Implementation Roadmap for Neural Networks in Array Databases

Otoniel José Campos Escobar
Computer Science & Electrical Engineering
Jacobs University Bremen
Bremen, Germany
o.camposescobar@jacobs-university.de

Peter Baumann
Computer Science & Electrical Engineering
Jacobs University Bremen
Bremen, Germany
p.baumann@jacobs-university.de

***Abstract*—Relational databases lack behind when handling array data and thus array databases were created to fill this gap. Array databases provide optimized storage, retrieval, and processing of multidimensional discrete data (MDD), also known as array data. Just like relational array databases, data processing in array databases is handled declaratively through an array query language that offers enough expressible power to define a myriad of operations. However, despite the advancements in array database technology, there is still a gap in describing machine learning (ML) algorithms and in particular neural networks which, in recent years, have been adopted for predicting phenomena in science and engineering.**

In this contribution, we outline an implementation roadmap for defining neural networks in an array database. We first identify the necessary linear algebra operators present in a feed-forward neural network and use them to define the training and prediction operations of that network. We also define other operators that, though they are not part of linear algebra, are essential for a complete machine-learning implementation.

Keywords—array databases, machine learning, linear algebra, array query language, datacubes

I. Introduction

Machine Learning (ML) is increasingly being utilized in many different applications in both industry and research. ML estimates results by training predictive models using datasets that can be several orders of magnitude larger than the main memory. Historically, these datasets tend to either sit in the file system or in a database for later retrieval by a third-party ML software for modelling and training. In this regard, databases are mostly used as mere data repositories. This approach overlooks key beneficial features for ML model processing that are provided by database systems, such as the declarative and expressive power of a query language; and the powerful data retrieval technologies that have been perfectioned over the years since their inception in the 1960s.

A considerable number of datasets that are currently used for ML are composed of multidimensional discrete data (MDD), also known as array data or datacubes [1][2]. These datacubes, which are often considered the "bread and butter" of scientific computing [3][4], frequently represent spatio-temporal sensors, images, simulations, and statistical data generated by a plethora of applications in science, engineering, and beyond. Examples of datacubes comprise 2-D satellite imagery, 3-D x/y/t image time series, x/y/z geophysical voxel data, and 4-D x/y/z/t climate data from the geoscience field; microarray, confocal microscopy, and human brain data from life sciences [5]. The amount of data produced and can reside in an Array Database is huge, in the order of Petabytes (PB), with Terabytes of new data being produced daily. This phenomenon can be seen in the growth of NASA's Earth Observing System Data and Information System (EOSDIS) data archive from 15 Petabytes (PB) of data in 2015 to more than 59 PB in 2022, with annual data ingestion rates only expected to increase by an order of magnitude by the end of 2021 [6].

Relational databases generally do not perform when handling array data, to fill the gap array databases were created [7]. Array Databases close a gap in the database world by providing modelling, storage, and processing support on datacubes. Currently, most ML applications do not use Array Databases, which results in these applications not using the latter's full array processing capabilities during model training and prediction. Among the benefits of using array databases for ML processing we have:

- Avoid massive data copying to external systems.
- Array database inherent efficient and scalable array data processing techniques.
- Standardized declarative query language for trimming and slicing multidimensional arrays.

Currently, most ML applications do not use Array Databases. At its heart, ML algorithms are composed of linear algebra. By defining the linear algebra operators in an array database query language, users can leverage the inherent data processing features of the array database with the inference capabilities of machine learning algorithms.

In this contribution, we propose an implementation roadmap of the necessary extensions that will be needed to define a neural network in the array database rasdaman [7]. Neural networks and especially deep neural networks are one of the most, if not the most, used ML algorithm for modelling non-linear decision boundaries. Its flexibility has made it ideal for many applications in science and engineering which also crossroads with array databases, e.g., Visual Query Answering for Remote Sensor Data (RSVQA) [8] and High Latitude Dust Detection [9]. To

This work is funded by the German Federal Ministry for Economic Affairs and Climate Action as project AI-Cube under contract 50 EE 2012.

build insight into the reader, we first define a feed-forward neural network in mathematical terms while highlighting the linear algebra components. This paper is organized as follows: Section II. presents the related work that has been done towards implementing ML in array databases. Section III describes the rasdaman array query language. Section IV presents the mathematical definition of a feed-forward neural network. Section V describes the necessary language extensions to the array query that implement a feed-forward neural network. Finally, sections VI and VII present the conclusions and future work respectively.

II. Related Work

There have already been some attempts towards integrating ML algorithms in array databases by incorporating user data types (UDT) and user-defined functions (UDF). Systems like MADlib [10] and Bismark [11] define a new UDT called *vector* that stores matrix rows and columns. Fig. 1 illustrates the *vector* UDT for matrices A and B. Once the UDT is defined, a dotproduct(v1, v2) UDF is defined which calculates the dot product of two matrices using the *vector* UDTs *v1* and *v2*. Fig. 2 defines the query that will output the dot product of two tables.

i	j
1	$[a_{11},..., a_{1l}]$
...	...
m	$[a_{m1},..., a_{ml}]$

i	j
1	$[b_{11},..., b_{1l}]$
...	...
n	$[b_{n1},..., b_{ln}]$

Fig. 1. Matrix storage in tables using vector representation

```
SELECT A.i, B.j, dotproduct(A.row * B.col)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i = B.j;
```

Fig. 2. Query calling the dotproduct user defined function

Other systems enable users to employ their preferred ML framework, in this case Python, to construct a model with all pre-processing steps and library dependencies. In this system, users are also able to query an inferred value from that model. The model and the query undergo static analysis to produce an intermediate representation (IR) that later results in an optimized query plan that is finally translated into an optimized SQL using a runtime code generator that finally runs into a runtime engine like ONNX. Its syntax is a melange of python with SQL query syntax. The RAVEN system [12] employs this type of approach.

In the case of full-stack array databases, rasdaman permits encapsulation of all linear algebra and ML-specific code into a portable and callable UDF that can be executed as part of an array query. This provides a way to extend the array database functionality to support ML but can also decrease the efficiency of ML algorithms because different database optimization strategies cannot always be applied. From the users' perspective, they only need to define the appropriate parameters and the database will output a result; this is a convenient and relatively easy implementation approach for ML. However, other than the ease of implementation and avoidance of data copying, there is no actual use of the underlying array database optimization techniques. In this contribution, we argue that the utilization of these techniques should not be overlooked because they can constitute a breakthrough in array data processing. The next section presents rasdaman's array query language and highlights its key language structures that can support ML algorithm definitions.

III. Array Query Language

In this section, we describe how rasdaman's array query language rasql, provides the necessary expressive power to describe arrays of arbitrary size and dimension. In 2019, the rasql query language has been adopted into ISO SQL – modulo syntax adaptations – into ISO SQL [13].

A. rasdaman query language

rasdaman's array query language [14], rasql, provides retrieval, filtering, and processing of MDD operators. The expressive power of rasql allows to state operations up to the complexity of the Discrete Fourier Transformation. Its cornerstone are two operators: the array constructor MARRAY and the array condenser MDCONDENSE.

The MARRAY constructor takes an n-D array extent and an expression and builds an array whose cells are filled by evaluating the expression for each array position. For example, Fig. 3 creates a 100x100 matrix filled with the pairwise difference of cells taken from existing arrays a and b. This can also be abbreviated as a-b.

```
MARRAY p in [ 0:99, 0:99]
VALUES a[p]-b[p]
```

Fig. 3. MARRAY pairwise difference example

Any general index computation is possible, though, such as determining changes in an x/y/t time-series tx. Fig. 4 exemplifies how this can be defined.

```
MARRAY   x in [ 0:99 ],
         y in [ 0:99 ],
         t in [ 0:99 ]
VALUES   ts[x,y,t] - ts[x,y,t-1]
```

Fig. 4. MARRAY example to determine changes in x/y/t

The condenser MDCONDENSE is somewhat dual in that it iterates over some array area and aggregates based on some aggregation function which is one of the usual suspects count, sum, avg, min, max, some, and all. Fig. 5 shows an expression which determines the maximum value of a MDD array.

```
MDCONDENSE max
OVER        p in sdom(a)
USING       a[p]
```

Fig. 5. MDCONDENSE maximum value example

Again, there is a shorthand for this simple case, written as mdmax(a). And as before, general expressions and addressing

schemes are possible. Altogether, a typical array SQL query looks like below. Fig. 6 contains an attribute data constituting an array with many satellite image spectral bands, including red and nir. From this, the difference of two bands is computed for every tuple, restricted to the x/y/t coordinates indicated in brackets. The result gets encoded in NetCDF, so the query response overall is a (possibly empty) set of NetCDF files.

```
SELECT encode( ls.data.red - ls.data.nir)
                    [ x0:x1, y0:y1, t0:t1 ],
                    "application/netcdf" )
FROM LandsatImageTimeseries as ls
```

Fig. 6. Array with many satellite image spectral bands example

This language allows expressing operations on vectors, matrices, and tensors up to the complexity of the Discrete Fourier Transform. What cannot be expressed are algorithms that are inherently iterative, such as matrix inversion. Adding iterative power to the language, ultimately enabling complete Linear Algebra, while retaining termination guarantees is an area of active research.

IV. Mathematical Definition of a Feed-Forward Neural Network

To build intuition into the necessary linear algebra operators that are needed for defining a neural network, the following section covers the mathematical definition of a simplified feed-forward neural network. We proceed to describe its inputs and outputs along with the necessary linear algebra operations needed to implement a neural network.

Neural networks are robust and widely used ML algorithms used for both regression and classification problems. Although firstly conceived to build machines that mimic the human brain, it has been widely adopted in many everyday applications like image recognition, sales forecasting, and text classification.

In its essence, a neural network is a group of connected neurons ordered in layers. Fig. 7 depicts an example of a neural network that consists of 4 layers. Layer 1 (input layer) contains the features of the training sample $\{x_1, x_2, x_3\}$. Layer 2 and Layer 3 are hidden layers that contain the activation nodes; each activation unit ***i*** in layer ***j*** can be identified as a_j^i. Finally, Layer 4 is the output layer, or the result of the hypothesis where $h_\theta(x) = \begin{bmatrix} a_1^{(4)} & a_2^{(4)} & a_3^{(4)} \end{bmatrix}$ and $K = 3$, where K is the number of labels. Any output layer with $K > 2$ is considered a multi-class classification neural network. The bias units x_o, $a_0^{(2)}$, and $a_0^{(3)}$; shift the activation function by adding a constant, usually 1, to the input. It is analogous to the role of a constant in a linear function.

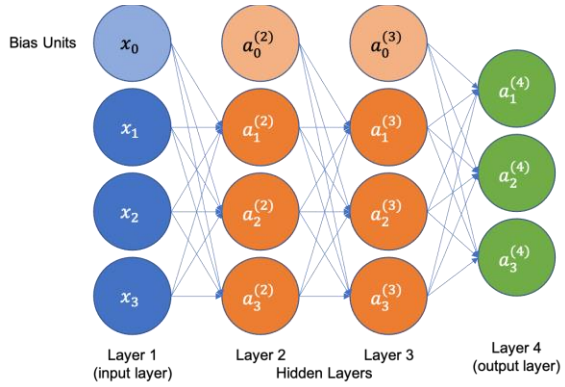

Fig. 7. Feed-forward neural network

For each layer j there exists a weight matrix $\theta^{(j)}$ that contains the weight mapping from layer j to layer $j + 1$. Figure below shows the weight matrix for the first layer $\theta^{(1)}$. Equation (1) shows the weight matrix for the first layer $\theta^{(1)}$. As a convention in this paper, let $\theta_{xy}^{(j)}$ represent the weight from unit y in a preceding layer j to unit x in subsequent layer.

$$\theta^{(1)} = \begin{bmatrix} a_{10}^{(1)} & a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{20}^{(1)} & a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} \\ a_{30}^{(1)} & a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} \quad (1)$$

To calculate the resulting hypothesis, we must perform a forward propagation to compute the unit's activation. Equation (2) defines the activation unit's computation for any layer j for the 4-layer neural network. Where g represents the activation function, which in this example is sigmoid.

$$\begin{aligned} a_1^{(j+1)} &= g(\theta_{10}^{(j)} a_0^{(j)} + \theta_{11}^{(j)} a_1^{(j)} + \theta_{12}^{(j)} a_2^{(j)} + \theta_{13}^{(j)} a_3^{(j)}) \\ a_2^{(j+1)} &= g(\theta_{20}^{(j)} a_0^{(j)} + \theta_{21}^{(j)} a_1^{(j)} + \theta_{22}^{(j)} a_2^{(j)} + \theta_{23}^{(j)} a_3^{(j)}) \\ a_3^{(j+1)} &= g(\theta_{30}^{(j)} a_0^{(j)} + \theta_{31}^{(j)} a_1^{(j)} + \theta_{32}^{(j)} a_2^{(j)} + \theta_{33}^{(j)} a_3^{(j)}) \end{aligned} \quad (2)$$

Finally, equation (3) expresses the hypothesis function h_θ. Here j denotes the last hidden layer of the neural network.

$$h_\theta(x) = \begin{bmatrix} g(\theta_{10}^{(j)} a_0^{(j)} + \theta_{11}^{(j)} a_1^{(j)} + \theta_{12}^{(j)} a_2^{(j)} + \theta_{13}^{(j)} a_3^{(j)}) \\ g(\theta_{20}^{(j)} a_0^{(j)} + \theta_{21}^{(j)} a_1^{(j)} + \theta_{22}^{(j)} a_2^{(j)} + \theta_{23}^{(j)} a_3^{(j)}) \\ g(\theta_{30}^{(j)} a_0^{(j)} + \theta_{31}^{(j)} a_1^{(j)} + \theta_{32}^{(j)} a_2^{(j)} + \theta_{33}^{(j)} a_3^{(j)}) \end{bmatrix} \quad (3)$$

The cost function $J(\theta)$for feed-forward neural networks is depicted in Equation (4). Where $y_k^{(i)}$ and $(h_\theta(x^{(i)}))_k$ are, respectively, the k^{th} elements of the expected output and the hypothesis output for input example $x^{(i)}$.

$$J(\theta) = \frac{-1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(h_\theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - \log(h_\theta(x^{(i)}))_k) \quad (4)$$

Once the cost is calculated, the next step is to calculate the gradient descent, for that it is necessary to use the cost function, its partial derivative, and execute the backpropagation algorithm. This algorithm computes the error for each input

sample, using the cost function, and subsequently calculates the partial derivative by backpropagating the errors from the output layer to the input layer. To exemplify this process, we will use a 4-layer neural network example from before and use its vectorized implementation for simplification.

For each training sample $(\boldsymbol{x}, \boldsymbol{y})$, we compute the activations of the units with forward propagation. Equations (5) – (10) define the sequence of operations in forward propagation.

$$a^{(1)} = x \tag{5}$$

$$z^{(2)} = \theta^{(1)} a^{(1)} \tag{6}$$

$$\mathrm{a}^{(2)} = \mathrm{g}\left(\mathrm{z}^{(2)}\right) (\text{add } \mathrm{a}_0^{(2)}) \tag{7}$$

$$\mathrm{z}^{(3)} = \theta^{(2)} \mathrm{a}^{(2)} \tag{8}$$

$$\mathrm{a}^{(3)} = \mathrm{g}\left(\mathrm{z}^{(3)}\right) (\text{add } \mathrm{a}_0^{(3)}) \tag{9}$$

$$\mathrm{z}^{(4)} = \theta^{(3)} \mathrm{a}^{(3)} \mathrm{a}^{(4)} = \mathrm{h}_\theta(\mathrm{x}) = \mathrm{g}\left(\mathrm{z}^{(4)}\right) \tag{10}$$

Once the resulting hypothesis is calculated, the backpropagation algorithm is used to compute each unit's error in the training example. Equations (11) – (13) depicts the sequence of operations for backward propagation. The errors for layer l are denoted as $\boldsymbol{\delta^{(l)}}$. In this approach, the errors for the output layer are calculated first. Subsequently we continue with layer l by backpropagating the errors in layer $l + 1$.

$$\delta^{(4)} = \mathrm{h}_\theta(\mathrm{x}) - \mathrm{y} = \mathrm{a}^{(4)} - \mathrm{y} \tag{11}$$

$$\delta^{(3)} = (\theta^{(3)})^{\mathrm{T}} \delta^{(4)} .* \mathrm{g}'(\mathrm{z}^{(3)}) \tag{12}$$

$$\delta^{(2)} = (\theta^{(2)})^{T} \delta^{(3)} .* g'(z^{(2)}) \tag{13}$$

When training the neural network, the errors computed for each training example are used to calculate the overall partial derivatives for the entire training set. Fig. 6 shows the implementation of the backpropagation algorithm.

Algorithm 1 Backpropagation Algorithm	
1:	training_set = [(x1, y1), (x2, y2), …, (xm, ym)]
2:	delta[l] = 0 (for all i, j, l)
3:	for i = 1; i < m; do
4:	a[1] = x[1]
5:	for l = 2; l <= L; l++ do
6:	forward_propagation(a[l])
7:	end for
8:	error(L) = a[L] – y[i]
9:	for j = 1; L – j >= 2; j++ do
10:	compute_errors(error(L-j))
12:	end for
13:	for l = 1; h >= ; h++ do
14:	delta[l] = delta[l] + error(l+1) *. transpose(a[l])
15:	end for
16:	end for
17:	partial_derivatives = delta[l] / m

Fig. 8. Backpropagation algorithm

The convention for assigning values is the "=". Line 1 assigns a 1-dimensional array of key pairs to variable training_set; xn contains the training example and yn the ground truth. In Line 2 the accumulated errors delta[l] for layer l, row i, and column j are initialized to zero. Lines 3-16 execute the forward propagation, error computation, and error accumulation for each training example. Finally, in line 17 the partial_derivatives variable stores the partial derivatives using the accumulated errors.

In summary, the backpropagation algorithm can be constructed by utilizing the following linear algebra operators:

- Vector/Matrix - Scalar operations: +, -, * , /
- Vector/Matrix unary operations: transpose
- Vector/Matrix binary operations: +, -, *, /
- Vector/Matrix element-wise operations: .*
- Vector/Matrix aggregate operations: sum

In addition, although not part of linear algebra, iteration constructs are also necessary for calculating the forward and back propagation of each training sample.

V. Array Query Language Extensions

This section comprehends the definition of two queries that constitute the training and prediction routines of a neural network. It is worth mentioning that it is not worth investing time and effort developing new implementations of linear algebra operators such as tensor transpose or tensor multiplication. There are already efficient open-source implementations of these operators that can be used ad-hoc to perform these operations inside an array database, such as BLAS [15], LAPACK [16], or architecture specific libraries such as Intel MKL [17].

In this section, we focus on a fully connected feed forward neural network. We define two queries that constitute the training and prediction routines of a neural network. For each routine, we define the queries inputs and expected outputs.

A. Training Query

This query has the objective of calculating the weights and biases that best fit the training examples to the ground truth. Inputs are the following:

- Training Data (*X*): MDD array containing the training examples. NOTE: For simplicity we consider only rasdaman's atomic types, like the ones in C.

- Ground Truth (*Y*): MDD array containing the labels or ground truth. Same as with the training data we consider only rasdaman's atomic types.
- Initial Weights (*initial_weights*): MDD array containing the user defined initial weights. If none is provided, then a MDD with all values equal to zero will be assumed.
- Initial Biases (*initial_biases*): MDD array containing the user defined initial biases. If none is provided, then a MDD with all values equal to zero will be assumed.
- Learning Rate (*alpha*): Scalar value which determines the steps which are taken in the minimization of the cost function.
- Number of Training Examples (*m*): Scalar value that determines the number of training examples in the input training data *X* this number can be determined automatically by rasdaman or provided by the user.
- Activation Function (default: sigmoid): It is the function that determines the activation value of each neuron. It can be expressed by the user either in mathematical form or using UDFs.
- Activation Function Derivative (default: sigmoidDerivative): The function is used in the back propagation step to update the errors MDD. Same as with activation function it can be expressed by the user either in mathematical form or using UDFs.

The outputs can either be temporary maintained in main memory or ingested into a collection in rasdaman. They are the following:

- Learned Weights (*learned_weights*): MDD array containing the trained weights.
- Learned Biases (*learned_biases*): MDD array containing the trained weights.

We commence on line 1 be defining a MARRAY construct as an iterator, the outer loop *MARRAY i IN [0:10] AS iters* represents the epochs which are user specified and represent how many times the neural network will adjust the weights and biases for all the layers in the network. Lines 2-36 encapsulate the forward and backpropagation routines defined in section IV. From lines 3-23, we are inside the the MARRAY iterator and start performing the forward pass of the neural network, for that line 3 defines another MARRAY construct *MARRAY j IN [0:10] AS layers* that represents each of the layers of the neural network; These are defined by the user. Next, on lines 3-23 a second MARRAY iterator construct is defined which encapsulates a *case* clause which performs four operations: 1) Update of the weights MDD, 2) Update of the bias MDD, 3) Update of the activations MDD, and 4) Provide control for when to calculate the errors. Lines 5-9 define the operations necessary in the *weights* case; the algorithm starts with the first step of the iteration, i.e,. $i = 0$, in line 7 where an already defined *initial_weights* MDD loads the user-specified weights on a training weights MDD called *iters.weights*. Next, the the *biases* case in line 12, the *initial_biases* are loaded into a training biases MDD iters.biases. Furthermore, in the *activations* case in line 18, the initial values of the training activations MDD *iters.activations* will be the user defined input values *X*, which is a MDD, or MDD expression (subsetting, slicing, etc), containing the training examples. Finally, the *errors* case in line 22 assures that no error during updating is done to the training errors. MDD *iters.errors* in the forward propagation step. This is reserved exclusively to the backpropagation step.

In the following steps of the iteration, i.e,. $i > 0$, the case clause will calculate the forward pass of the neural network updating the weights, biases, and activations MDDs i.e,. *iters.weights*, *iters.biases*, and *iters.activations* using the defined linear algebra operators #* (tensor multiplication), and +, - for element-wise addition and subtraction respectively. For the weights and biases case, we reference a *iters.errors* MDD in line 8 which contain the backpropagated errors from the backpropagation step. Additionally, two scalars *alpha* and *m* are introduced. The first scalar *alpha* is user defined and refers to the learning rate. The second scalar *m* can either be a value defined by the user or calculated internally and refers to the number of training examples. In the *activations* case, in line 20, an activation function must be defined. In this opportunity, we continue with the *sigmoid()* function following the feed-forward neural network example from section IV; the activation function can be replace with any other activation function, e.g., Relu, TanH, etc. The activation function can be defined either as a mathematical expression, or most convenient, by using a UDF. For the biases case, in line 13, the reduceByRow() function is introduced, this function takes a MDD as input and performs and induced sum of rows for this MDD. It is important to keep the order of the elements in the case clause when constructing the neural network expression as the order in which the weights, biases, and activations are calculated does matter and rely on the proper updating of previous MDDs. With this the forward propagation step is done.

Now we proceed with the backpropagation step. First, we introduce an *OVERLAY* operator in line 24 which in rasdaman allows to combine two equally sized MDDs by placing one "on top "of the other, in this case we use it to maintain the MDDs calculated from the forward propagation step into the backpropagation step. Next, we continue defining another *MARRAY* construct that will work as a backward iterator. The *MARRAY* construct loop *MARRAY j IN [9:1] AS layers* in line 25 iterates from the output and calculate the errors for each layer of the neural network and save the errors in a *iters.errors* MDD. When the iterator *layers* is positioned in the output layer, in this case $j=9$ in line 32, then an element-wise subtraction is performed to calculate the differences between the last layer activations from *iters.activations* and the ground truth *Y* which is user defined and same as with the input values *X* it can be a MDD or a MDD expression.

When the iterator is positioned in any layer different from the output layer, i.e., j<9 in line 33, then the errors are calculated using the errors from the previous layer, i.e., j+1 matrix multiplied by the transpose of the *iters.weights* here denoted with a *'* in *iters.weights[i][j+1]'* and the derivative of the sigmoid function, here denoted as the *sigmoidDerivative()* function. The *sigmoidDerivative()* function once again follows the same pattern as with the *sigmoid()* function defined in section IV; it can be defined as a mathematical expression or a

UDF. In this step we also make use of the element-wise multiplication with operator * that will update the errors in *iters.errors* MDD. With this we conclude the backpropagation step.

After performing the forwarding propagation and backpropagation steps the external MARRAY construct *MARRAY i IN [0:10] AS iters* in line 1 will continue to iterate until the maximum number of iterations are reached. The output will be updated weights and biases MDDs *iters.weights* and *iters.biases* respectively. These MDDs can later be used for the prediction function which will be explained in section

Query 1 Training Query	
1:	MARRAY i IN [0:10] AS iters
2:	VALUES (
3:	MARRAY j IN [0:9] AS layers
4:	VALUES {
5:	weights:
6:	case
7:	when i = 0 then initial_weights[j]
8:	else iters.weights[i-1][j] - (alpha/m) #* (iters.activations[i-1][j-1] #* iters.errors[i-1][j])
9:	end,
10:	biases:
11:	case
12:	when i = 0 then initial_biases[j]
13:	else iters.biases[i-1][j] - (alpha/m) #* reduceByRow(iters.errors[i-1][j])
15:	end,
16:	activations:
17:	case
18:	when j = 0 then X
20:	else sigmoid((iters.activations[i][j-1] #* iters.weights[i][j]) + iters.biases[i][j])
21:	end,
22:	errors: null
23:	}
24:	OVERLAY
25:	MARRAY j IN [9:1] AS layers
26:	VALUES {
27:	weights: null,
28:	biases: null,
29:	activations: null,
30:	errors:
31:	case
32:	when j = 9 then iters.activations[i][j] - Y
33:	else (iters.errors[i][j+1] #* iters.weights[i][j+1]') * sigmoidDerivative(iters.activations[i][j])
34:	end,
35:	}
36:	)

B. Prediction Query

This query uses a pretrained model and computes a prediction MDD that is the result of applying the model to one or several training examples. The model is either trained from the training algorithm in the previous section, or through another third-party, e.g., Pytorch, Tensorflow etc. The only requirement is that also needs to be converted into a MDD array in rasdaman.

The inputs are the following:

- Test Data (*X*): MDD array containing the test examples. NOTE: For simplicity we consider only rasdaman's atomic types, like the ones in C.
- Trained Weights (*trained_weights*): MDD array containing the trained weights. The trained can come from rasdaman training or using a third-party; however, it is essential that they are expressed as rasdaman MDDs.
- Trained Biases (*trained_biases*): MDD array containing the trained biases. Same as with the trained weights, they can come from rasdaman training or third party but need to be rasdaman MDDs.

The outputs are the following:

- Predicted Array (predicted_array): MDD array containing the resulting predicted values from the neural network. Same as with the output from the predicted function, the outputs can either be temporary maintained in main memory or ingested into a collection in rasdaman.
- Learned Biases (*learned_biases*): MDD array containing the trained weights.

The prediction algorithm is nothing more than calculating the forward pass for 1 or many examples of test data *X* with the particularity of using already predefined trained weights and biases. After reaching the output layer we return the *iters.activations* MDD which corresponds to our predicted_array. The prediction query is described below:

Query 2 Prediction Query	
1:	MARRAY j IN [0:9] AS layers
2:	VALUES {
3:	weights:

4:	case
5:	when i = 0 then trained_weights[j]
6:	else iters.weights[j]
7:	end,
8:	biases:
9:	case
10:	trained_biases[j]
11:	else iters.biases[j]
12:	end,
13:	activations:
14:	case
15:	when j = 0 then X
16:	else sigmoid((iters.activations[i][j-1] #* iters.weights[i][j]) + iters.biases[i][j])
17:	end,
18:	}

VI. Conclusions

In this contribution we presented an implementation roadmap for neural network in array databases using the array database rasdaman. We commenced highlighting the linear algebra operations that are present in neural networks and later expressed the neural network training and prediction algorithms using rasdaman's array query language rasql. By expressing a neural network using the array query language we get many of the query and storage optimizations that come with the array database. The biggest advantage is that we can avoid copying data all together. Training data, test data, and model itself will always reside on the server and experiments can be customized by simply using rasdaman's slicing and subsetting operators.

VII. Future Work

In this contribution we presented an implementation roadmap for neural network in array databases using the array database rasdaman. We commenced highlighting the linear algebra operations that are present in neural networks and later expressed the neural network training and prediction algorithms using rasdaman's array query language rasql. By expressing a neural network using the array query language we get many of the query and storage optimizations that come with the array database. The biggest advantage is that we can avoid copying data all together which generates a considerable amount of storage consumption and . Training data, test data, and model itself will always reside on the server and experiments can be customized by simply using rasdaman's slicing and subsetting operators. Finally, we would like to make this solution domain agnostic by implementing in datacubes in fields or science like medicine, astrophysics, economics etc.

Acknowledgment

The authors are grateful to Begüm Demir, Kai Norman Clasen, and Leonard Hackel for the valuable discussions and insights on various neural networks details. This work is funded by the German Federal Ministry for Economic Affairs and Climate Action as project AI-Cube under contract 50 EE 2012. We also want to thank our colleague Dimitar Misev for laying the basis of the extensions in rasql to express neural network operations.

References

[1] P. Baumann, "The Datacube Manifesto", http://earthserver.eu/tech/datacube-manifesto", seen 2017-08-20.

[2] Baumann, P., Misev, D., Merticariu, V. et al. Array databases: concepts, standards, implementations. J Big Data 8, 28 (2021). https://doi.org/10.1186/s40537-020-00399-2

[3] 2001. The Perl Journal. Sys Admin 10, 12 (December 2001), 47–49.

[4] Dr. Dobb's journal staff. 2002. Dr. Dobb's news & views. Dr. Dobb's J. 27, 6 (June 2002), 14.

[5] Escobar, Otoniel José Campos, Dimitar Misev, and Peter Baumann. "Making an Array Database Language Server-Side Extensible." 2020 IEEE International Conference on Big Data (Big Data). IEEE, 2020.

[6] EarthData, "Earthdata cloud evolution", https://earthdata.nasa.gov/eosdis/cloud-evolution, seen 2022-nov-23.

[7] Baumann, Peter, et al. "The multidimensional database system RasDaMan." Proceedings of the 1998 ACM SIGMOD international conference on Management of data. 1998.

[8] Lobry, Sylvain, et al. "RSVQA: Visual question answering for remote sensing data." *IEEE Transactions on Geoscience and Remote Sensing* 58.12 (2020): 8555-8566.

[9] Priftis, Georgios, et al. "Pixel Based Model For High Latitude Dust Detection." *AGU 2019 Fall Meeting*. No. MSFC-E-DAA-TN76101. 2019.

[10] Hellerstein, J., Ré, C., Schoppmann, F., Wang, D. Z., Fratkin, E., Gorajek, A., ... & Kumar, A. (2012). The MADlib analytics library or MAD skills, the SQL. arXiv preprint arXiv:1208.4165.

[11] Feng, Xixuan, et al. "Towards a unified architecture for in-RDBMS analytics." Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012.

[12] Park, Kwanghyun, et al. "End-to-end Optimization of Machine Learning Prediction Queries." arXiv preprint arXiv:2206.00136 (2022).

[13] ISO, "9075-15:2019 SQL/MDA (MultiDimensional Arrays)", https://www.iso.org/ standard/67382.html, seen 2022-07-27.

[14] P. Baumann: A Database Array Algebra for Spatio-Temporal Data and Beyond. Proc. Intl. Workshop on Next Generation Information Technologies and Systems (NGITS), July 5-7, 1999, Zikhron Yaakov, Israel, Springer LNCS 1649, 1999.

[15] Zhuliang Chen and Arne Storjohann. 2005. A BLAS based C library for exact linear algebra on integer matrices. In Proceedings of the 2005 international symposium on Symbolic and algebraic computation (ISSAC '05). Association for Computing Machinery, New York, NY, USA, 92–99. https://doi.org/10.1145/1073884.1073899

[16] Anderson, E., et al. "LAPACK: A Portable Linear Algebr a Library for Hi gh-Performance Computers." (1990).

[17] Wang, E. et al. (2014). Intel Math Kernel Library. In: High-Performance Computing on the Intel® Xeon Phi™. Springer, Cham. https://doi.org/10.1007/978-3-319-06486-4_7