will use to update the initial weights can be selected. Weights should be initialized as a small value centered around 0.

Before training, a percentage of samples should be set aside and not used in training to ensure the model does not overfit, or "memorize", the samples it trained on. This would cause artificially high accuracies. Once the value that marks a sample for use in training is set, then the database implementation of the gradient descent algorithm as described in Section 3.2.1 can begin. Parameters set by the application include the number of epochs, or the number of times the entire forward and backward propagation cycles, the sample batch size, and the learning rate. The per batch or per cycle loss should be displayed to indicate that the training is lowering the loss value, and to give an idea of an appropriate number of epochs required.

The success of training can be determined by using the samples that were set aside for testing. The test samples will be used in the model and perform the forward propagation and error calculation phases, but do not update the weights. A low loss and low error calculation indicates a model can perform well when using novel inputs.

2.4 System Standards

Coding Standards

- Database tables should be normalized as to not duplicate data.
- Database stored procedures are used to modify data and not return data.
- Database functions are used to return data and not modify data.
- Naming Conventions
  - All database objects will be named in lower case with words separated by underscores.
  - Table - $<$ *name* $>$
  - Temporary Table - tmp_$<$ *name* $>$
  - Functions
    - Scalar - fn_$<$ *name* $>$
    - Table - fx_$<$ *name* $>$
  - Stored Procedure - sp_$<$ *name* $>$
  - Input Parameter (variable) - v_$<$ *name* $>$
  - Local Variable - lv_$<$ *name* $>$

2.5 Design Constraints

- Speed - The time the system takes to train a model should be comparable to alternative machine learning tools to make this product viable. Training time is a major limiting factor in the cost of generating a working model. Quick creation of models and training times can help produce the best working models.
- Accuracy - The models generated by MLDB should be trainable to attain a prediction accuracy at least comparable to an equivalent machine learning framework's accuracy. This constraint ensures the machine learning algorithms are applied correctly. The output accuracy is highly dependent on the input data and network architecture, but high accuracy should still be achievable in the right circumstances. Without an accurate model, MLDB will be unusable for real world applications.
- Limited interference - The usage of machine learning algorithms should only minimally interfere with the production system, such as having read only access. Data should be read and inserted into a new table for two reasons. First is that training a neural network can take minutes to hours, and relational databases could place locks on the row being read so that no other process can write to the same row, slowing down regular processes. Second is that before the translation the production data can be pre-processed to be better suited as inputs to a neural network.

2.6 Security and Privacy Considerations

Because MLDB is designed to work with an existing database, and current security policies should be applied to the new objects created as well. Access control on the table of sample data must match that of the highest security level production table. Any data duplication for usage by MLDB should not cause a security loophole that could cause an unwanted breach of data. Access to MLDB should be from users without write access to the rest of the database in the event of an exploit being found. This can be done by separating all objects into a new schema and creating users with read and write permission to MLDB and only read to any other schema.

Tight backend security must be in place because the new interface to MLDB introduces another possible attack vector. The interface should be free from common web based attacks such as Cross Site Scripting and SQL Injection, especially considering the number of entry fields the interface will provide to users.

2.7 System Design Diagrams

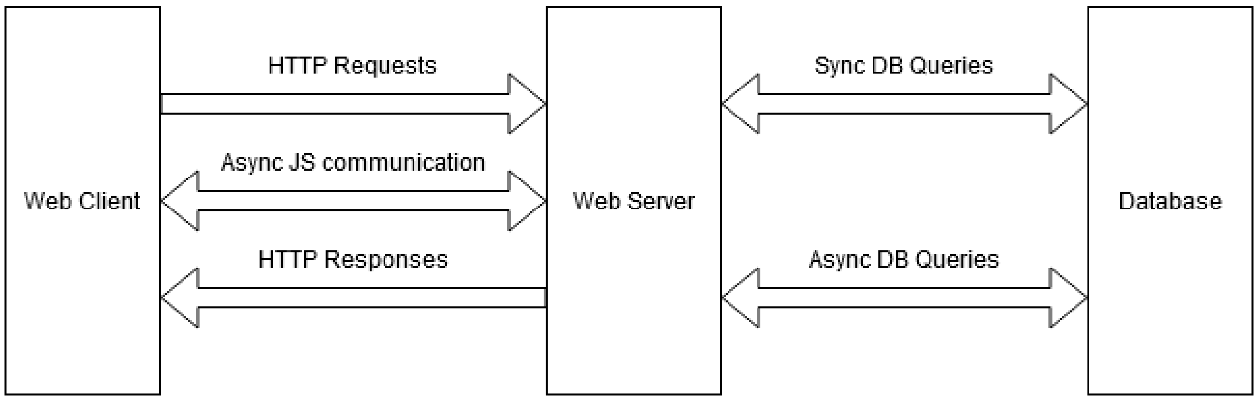


Figure 1: Overview

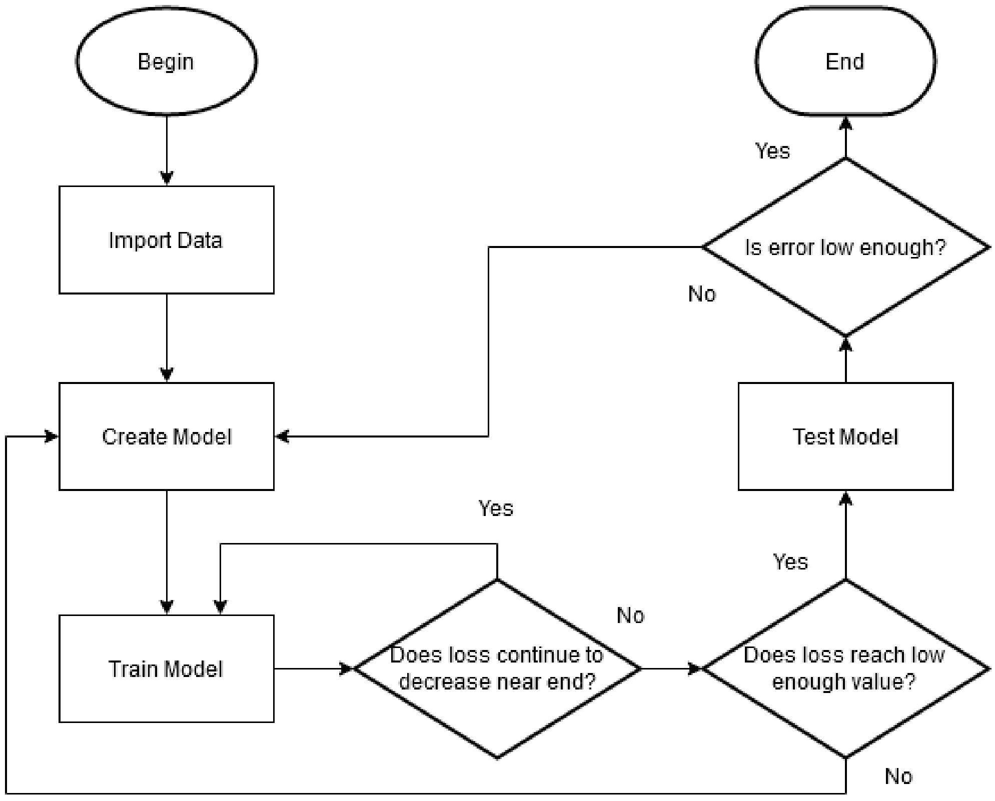
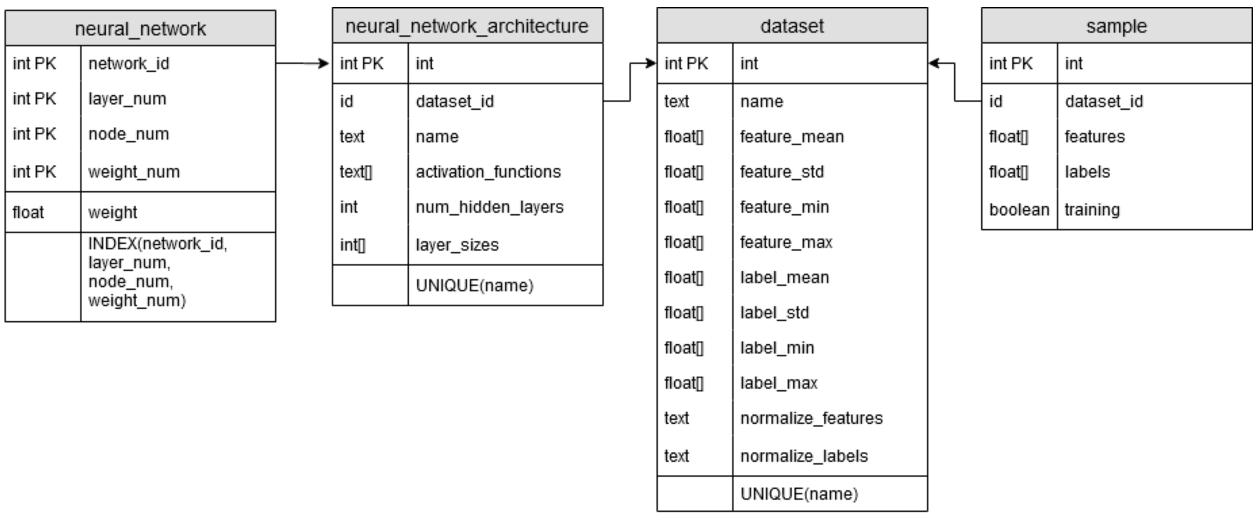


Figure 2: Flowchart



neural_network	
int PK	network_id
int PK	layer_num
int PK	node_num
int PK	weight_num
float	weight
	INDEX(network_id, layer_num, node_num, weight_num)

neural_network_architecture	
int PK	int
id	dataset_id
text	name
text[]	activation_functions
int	num_hidden_layers
int[]	layer_sizes
	UNIQUE(name)

dataset	
int PK	int
text	name
float[]	feature_mean
float[]	feature_std
float[]	feature_min
float[]	feature_max
float[]	label_mean
float[]	label_std
float[]	label_min
float[]	label_max
text	normalize_features
text	normalize_labels
	UNIQUE(name)

sample	
int PK	int
id	dataset_id
float[]	features
float[]	labels
boolean	training

Figure 3: Table Schema

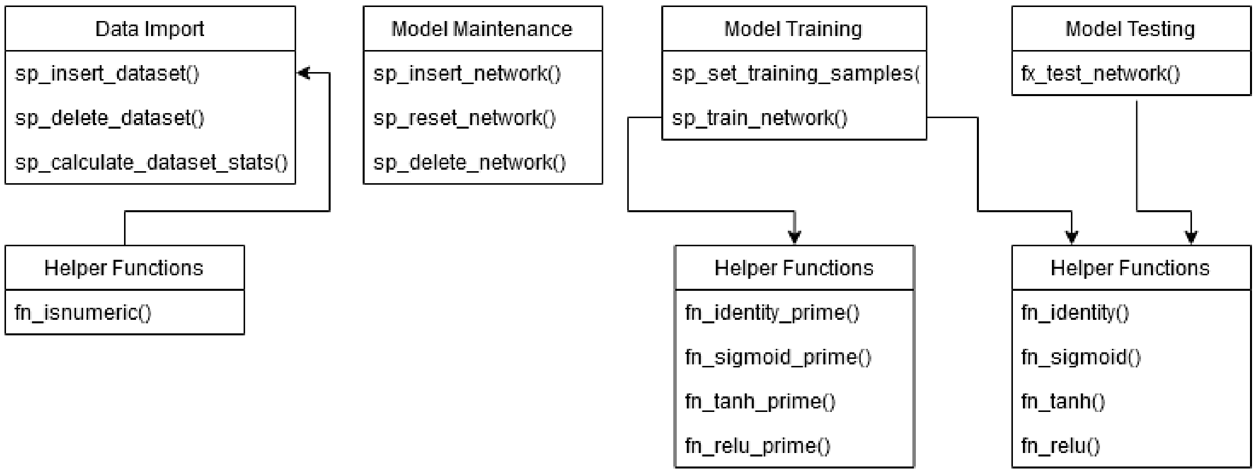


Figure 4: Database Dependency Grouped By Use

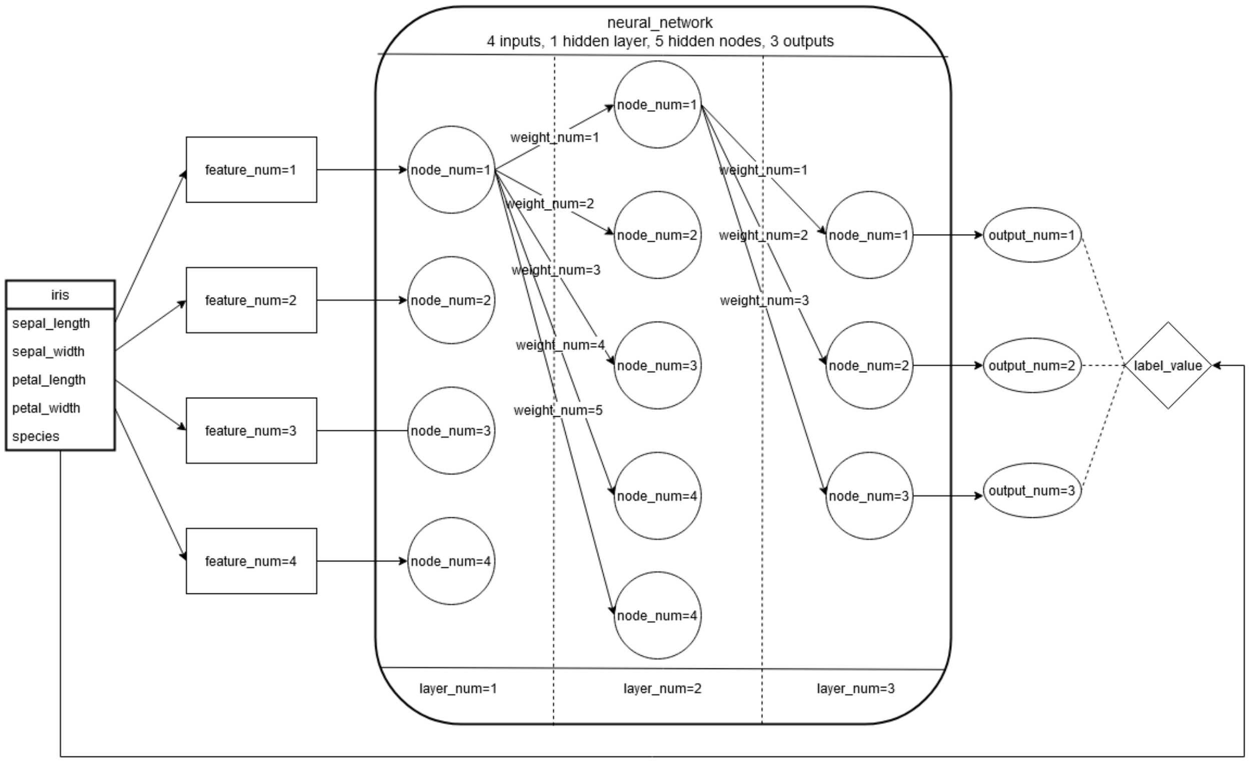


Figure 5: Network Diagram

2.8 Hardware/Software Overview

- Software Modules
  - PostgreSQL Relational Database Server
  - Flask Web App
- Hardware
  - N/A

2.9 Economical, Technical, and Time Constraints

- Economical

  - An economic constraint is the cost of the computer used to host the servers needed for this project. Faster processing is desired to have lower training times, but is not absolutely necessary.
  - There is technically the cost of the energy consumption of the server, but it is negligible.
- Technical
  - SQL was a major limiting factor in how easily machine learning algorithms can be implemented for this project. Some features that could improve the accuracy outcomes were not considered because of the complexity of implementation and efficiency costs.
  - Having a user-facing front end always presents the possibility of security flaws that can be exploited. Basic security principles were considered and implemented, but some important security features may have been missed because of technical difficulty.

3 Implementation Details

3.1 Database Server

3.1.1 System

PostgreSQL was chosen as the database client because of its widespread use and reliability. Postgres is also open source, making it ideal for educational purposes. The version used for this project was 11.6.

3.1.2 Table Structure

Figure 3 shows the schema of tables required to use MLDB. The table "dataset" holds information about information from data derived from the production system. Each column of a float array type holds statistical information for each feature or label that is calculated when standardizing the inputs and used to convert new data into a standardized. Because models are trained using standardized inputs, new inputs should be converted into the same form. The "sample" table holds individual data points with an array of values for features and labels. For an implementation in a database system that does not support array types, the sample table would have to be broken into "sample_features" and "sample_labels", where a row in each is for an individual value.

The table "neural_network_architecture" contains data that describes a model. Each model is keyed to the dataset it is meant to train on, and comes with a unique name for when there are multiple models per dataset. The "activation_functions" column has a value for each layer, including input and output layers, that the training procedure uses to calculate the outputs of each layer. The columns "num_hidden_layers" and "layer_sizes" are for ease of use when using the model, though the values can be calculated directly from "neural_network".

This table holds the weights for the model. Its key is a combination of its architecture, layer, node, and weight number, followed by a dimension of a float type value. Weights could have been collected into an array such as was done with "sample" to save table size on disk, but this would create many layers of nesting that would be inefficient to unroll. This table must also be read and updated constantly during training, and having the "weight" column indexable by the key values would improve lookup times.

3.1.3 Functions and Procedures

Procedure Name	**Description**	**Parameters**
sp_calculate_dataset_stats	calculate statistical values like min, max, mean, and standard dev. for a dataset	dataset_id int
sp_delete_dataset	deletes from dataset and all dependent tables	dataset_id int
sp_delete_network	deletes from the network and neural_network_architecture table	network_id int
sp_insert_dataset	creates a dataset record and imports data from an external table into sample	name text, source_table_name text, feature_column_names text, label_column_names text, normalize_features text, normalize_labels text
sp_insert_network	creates a neural_network_architecture record and weights in neural_network	name text, dataset_id int, activation_functions text[], loss_function text, num_inputs int, num_hidden_layers int, num_hidden_nodes int, num_outputs int
sp_reset_network	resets network weights to random values	network_id int
sp_set_training_samples	assigns training boolean to a random percentage of samples	network_id int, percent_test float
sp_train_network	trains neural network	network_id int, num_epochs int, batch_size int, fearning_rate float

Table 1: Procedures

Function Name	Description	Parameters	Outputs
fn_identity	Identity AF	x float	Equation 17
fn_isnumeric	Tests for numeric value	value text	True if numeric, else False
fn_relu	RELU AF	x float	Equation 23
fn_relu_prime	RELU' AF	x float	Equation 24
fn_sigmoid	SIGMOID AF	x float	Equation 19
fn_sigmoid_prime	SIGMOID' AF	x float	Equation 20
fn_tanh	TANH AF	x float	Equation 21
fn_tanh_prime	TANH' AF	x float	Equation 22
fn_test_network	test network with training or testing samples	network_id int, test_samples_only boolean, show_samples boolean, num_samples int	type int, test_sample_id int, output_num int, label_value float, output_value float, sm_output_value float, loss float, error float

Table 2: Functions

3.1.4 Data Import

The procedure "sp_insert_dataset" handles creating a MLDB dataset and importing data from an external table to store int the "sample" table. It accepts parameters of the dataset's name, the source table's name, a comma delimited list of feature column names, and a list of the label column names. After inserting a new record into the "dataset" table, it uses dynamic SQL to retrieve the data from the source table. The column names from the comma delimited lists are extracted from the table using the hstore and unnest functions from PostgreSQL. The columns are abstracted into "feature_value" and "feature_num" identifiers so that the "sample" table only needs an index to reference the feature. This is especially useful for training, as the code does not need to be modified depending on what inputs are given.

In the case that the mentioned column values from the source table are not numeric, as determined by the "fn_isnumeric" function, then the value of the "rank" window function is used instead of the text value. The "rank" function orders the unique values of that column ascending and assigns a number starting at 1 to each of them. The "normalize_features" and "normalize_labels" parameters indicate what algorithm, if any, should be used to normalize the values. Options are using Z-Score (Equation 15) or Min-Max (Equation 16) techniques. For classification, the label numbers should not be normalized and the value will be set to NULL. The statistical values for the labels and features are calculated with "sp_calculate_dataset_stats" and stored on the "dataset" table so that when a novel input is meant to be used by a model, then it can be normalized just like the samples the model was trained on were.

The procedure "sp_delete_network" will delete all "neural_network_architecture", "neural_network", and "sample" records with a foreign key to the "dataset" record before deleting itself.

3.1.5 Model Maintenance

Neural network models can be created using the "sp_insert_network" procedure. A "neural_network_architecture" record is created with the parameters specified in Table 1. Using the parameters describing the shape of the network, "neural_network" records are created matching all layers, nodes, and weight numbers using the built-in "generate_series" function. Weight values are initialized with the "random" function, which returns a float value from -0.05 to 0.05. The bias values are handled by appending a new node for each layer. This node has connections to all nodes in the next layer, and its value is initialized to 0.0.

If the training parameters did not generate a successful model, then network weights can be reset using "sp_reset_network" to randomly set the values within the range specified in the paragraph above. Networks can be safely deleted with "sp_delete_network".

3.1.6 Model Training

Only a certain percentage of the samples available should be used to train a model. The procedure "sp_set_training_samples" accepts a percentage for the amount of samples in a dataset that should be marked for testing. This value should be around 10 to 15 percent. The procedure will update the "sample" records with a "training" column of type boolean. This is done by ordering the samples by the "random" function and marking the row number. If the row number is less than the total number of samples times the percentage marked for testing, then the "training" value should be false.

If the value is false, then the sample will not be used for training the model. This would affect all models that use a shared dataset, but it was a tradeoff as to not have a boolean value mapped to each sample for each dataset.

"sp_train_network" is responsible for implementing the gradient descent and weight updating algorithm as described in Section 2.2. The parameters are the dataset id, number of epochs, batch size, and the learning rate. A decision was made to calculate the normalized sample values whenever this procedure is called, instead of storing them with a normalized value. This sacrifices startup time to begin training on each call, but allows the original sample values to be stored without repetition. Also, the normalized values could fluctuate based on new samples being added to the table. The statistics and normalization algorithm stored on the "dataset" table are used in this calculation. These values are stored in a temporary table called "tmp_sample_feature_norm" and "tmp_sample_label_norm".

When inserting normalized sample features into "tmp_sample_feature_norm", the "NTILE" function is used to randomly group samples into batches. The function accepts a parameter of the number of batches, which is equal to the number of samples divided by the batch size. Weights will be updated based on the gradient calculated for samples in a single batch at a time. This generally increases training time with more updates and looping, but batch updates provide a balance between updating the weights for each sample against updating the weights once for all samples. Updating on a per sample basis is prone to incorporating noise from a single sample into the weights, while updating for the entire sample space will take longer to calculate the gradients and take longer to converge because the gradient is averaged across all samples [3].

The number of epoch parameters is used to start a for loop surrounding the main block of training code. Immediately under the epoch loop is another for loop across the sample batches. Samples will be selected from the "tmp_sample_feature_norm" table with the loop's batch number. An index was created on the table's "batch_num" column to improve performance because it is frequently used in WHERE clauses.

A "tmp_layer_state" temporary table will store the output from nodes in each layer as a direct result and the result after it has passed through the layer's activation function, called the activity. The table also stores the sample's ID corresponding to the input so that the final network output can be matched to the sample's label. Insertion into the table follows the Equations 1 and 2. A set of 1 valued features are appended to the sample features to match with the bias node. This makes the calculation include only multiplication without worrying about which nodes are bias nodes that need to be added to the final result. Finally, a case statement over the layer's activation function retrieved from the table "neural_network_architecture" is used to call the correct function.

After the first layer results are calculated, then the "tmp_layer_state" table can be used as inputs to the hidden and output layer nodes. Another for loop is used to loop over the rest of the layers. A bitmap index was created on the table's "layer_num" column so that subsequent queries filtering by layer will have a better search time. More 1-valued inputs are appended to the layer inputs. Again, the previous layer's activity values are multiplied with the network's weight for the input node and all nodes in the next layer. Either the activity function defined for a hidden layer or output layer is used. Forward propagation is complete after the output layer results are calculated.

Backward propagation is begun by calculating the delta values for each layer. The delta is dependent on which loss function was selected for the network. Currently implemented functions are Mean Squared Error Loss (Equation 7) and Cross Entropy Loss (Equation 12). Along with calculating the delta values, the real loss is printed to the console with a "RAISE NOTICE" statement so a user can see if the loss is decreasing.

For Cross Entropy Loss, the typical output layer's activation function is softmax (Equation 9). Because the individual node values are dependent on the others, the normal function calls as done when inserting into "tmp_layer_state" would not be sufficient. Softmax is built into the cross entropy loss function calculation.

Therefore, the output layer's activation function should be set to Identity.

Finally, the gradients can be calculated for the batch. The layer state table is unioned with the sample features, as they make up the first layer's inputs and affect the rate of change of the weights. Then all node layers are augmented again with 1-valued features that correspond to the biases. The gradient is defined in Equation 6 as the delta of the previous layer times the activity of the current layer. The weights are updated with Equation 5, which decreases the gradient by a small learning rate and a high number of samples.

3.1.7 Model Testing

Models are testing by using the samples from the dataset that were not used in training. The table valued function "fx_test_network" can optionally include training samples as well by using a parameter. The desired outputs for testing are the loss and accuracy that is averaged across the selection, but both can be shown for each sample. Individual samples can help get a realization of what actual values the model generated, while the whole gives a generalized metric that allows models to be compared.

Generating the loss and accuracy requires the forward propagation phase to be recreated. Samples must be normalized to match the samples the model was trained on. The temporary table "tmp_sample_feature_norm" is required again to store intermediate layer results using the same equations as before.

The outputs must be denormalized using the dataset's statistics if the labels are normalized in the first place. This makes it easier to compare to the actual label values when the individual sample results are printed.

The error is calculated by the equation $(actual - observed)/actual$. The accuracy is 1 - error. For MSE, the activity and label_value are compared, and for CSE, the amount of times the correct output number matches the label value. Both the loss and accuracy is averaged for all samples and unioned with the rest of the individual samples.

3.2 Web Application

3.2.1 Web Server

A Flask web app was created as an interface to easily manipulate the database as a part of MLDB. Flask is an easy to develop web application framework that can be used with Python for the server side code. Flask allows you to specify URL locations on the web server that outputs data based on the return value of a Python function. The response can be generated in Python or served from an HTML template. The majority of the web code for MLDB with Flask is contained in the "flask_app.py" file.

All of the functionality required for MLDB were fit into a single page indexed under "/mldb". Upon a GET request at that address, the server will begin collecting data from the database to display. This includes a list of datasets and networks. Other data such as activation and loss functions are coded into the server, but could be retrieved from the database if stored there. Finally, an HTML template for the page as "mldb_home.html" is served with the data that will be inserted to the page with Jinja.

3.2.2 Database Adapter

Communication with the database is done with psycopg2, a popular database adapter for Python. Code for this is written in the file "db_controller.py", and is mostly interfaced through the function "db_execute", which accepts a formatted query string and the parameters to be added to it. Psycopg2 can safely execute SQL commands with user defined values with a format similar to prepared statements, where the query string is prepared with the "%s" placeholders, independent of the actual value's type. The adapter's cursor "execute" function accepts the query string and the values list separately. Any rows retrieved from a SELECT statement or function call are returned to the calling function as a list.

Another important feature of psycopg2 is that it can execute database queries asynchronously. That means for long query executions it does not lock up server resources and the adapter can poll for data continuously returned from the query. This is used for training networks so that the “RAISE NOTICE” messages can be retrieved from the server every few milliseconds as opposed to being available only at the end of execution. Only a single thread is allowed to execute at once, and multiple attempts to execute an asynchronous query will also poll the connection for an OK status in the function “db_wait”. Once a thread is allowed to begin execution, then the “db_wait_close” is called that will close the connection once complete.

3.2.3 Communication with the Server

When a notice is retrieved from the database through asynchronous polling, the data must be sent to the webpage without causing the page to render again. This is accomplished with Socket IO, a framework that allows for real time communication between client and server that is available for Flask. Networking is supported by gevent. When a client opens the webpage, a connection is established on the “mldbns” namespace that will issue keep alive messages for the duration of the page being open. Both server and client can emit messages that will be sent to the other. Therefore, whenever a poll does retrieve a notice message, it can immediately be sent to the client.

This communication channel is also used to issue commands from the client to server, commonly in the form of button presses. These commands do not require HTTP requests and preserve current web page state. The commands are sent with a “type” for what event has occurred such as a request to refresh the list of networks or create a new network. Data can also be sent to and from the server in the form of key-value pairs.

3.2.4 Webpage

The “mldb_home.html” template file contains all the markup required for MLDB. It is designed to clearly identify what operation should be done with the elements on the page. Tabbed elements are used to only show sections of the page at a time. The tab functionality is implemented with simple javascript functions to hide the other containers.

While the dynamic data on the page, such as the network and dataset tables, is initially constructed with Jinja, updates are performed with the socket communication as described in Section 4.2.3. On the page load a socket connection is established with the server on the “mldbns” namespace. Any received messages are captured with a “socket.on()” function with the first parameter being the event’s name. These functions use JQuery to dynamically create HTML elements from the message’s data.

Button events are intercepted with JQuery so that socket messages can be emitted to the server. Data from the page elements such as table row selections and text box values are added to a key-value collection that is sent with the message for processing. The server can respond on its own time with the data requested or for real time messages in the case of asynchronous database communication.

4 Experimental Procedure and Anaysis

4.1 Procedure

The purpose of this experimental procedure is to validate the constraints as detailed in Section 2.5. Datasets will be imported from tables and data manually created on the database to simulate the production environment. The datasets will come from popular machine learning sources including Iris, Wine Cultivar, and Airfoil, all retrieved from the UCI Machine Learning Database [5].

The datasets were downloaded to a comma or tab delimited file. Appropriately named tables were created with either text or float type columns. The data was imported using the “ psql console command. On the MLDB site, the Datasource tab was used to transfer the table data to the “sample” table by listing the feature and label columns and giving the local dataset a name. From here, testing was performed by creating models of varying sizes against the datasets and recording the runtime and accuracies during and after training.

4.2 Results

4.2.1 Iris Dataset

The Iris dataset is a collection of 150 samples of measurements of 3 types of iris flowers. The measurements are of the sepal length, sepal width, petal length, and petal width. Three networks were created to use this dataset, all with 4 inputs and 3 outputs and use Cross Entropy Loss. The hidden layer sizes were one layer with 8 nodes (1x8), two layers with 6 nodes (2x6), and two layers with 12 nodes (2x12). The input and hidden layers used the tanh activation function while the output was set to Linear because softmax will be used on the values. The batch size was set to 25 samples.

Training was performed 5 separate times for each network and the results averaged together. Accuracy and test loss metrics were unavailable during training, and are only shown for the network before training was begun and immediately after. 15% of the total samples were set aside for testing.

To get a baseline of a tested machine learning framework, each model was duplicated in Keras to compare results. To ensure the architecture was the same, the number of rows in "neural_network" for the network tested was checked to match the Keras "summary()" function's number of trainable parameters value. The initial values of the model had the same uniform distribution from -0.05 to 0.05 for weights and 0.0 for biases.

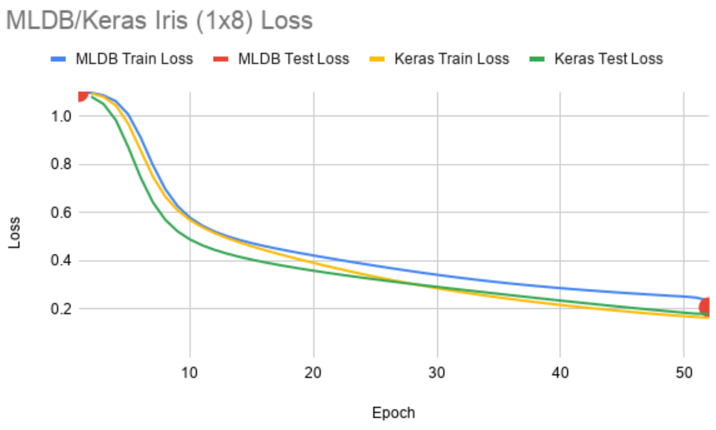


Figure 6: Iris (1x8) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	83.18	20.47	82.47	4.34
25	97.12	95.91	97.26	93.91

Table 3: Iris (1x8) Accuracy

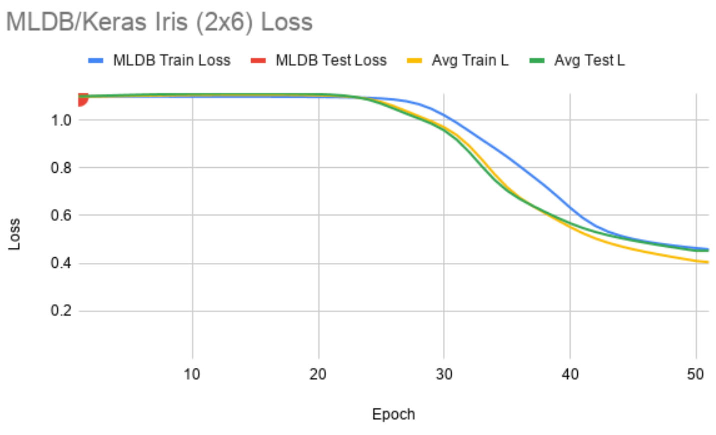


Figure 7: Iris (2x6) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	33.59	26.77	30.00	21.74
50	80.16	85.67	89.09	83.47

Table 4: Iris (2x6) Accuracy

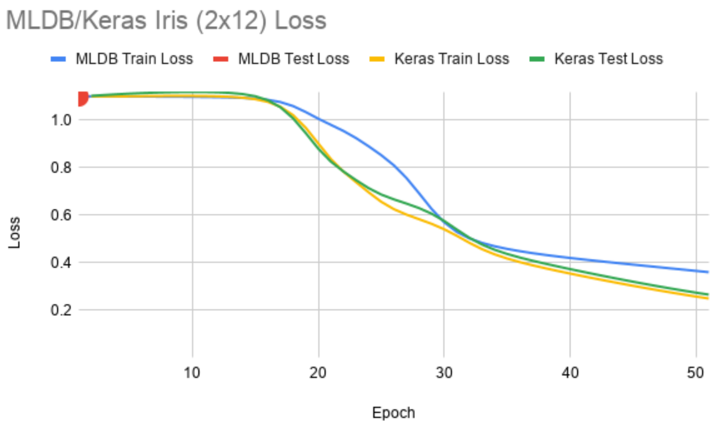


Figure 8: Iris (2x12) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	22.34	8.66	22.73	8.69
25	88.91	95.28	99.09	93.91

Table 5: Iris (2x12) Accuracy

The results of the Iris test show that simple network architectures can successfully make predictions after a short training session with MLDB, and that the results are comparable to the Keras model equivalent. The training loss decreased as the epochs progressed, and the test accuracy from before and after training jumped from a 97% for the (1x8) network and 99% for (2x12). It is unusual that the test accuracy tended to be larger than the training accuracy.

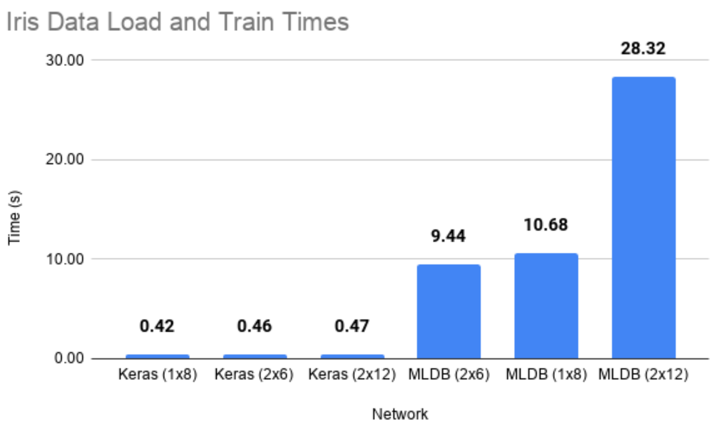


Figure 9: Iris Timing

The time to prepare the data and train the network was much higher for MLDB than Keras for all models. While a training time of nearly 30 seconds is not a lot, Iris is one of the smallest datasets of practical use in benchmarking. Also, the models are tiny with a few hundred weights at most.

Another issue with timing is that the time between batches linearly increases for successive epochs. This was attempted to be mitigated by clearing the "tmp_sample_features" and "tmp_delta" with the PostgreSQL unique "TRUNCATE" command instead of "DELETE" after each batch, but the issue still persisted. A test with a 1x8 network was done for 150 epochs and the time between epochs was recorded as "since last".

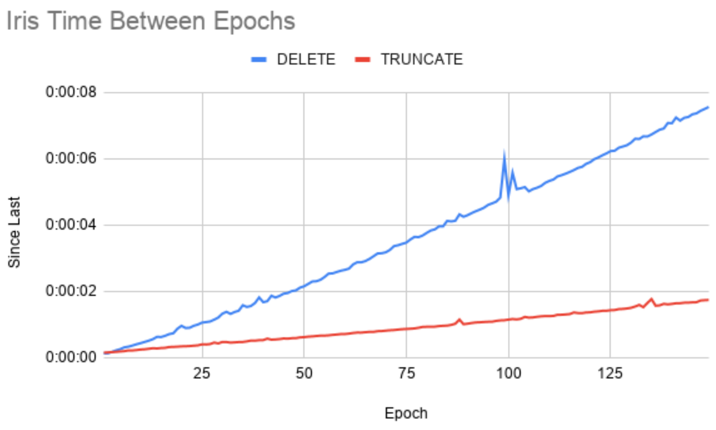


Figure 10: Iris Time Between Epochs

4.2.2 Wine Dataset

This dataset is similar to Iris in that it is a classification problem with only 3 classes, but has 13 input features related to the chemical properties of the wine from 3 different cultivars. The testing procedure was the same as with Iris.

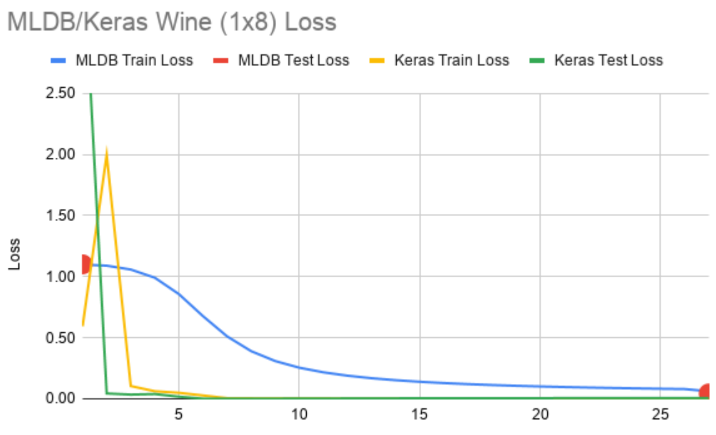


Figure 11: Wine (1x8) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	0.28	0.71	0.16	0.37
25	0.99	1.00	1.00	1.00

Table 6: Wine (1x8) Accuracy

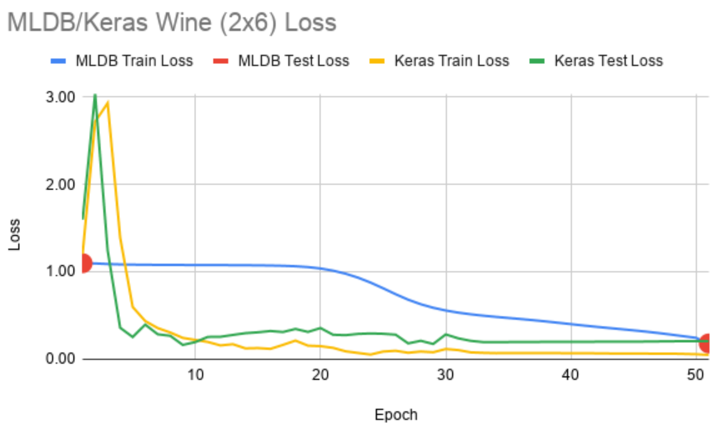


Figure 12: Wine (2x6) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	0.36	0.33	0.32	0.36
50	0.96	0.97	0.95	0.95

Table 7: Wine (2x6) Accuracy

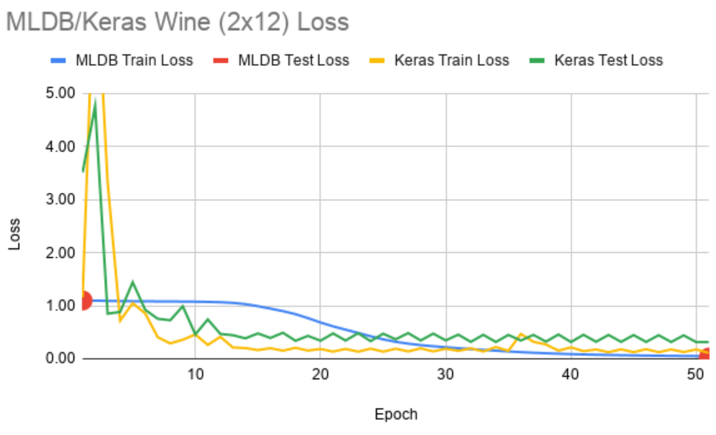


Figure 13: Wine (2x12) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	0.20	0.43	0.12	0.47
50	1.00	0.97	1.00	0.94

Table 8: Wine (2x12) Accuracy

The Wine dataset with MLDB was even more successful than Iris in predicting the three classes, even with more features. The MLDB training loss decreased slower than in Keras, but still ended up resulting in near 100% accuracy. Of course, the dataset was small and not too difficult to get a model to train with, but it shows the machine learning foundation of MLDB is sound.

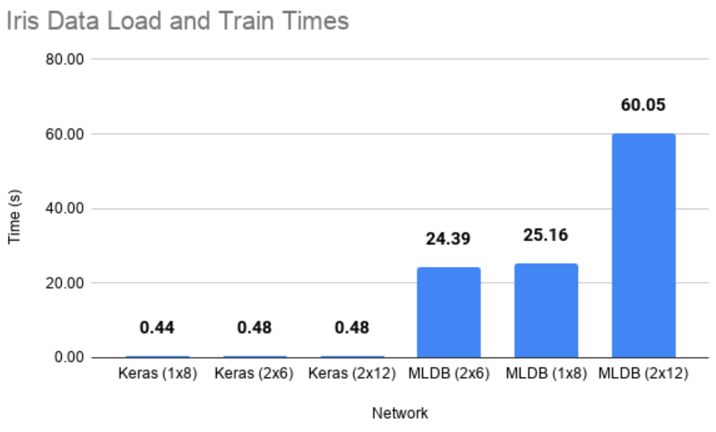


Figure 14: Wine Timing

Still, MLDB cannot compare with Keras in terms of computational time.

4.2.3 Airfoil Dataset

The airfoil data set of 1503 measures of the noise in decibels that results from an airfoil in a wind tunnel for different configurations. The measurements are the frequency (Hz), angle of attack (degrees), chord length (m), free stream velocity (m/s), and the suction side displacement thickness (m). Again, three networks were created and the results averaged across 5 training sessions with 15% of the samples used for testing. The Mean Squared Error loss function will be used, as this is a regression problem.

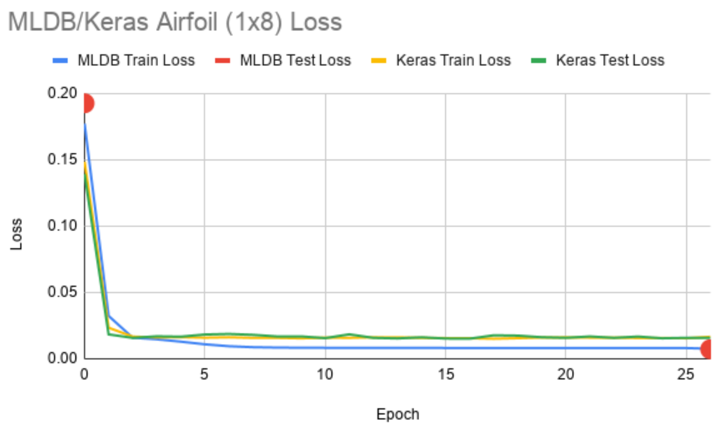


Figure 15: Airfoil (1x8) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	83.18	81.94	82.47	82.20
25	97.12	95.56	97.26	95.72

Table 9: Airfoil (1x8) Accuracy

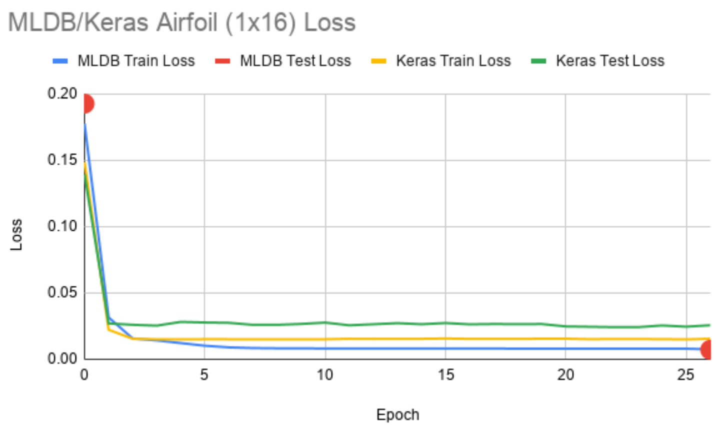


Figure 16: Airfoil (1x16) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	83.18	86.50	82.48	82.20
25	97.12	95.71	97.25	94.72

Table 10: Airfoil (1x16) Accuracy

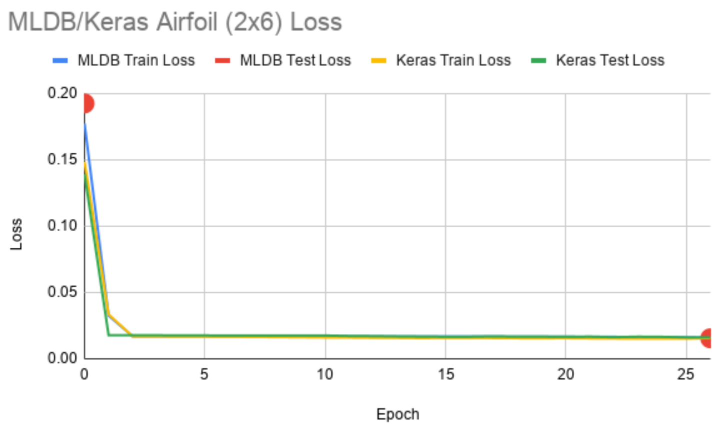


Figure 17: Airfoil (2x6) Loss

Epoch	MLDB Train Acc.	Keras Train Acc.	MLDB Test Acc.	Keras Test Acc.
0	83.18	82.47	85.52	85.93
25	95.53	95.61	95.67	95.60

Table 11: Airfoil (2x6) Accuracy

MLDB also seems to perform well for regression training with the Airfoil dataset. The accuracy increases from values in the low 80% range to around 95%. Upon inspecting the returned values, they tend to remain in a certain range as seen in the table below.

Label Value	Output Value	Loss	Accuracy
114.57	124.95	0.0381	90.94
114.75	116.80	0.0015	98.21
115.46	115.27	0.0000	99.84
119.81	124.44	0.0076	96.14
120.06	125.72	0.0113	95.28
120.84	116.37	0.0071	96.30
123.45	121.26	0.0017	98.22
126.20	122.86	0.0039	97.36
133.06	124.17	0.0280	93.31

Table 12: Airfoil Sample Test

The output values for the (2x6) network returns mostly values around 124. The output values from the network are closer to the real label for small network sizes, especially with just one hidden layer. Inspecting the data shows that the labels are heavily imbalanced towards this region. The model trains well as shown by the decreasing error function, but examining the real output values shows that the model just outputs the mean label value instead of learning how to map the features to the labels. This is an issue with the dataset, and not the model.

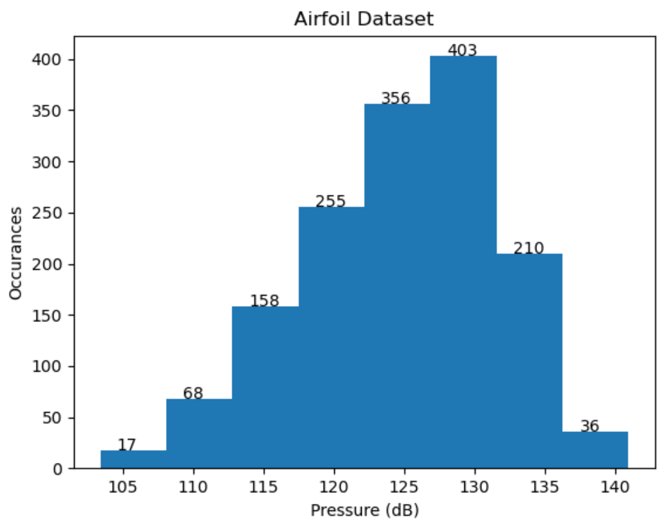


Figure 18: Airfoil Label Histogram

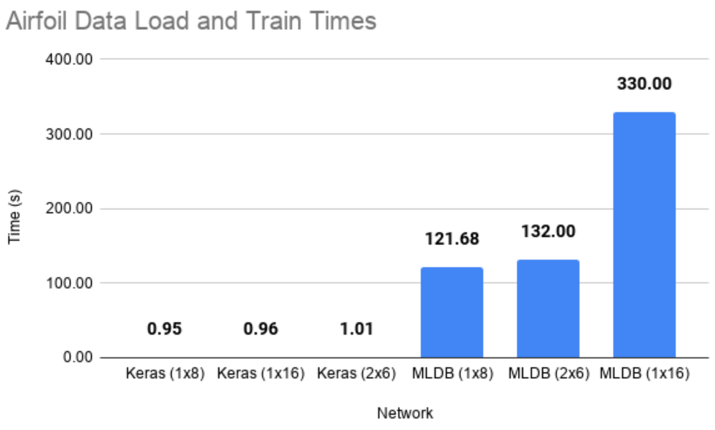


Figure 19: Airfoil Timing

Like with Iris, the training times for MLDB with the Airfoil dataset are atritious with comparable networks and results.

5 Societal Impact

All references used in the creation of the product or writing of this report are listed in Section 8. This project was written using all open source or freely available software. The source code for the project is listed in the GitHub link in Section 9.

6 Societal Contribution

MLDB was created in an attempt to expand the applications of machine learning. Many frameworks were used together to create a product that can reliably generate well trained models, at least for small datasets. If an organization cannot use other machine learning frameworks to improve their operations, MLDB could theoretically aid in predictive analysis or other useful functions.

Because of the the lack of fine control and optimization of MLDB, more complex data would not be usable by this product. The limitations of the relational database architecture cause a significant increase in computation time compared to traditional resources. The implementation is notable in its relative novelty, though another paper cited in Section 7 already outlines the drawbacks. MLDB comes from an idea to combine the data storage source and means of modelling, though the real outcomes are probably not too useful for practical purposes in society.

7 Conclusion

MLDB was created to have an easy to use machine learning tool that did not require large external frameworks. The algorithms required for deep learning were written in SQL to perform regression or classification with the data in the database. Combining the source of data and the logic required to train models have potential benefits to reduce complexity and dependencies in a software system.

The results shown in Section 4.2 show that the models generated and trained by MLDB are comparable in terms of accuracy with the popular machine learning library Keras. This proves that the implementation of the required algorithms was successful. The datasets that were used in testing all had a small number of samples and features. The relationship between the features and labels was simple enough that only small models were necessary to produce good results.

The most notable drawback from MLDB is the time required to train models, with results showing that it can take over 100 times longer than by using Keras. This issue was highlighted in [1] that relational databases do not support recursion well that is required for the many looping structures. The opmtimizer struggles to materialize the data that needs to be passed from one layer to the next. Features were added like indices and PostgreSQL specific commands to bring down the timing, but the results were still far too slow for MLDB to be a feasible machine learning tool.

8 References

8.1 Citations

[1] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, Z. Gao, Declarative Recursive Computation on an RDBMS [https://doi.org/10.14778/3317315.3317323]. VLDB Endowment, 2019.

[2] J. Zaruda, Introduction to Artificial Neural Systems. St. Paul, MN: West Publishing Company, 1992.

[3] I. Vasilev, D. Slater, G. Spacagna, P. Roelants, V. Zocca, Python Deep Learning, 2nd ed. Birmingham, Mumbai: Packt Publishing, 2019.

[4] E. Charniak, Introduction to Deep Learning. Cambridge, Massachusetts: The Massachusetts Institute of Technology, 2018.

[5] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

8.2 Resources

`https://www.postgresql.org/`

`https://flask.palletsprojects.com/en/1.1.x/`

`https://flask-socketio.readthedocs.io/en/latest/`

`https://www.psycopg.org/`

8.3 Datasets

https://archive.ics.uci.edu/ml/datasets/Iris

https://archive.ics.uci.edu/ml/datasets/Wine

https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

9 Source Code

https://github.com/JKNags/MLDB