Secondary Publication

Schüle, Maximilian; Neumann, Thomas; Kemper, Alfons

The Duck's Brain : Training and Inference of Neural Networks within Database Engines

Date of secondary publication: 24.03.2025

Version of Record (Published Version), Article

Persistent identifier: urn:nbn:de:bvb:473-irb-1071155

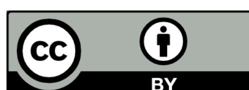
Primary publication

Schüle, Maximilian; Neumann, Thomas; Kemper, Alfons (2024): The Duck's Brain : Training and Inference of Neural Networks within Database Engines, in: Datenbank-Spektrum, Berlin ; Heidelberg: Springer, Vol. 24, Nr. 3, pp. 209–221, doi: 10.1007/s13222-024-00485-2.

Legal Notice

This work is protected by copyright and/or the indication of a licence. You are free to use this work in any way permitted by the copyright and/or the licence that applies to your usage. For other uses, you must obtain permission from the rights-holders.

This document is made available under a Creative Commons license.



The license information is available online:

https://creativecommons.org/licenses/by/4.0/legalcode

Datenbank-Spektrum (2024) 24:209–221
https://doi.org/10.1007/s13222-024-00485-2

SCHWERPUNKTBEITRAG

The Duck's Brain

Training and Inference of Neural Networks within Database Engines

Maximilian Schüle[1] · **Thomas Neumann**[2] · **Alfons Kemper**[2]

Received: 31 May 2024 / Accepted: 12 September 2024 / Published online: 9 October 2024
© The Author(s) 2024

Abstract

Although database systems perform well in data access and manipulation, their relational model hinders data scientists from formulating machine learning algorithms in SQL. Nevertheless, we argue that modern database systems perform well for machine learning algorithms expressed in relational algebra. To overcome the barrier of the relational model, this paper shows how to transform data into a coordinate relational representation for training neural networks in SQL: We f rst describe building blocks for data transformation, model training and inference in SQL-92 and their counterparts using an extended array data type. Then, we compare the implementation for model training and inference using array data types to the one using a coordinate relational representation in SQL-92 only. The evaluation in terms of runtime and memory consumption proves the suitability of modern database systems for matrix algebra, although specialised array data types perform better than matrices in coordinate relational representation.

Keywords SQL-92 · Neural Networks · Automatic Differentiation · In-Memory Database Systems · Machine Learning

1 Introduction

Modern database systems generate code to achieve a nearly hard-coded performance. In pipelined processing, code-generation eliminates interpreted function calls, so that the generated machine code processes data in-place of CPU registers. Together with modern hardware trends leading to a performance increase of database servers, code-generation allows database systems to take over more complex computations. One example for complex computations is the emergence of machine learning [8] to solve several tasks such as image classifcation or even replacing database system's components [17, 24]. These tasks rarely happen within database systems but in external tools [33, 47] requiring the data to be extracted from database systems [27]. Thus, current research mostly focuses on eliminating the extraction process [44, 49] and developing systems that combine data management and machine learning [31]. In contrast, in the paper, we argue that code generation allows database systems to perform well for machine learning when training neural networks [48] based on matrix algebra in SQL only [2, 6].

In a previous study, we stated that training neural networks in SQL is possible as long as the database system provides an array data type and recursive tables for gradient descent [41]. However, the use of an array as a nested data type interferes with the f rst normal form (referring to the defnition of arrays as a non-atomic data type) and requires copying the data between operations. Instead, to process data in-place of CPU registers, we suggested an array backend for code-generating database systems [38], which stores matrices in a coordinate relational representation (cf. Fig. 1). The coordinate relational representation stores matrices in normal form with the indices and the elements as table attributes [39]. In a vision paper, Blacher et al. [3] combined our both approaches to show that recursive CTEs (common table expressions) [10] can deal with

✉ Maximilian Schüle
maximilian.schuele@uni-bamberg.de

Thomas Neumann
neumann@in.tum.de

Alfons Kemper
kemper@in.tum.de

[1] University of Bamberg, Bamberg, Germany

[2] TUM, Munich, Germany

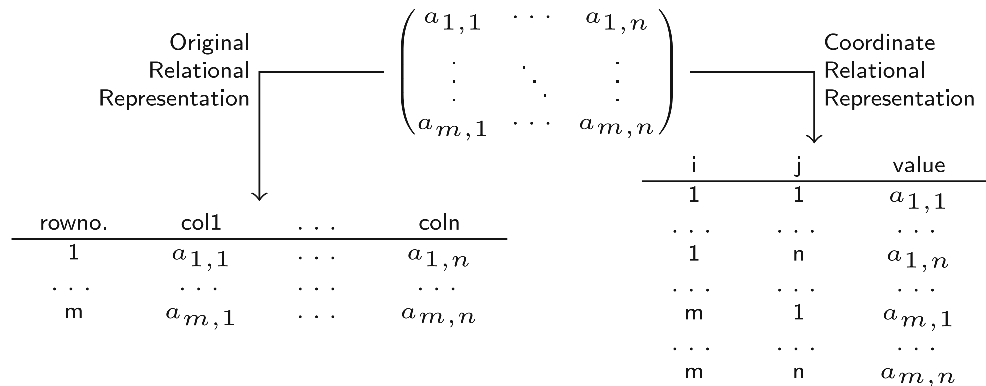


Fig. 1 Original and coordinate relational representation of matrices in database systems: the latter is used in this study for representing the weights and training neural networks

matrices in coordinate relational representation as input. Nevertheless, their study was limited to logistic regression using matrix algebra and no study has benchmarked training neural networks in SQL without further extensions such as arrays before.

In this paper, we even argue that the coordinate relational representation allows database systems to efficiently process the computations along with neural networks. In a preliminary study [40], we demonstrated the ability of database systems to train neural networks using the coordinate relational representation. For training, the performance results were promising for batched processing only and we did not focus on the memory consumption of a coordinate relational representation. Therefore, this paper extends our previous study by model inference, by measuring the memory consumption and by including DuckDB [32] as an open-source and modern database engine. We first describe the mathematical background for reverse mode automatic differentiation that is needed to understand the individual matrix operations (Sect. 2). We then discuss the intuitive implementation in Python (Sect. 3) and deduce an implementation in SQL using the coordinate relational representation (Sect. 4). This includes building blocks for data transformation using one-hot-encoding, matrix/Hadamard product and recursive tables to imitate procedural loops. Afterwards, the paper shows how to transform the coordinate relational representation into a block relational representation (arrays that are extended for matrix algebra) to support model training and inference (Sect. 5). The evaluation (Sect. 6) compares the coordinate relational representation to the use of array data types in terms of runtime and memory consumption. A Python implementation provides the baseline, whose runtime is compared depending on the batch size and the hidden layer size. Sect. 7 compares our work to existing research on (relational) matrix representations for in-database machine learning. We conclude with an outlook on optimising recursive tables for this context and on automatically generating the proposed queries (Sect. 8).

2 Backpropagation for Neural Networks

This section first describes the theoretical background for training neural networks and names the variables, which are later used to name the CTEs. Each variable represents one cached expression computed in the forward pass on function evaluation or in the backward pass on deriving the weight matrices. To discuss the derivation rules, we exemplary choose a neural network with one hidden layer. Although this limits the number of hidden layers, the derivation rules can be applied similarly to deep neural networks with further weight matrices in-between. Thus, the limitation keeps the example short enough to present the implementations in SQL.

2.1 Gradient Descent in SQL

A machine learning model can be abstracted as a parameterised model function $m_w(x)$. Its evaluation on data to classify unlabelled data is called *inference*. *Model training* returns the optimal parameters w_∞ minimising a *loss function* $l_{x,y}(w)$, which measures the difference between given labels y and the model $m_w(x)$. Gradient descent minimises a loss function iteratively by moving in the opposite direction of steepest ascent (learning rate γ), which requires the derivations with respect to each weight (gradient): $w_{t+1} = \vec{w}_t - \gamma \nabla l_{x,y}(w_t)$, $w_\infty \approx \lim_{t\to\infty} w_t$. SQL with recursive CTEs can express the required iterations. Fig. 2 shows gradient descent for simple linear regression:

$$l_{x,y}(a,b) = (m_{a,b}(x) - y)^2 = (a \cdot x + b - y)^2, \tag{1}$$

$$\nabla l_{x,y}(a,b) = \begin{pmatrix} \partial l/\partial a \\ \partial l/\partial b \end{pmatrix} = \begin{pmatrix} 2(ax+b-y)\cdot x \\ 2(ax+b-y) \end{pmatrix}. \tag{2}$$

The base case initialises the weights, each recursion maps to an iteration that updates the weights based on manually derived gradients. Sect. 4.2 uses recursion with the derivations of Sect. 2.3 for a simple neural network as model.

```sql
create table data (x float, y float);
insert into data ...

with recursive w (id, a, b) as (
 select 0,1::float,1::float
union all
 select id+1, a-0.01*avg(2*x*(a*x+b-y)),
              b-0.01*avg(2*(a*x+b-y))
 from w, data where id<5 group by id,a,b)
select * from w order by id;
```

Fig. 2 Gradient descent in SQL for linear regression: $m_{a,b}(x) = a \cdot x + b \approx y$, five iterations and $\gamma = 0.01$

2.2 Inference of Neural Networks

Neural networks consist of subsequently applied matrix multiplications each followed by an activation function. They transform an input vector x with m attributes into a vector of probabilities for l categories. With one hidden layer of size h, we gain two weight matrices $w_{xh} \in \mathbb{R}^{m \times h}$ and $w_{ho} \in \mathbb{R}^{h \times l}$. The first one computes the vector $a_{xh} \in \mathbb{R}^h$ for the hidden layer, the second one the result vector $a_{ho} \in \mathbb{R}^l$. Each activation function returns a normalised value (e.g. $\text{sig}(x) \in [0,1]$, Eq. 3) that is interpreted as the probability per category. The result vector is compared to the one-hot-encoded categorical label (y_{ones}). The difference is elementwisely taken to the power of two ($\square^{\circ 2}$), which is called mean squared error, a common loss function (Eq. 5).

$$\text{sig}(x) = (1 + e^{-x})^{-1}, \tag{3}$$

$$m_{w_{xh},w_{ho}}(x) = \underbrace{\text{sig}(\overbrace{\text{sig}(x \cdot w_{xh})}^{a_{xh}} \cdot w_{ho})}_{a_{ho}}, \tag{4}$$

$$l(x, y_{ones}) = (m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}. \tag{5}$$

After computing the loss, reverse mode automatic differentiation computes the derivatives per weight matrix in one pass. Reverse mode derives a function $f(g(l))$ by decomposing and partially deriving its parts in top-down order: $\frac{\partial f(g(l))}{\partial l} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial l}$.

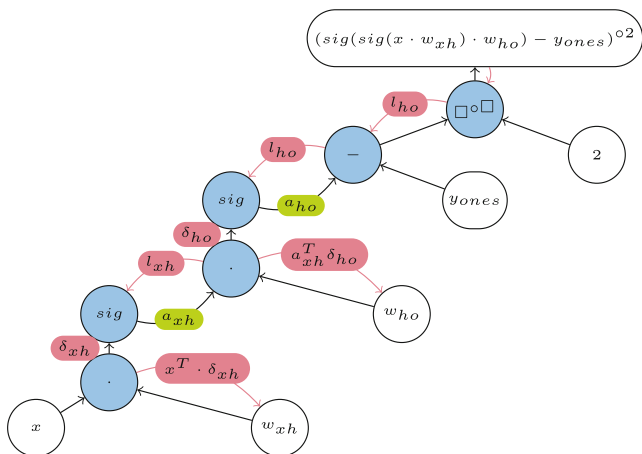


Fig. 3 Automatic differentiation for $(m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}$ with forward (green) and reverse (red) pass

2.3 Derivation

By step-wise applying the derivation rules, we obtain the expression tree shown in Fig. 3. The derivative of mean squared error calculates the difference between propagated probabilities and the one-hot-encoded labels (Eq. 6). The value gets propagated as initial seed value. Each seed value is elementwise multiplied to each partial derivation, so either the derivation of each activation function (Eqs. 7, 9) or the matrix multiplication (Eq. 8). Finally, the derivation of each weight matrix times the learning rate γ is subtracted from the weight matrix to form the updated weights (Eqs. 10, 11).

$$l_{ho} = 2 \cdot (m_{w_{xh},w_{ho}}(x) - y_{ones}), \tag{6}$$

$$\delta_{ho} = l_{ho} \circ \text{sig}'(a_{ho}) = l_{ho} \circ a_{ho} \circ (1 - a_{ho}), \tag{7}$$

$$l_{xh} = \delta_{ho} \cdot w_{ho}^T, \tag{8}$$

$$\delta_{xh} = l_{xh} \circ \text{sig}'(a_{xh}) = l_{xh} \circ a_{xh} \circ (1 - a_{xh}), \tag{9}$$

$$w_{ho}' = w_{ho} - \gamma \cdot a_{xh}^T \cdot \delta_{ho}, \tag{10}$$

$$w_{xh}' = w_{xh} - \gamma \cdot x^T \cdot \delta_{xh}. \tag{11}$$

3 Implementation in Python

Having defined the equations for training a neural network, we can deduce a Python implementation (Fig. 4) that uses

```python
import numpy as np
# load data
arr = np.loadtxt("iris.csv", delimiter=",",
        dtype=float,skiprows=1)
X = arr[:,0:4]/10
y = arr[:,4].astype(int)
# one-hot-encode y
y_oh = np.zeros((y.size, y.max()+1))
# one-hot-encode||
y_oh[np.arange(y.size),y] = 1
# initialise weights
np.random.seed(1)
w_xh = 2*np.random.random((X[0].size,20))-1
w_ho = 2*np.random.random((20,3)) - 1
# train
for j in range(10):
 print("Iteration:␣" + str(j))
 # sigmoid(x*w_xh)
 a_xh = 1/(1+np.exp(-np.dot(X,w_xh)))
 # sigmoid(a_xh*w_ho)
 a_ho = 1/(1+np.exp(-np.dot(a_xh,w_ho)))
 l_ho = 2*(a_ho - y_oh)
 print("Loss:␣"+str(np.mean(np.abs(l_ho))))
 d_ho = l_ho * a_ho * (1-a_ho)
 l_xh = d_ho.dot(w_ho.T)
 d_xh = l_xh * a_xh * (1-a_xh)
 w_ho -= 0.01 * a_xh.T.dot(d_ho)
 w_xh -= 0.01 * X.T.dot(d_xh)
```

Fig. 4 Training a neural network with NumPy

```
1  m.dot(n)             # matrix multiplication
2  m * n                # hadamard multiplication
3  1/(1+np.exp(-m))     # sigmoid function
4  m.T                  # transpose
```

Fig. 5 Building blocks for matrices in NumPy

```
1  -- create two matrices m and n
2  create table m (i int, j int, v float);
3  create table n (i int, j int, v float);
4  insert into m ...
5  -- matrix multiplication
6  select m.i, n.j, SUM(m.v*n.v))
7  from m inner join n on m.j=n.i
8  group by m.i, n.j
9  -- hadamard multiplication
10 select m.i, m.j, m.v*n.v
11 from m inner join n on m.i=n.i and m.j=n.j
12 -- sigmoid function
13 select i, j,  1/(1+exp(-v)) from m;
14 -- transpose
15 select i as j, j as i, v from m;
```

Fig. 6 Building blocks for matrices in SQL-92

NumPy for data loading (line 3), transformation (lines 4–8) and generating randomised weights (lines 11–13). Afterwards, a procedural loop (line 15) performs gradient descent that updates the weights according to the derivation rules in each iteration (lines 16–27). So each variable represents one equation needed to backpropagate the loss.

4 Implementation in SQL-92

In order to update the weight matrices of neural networks in SQL, we need to map matrix multiplication ($X \cdot Y$), function application ($f(X)$) and elementwise operations (addition: $X + Y$, Hadamard multiplication $X \circ Y$) to the coordinate relational representation in SQL. For binary elementwise operations such as Hadamard multiplication or addition/subtraction, a join on the indices combines both tables so that the arithmetic operation is part of the select-clause. Multiplication of two matrices $m \in \mathbb{R}^{m \times o}$ and $n \in \mathbb{R}^{o \times n}$ with equal inner dimensions is defined as the sum of the product over o row/column elements for each entry $(m \cdot n)_{ij} = \sum_{k=1}^{o} m_{ik} n_{kj}$. In relational algebra, this means a join on the inner index, followed by a summation: $\gamma_{m.i,n.j,\text{sum}(m.v \cdot n.v)}(m \bowtie_{m.j=n.i} n)$. To transpose a matrix in coordinate relational representation, only the indices have to be renamed. The corresponding SQL building blocks are shown in Fig. 6 with their NumPy counterparts in Fig. 5.

4.1 Transformation into Relational Representation

To train the neural network in SQL, we first have to convert the data into the coordinate relational representation (Fig. 7, 8). Therefore, we create a table of two indices and a value corresponding to the two-dimensional feature matrix (`img`: $\{[i, j, v]\}$, line 3). We assign a column index j to each attribute of the original input table (lines 5–12) and use the row number as index i. Afterwards, we one-hot-encode the label: We generate a sparse matrix containing only the one values (line 15) and a matrix shape—defined by all indices within the dimensions—out of `null` values (lines 17–21). Then, an outer join (lines 15/22) combines both tables and assigns zero to missing values (`coalesce`: line 14).

4.2 Training in SQL-92

After transforming the data, we can create and initialise the weights again in coordinate relational representation. Using `generate_series` according to the matrix dimensions together with `random`, we initialise all required weight matrices.

The feature matrix in coordinate relational representation forms the input for training the neural network within a recursive CTE (Fig. 10) that computes the weights per iteration of gradient descent. As we need to compute all weights within the recursive CTE, a unique number (`id`) identifies each weight matrix. Thus a union of all weight matrices forms the base case for the recursion. Within the recursive step, nested CTEs help to evaluate the model (lines 8–19), to backpropagate the loss (lines 20–33) and to compute the derivative per weight matrix (lines 34–45). The first CTE `w_`—just referring to the original weights (Fig. 9)—is necessary, as PostgreSQL only allows one reference to the recursive table. Each following CTE computes one matrix operation, so either a matrix or a Hadamard multiplication, whose CTE name refers to the variable name (cf. Sect. 2.2). Finally, the weights were updated by subtracting their derivatives (lines 47–49).

row	sepal length	s. width	petal length	p. width	species
1	5.1	3.5	1.4	0.2	0
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
150	5.9	3.0	5.1	1.8	2

One-Hot-Encoded

i	j	v
1	1	1
1	2	0
1	3	0
...	...	...
150	3	1

Feature Matrix

i	j	v
1	1	5.1
1	2	3.5
1	3	1.4
1	4	0.2
...	...	...

Fig. 7 Transformation of the original data set into the coordinate relational representation

```
1  create table if not exists iris (id serial,
       sepal_length float, sepal_width float,
       petal_length float, petal_width float,
       species int);
2  copy iris from './iris.csv' delimiter ','
       HEADER CSV;
3  create table img (i int, j int, v float);
4  create table one_hot(i int, j int, v int);
5  insert into img (
6    select id,1,sepal_length/10 from iris);
7  insert into img (
8    select id,2,sepal_width/10 from iris);
9  insert into img (
10   select id,3,petal_length/10 from iris);
11 insert into img (
12   select id,4,petal_width/10 from iris);
13 insert into one_hot(
14  select n.i, n.j, coalesce(i.v,0), i.v
15  from (select id,species+1 as species,1 as v
16       from iris) i right outer join
17   (select a.i, b.j
18    from (select generate_series as i
19          from generate_series(1,select count
               (*) from iris)) a,
20         (select generate_series as j
21          from generate_series(1,4)) b
22  ) n on n.i=i.id and n.j=i.species);
```

Fig. 8 Data transformation: Feature matrix `img` and one-hot-encoded label `one_hot`

```
1  create table w_xh (i int, j int, v float);
2  create table w_ho (i int, j int, v float);
3  insert into w_xh (
4   select i.*,j.*,random()*2-1
5   from generate_series(1,4) i,
6   generate_series(1,20) j);
7  insert into w_ho (
8   select i.*,j.*,random()*2-1
9   from generate_series(1,20) i,
10       generate_series(1,3) j);
```

Fig. 9 Create and initialise weights in SQL-92

4.3 Inference in SQL-92

In order to predict the accuracy of the trained weights, an SQL query measures the number of correctly classifed labels (Fig. 11). Evaluating the model (lines 8–17) returns a vector of probabilities per tuple and category. The SQL query ranks the predicted probabilities per tuple (line 2) and the one-hot-encoded vector of the original labels (line 19) to compare whether the index of the highest probability matches the index of the one value (line 23). Although window functions were used for the ranking, they could be replaced by an anti-join using `not exists` to conform SQL-92.

Fig. 12 shows the query plan for model inference (without prediction accuracy) in a coordinate relational representation. Each weight matrix is materialised for the equi-join, so the pipeline for the data can fo w until reaching an aggregation. Each matrix multiplication consists of an aggregation and thus forms a full pipeline breaker. As we will later see in the evaluation, this limits the number of pos-

```
1  with recursive w (iter,id,i,j,v) as (
2   (select 0,0,* from w_xh union
3    select 0,1,* from w_ho)
4   union all
5   ( with w_ as (
6    -- recursive reference only allowed once in PSQL
7     select * from w
8    ), a_xh(i,j,v) as ( -- sig(img * w_xh)
9     select m.i, n.j, 1/(1+exp(-SUM(m.v*n.v)))
10    from img as m inner join w_ as n on m.j=n.i
11    where n.id=0 and -- w_xh
12      n.iter=(select max(iter) from w_)
13    group by m.i, n.j
14   ), a_ho(i,j,v) as ( -- sig(a_xh * w_ho)
15    select m.i, n.j, 1/(1+exp(-SUM(m.v*n.v)))
16    from a_xh as m inner join w_ as n on m.j=n.i
17    where n.id=1 and  -- w_ho
18      n.iter=(select max(iter) from w_)
19    group by m.i, n.j
20   ), l_ho(i,j,v) as ( -- 2 * (a_ho-y_ones)
21    select m.i, m.j, 2*(m.v-n.v)
22    from a_ho as m inner join one_hot as n
23         on m.i=n.i and m.j=n.j
24   ), d_ho(i,j,v) as ( -- l_ho ° a_ho ° (1-a_ho)
25    select m.i, m.j, m.v*n.v*(1-n.v)
26    from l_ho as m inner join a_ho as n
27         on m.i=n.i and m.j=n.j
28   ), l_xh(i,j,v) as ( -- d_ho * w_ho^ T
29    select m.i, n.i as j, SUM (m.v*n.v)
30    from d_ho as m inner join w_ as n on m.j=n.j
31    where n.id=1 and
32      n.iter=(select max(iter) from w_)  -- w_ho
33    group by m.i, n.i
34   ), d_xh(i,j,v) as ( -- l_xh ° a_xh ° (1-a_ho)||
35    select m.i, m.j, m.v*n.v*(1-n.v)
36    from l_xh as m inner join a_xh as n
37         on m.i=n.i and m.j=n.j
38   ), d_w(id,i,j,v) as (
39    select 0, m.j as i, n.j, SUM (m.v*n.v)
40    from img as m inner join d_xh as n on m.i=n.i
41    group by m.j, n.j
42    union
43    select 1, m.j as i, n.j, SUM (m.v*n.v)
44    from a_xh as m inner join d_ho as n on m.i=n.i
45    group by m.j, n.j
46   )
47  select iter+1, w.id, w.i, w.j, w.v-0.01*d_w.v
48  from w_ as w, d_w
49  where iter < 20 and w.id=d_w.id and w.i=d_w.i and w.j=d_w.j
50  )
51 ) select * from w;
```

Fig. 10 Training a neural network in SQL-92

```
1  select iter, count(*)::float/(
2     select count(distinct i) from one_hot)
3  from (
4   select *, rank() over
5      (partition by m.i,iter order by v desc)
6   from (
7    select iter,
8     m.i,n.j,1/(1+exp(-sum(m.v*n.v))) as v
9    from (
10    select iter,
11     m.i,n.j,1/(1+exp(-sum(m.v*n.v))) as v
12    from img AS m, w as n
13    where m.j=n.i and n.id=0
14    group by m.i, n.j, iter ) AS m
15   inner join w as n on m.j=n.i
16   where n.id=1 and n.iter=m.iter
17   group by m.i, n.j, m.iter
18  ) m ) pred,
19  (select *, rank() over (partition by m.i
        order by v desc) from one_hot m) test
20 where pred.i=test.i and pred.rank = 1
21       and test.rank=1
22 group by iter, pred.j=test.j
23 having (pred.j=test.j)=true
24 order by iter
```

Fig. 11 Prediction in SQL:2003 (with window functions)

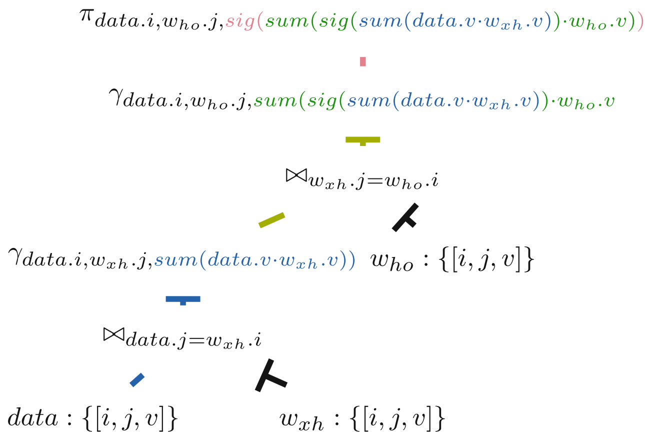


Fig. 12 Query plan for model inference in coordinate relational representation: Five pipelines, one for materialising each weight matrix (black), one for each matrix multiplication (blue, green), one for the output (red)

```
1  create table weights (
2     w_xh float[][], w_ho float[][])
3  insert into weights (select
4   (select array_agg(js order by i) from (
5      select array_agg(v order by j) as js
6      from w_xh group by i) tmp),
7   (select array_agg(js order by i) from (
8      select array_agg(v order by j) as js
9      from w_ho group by i) tmp));
```

Fig. 13 Transformation from coordinate relational into block relational representation

sibly processed tuples for model inference by the memory required to materialise the data within each aggregation. A solution to overcome materialisation for an aggregation would be another physical implementation: Instead of hashing the values for aggregation, sorted input (ordered by the index for the row number `i`) would allow for continuous output. This would work well for data stream management systems and when the result table can be stored to disk.

5 Implementation in SQL with Arrays

The presence of an extended array data type that supports matrix algebra allows SQL to train models without the need for one CTE representing each matrix. In a previous study [37], we compared the performance of gradient descent as an operator to using recursive tables. In contrast, this section focuses on the transformation of the coordinate relational into block relational representation (array), its usage for training within a recursive table and inference as later used in the evaluation for comparison with the coordinate relational representation.

5.1 Transformation into Block Relational Representation

Transforming from coordinate relational into block relational representation means reconstructing the matrix based on its indices created in Sect. 4.1. Fig. 13 shows the transformation by frst creating a table for the weight matrices with one attribute (`float[][]`) per weight matrix (line 1–2). Afterwards, the matrix in coordinate relational representation can be constructed for which `array_agg` as aggregation function is used: First for the inner dimension (`j`), we group by the higher dimension (`i`) and—within the aggregation—sort by the current dimension that is condensed (`j`) (line 5, 8). Finally for the last dimension (`i`) we can group the matrix together without any aggregation key (line 4, 7).

5.2 Training

A recursive table representing the weights is used for model training using gradient descent similar to Sect. 4.2. The base case takes the initial weights as input and assigns them the number 0, marking them as the initial weights for the frst iteration (Fig. 14, line 2). The recursive case evaluates the model function frst (line 13–16), computes the loss (line 12) and applies the backpropagation rules (line 8–11) to fnally update the weights per iteration (line 4–6). The required extensions to an array data type are elementwise operations (`-`: subtraction, `*`: Hadamard multiplication $\circ$), matrix multiplication (`**`), elementwise function application like a map function (here hard-coded for `sig`), transposition, and elementwise aggregation (`sum`).

5.3 Inference

For model inference, the model function has to be evaluated similarly as for training but with the optimal weights.

```
1  with recursive w (id,w_xh,w_ho) as (
2   select 0, w_xh, w_ho from weights
3  union all
4   select id+1,
5    w_xh - 0.01 * sum(transpose(img)*d_xh),
6    w_ho - 0.01 * sum(transpose(a_xh)*d_ho)
7   from (
8    select l_xh**(a_xh**(1-a_xh)) as d_xh, *
9    from(select d_ho*transpose(w_ho) as l_xh,*
10    from (
11     select l_ho**(a_ho**(1-a_ho)) as d_ho, *
12     from ( select 2*(a_ho-one_hot) as l_ho,*
13      from ( select sig(a_xh*w_ho) as a_ho, *
14       from ( select sig(img*w_xh) as a_xh, *
15        from ( select * from data),w
16        where id < 20))))))
17   group by id, w_ho, w_xh
18 ) select * from w:
```

Fig. 14 Backpropagation for a neural network using a recursive table and an array data type

```
1 with test as (
2  select correct, count(*) as ct from (
3   select highestposition(sig(sig(img**w_xh)
        **w_ho))=highestposition(one_hot) as
        correct
4   from data,weights)
5  group by correct)
6 select ct*1.0/(select sum(ct) from test t2)
7 from test t1 where correct=true;
```

Fig. 15 Accuracy evaluation on predicted probabilities with an array data type

In order to calculate the prediction accuracy defned as the number of correctly classif ed labels, the output vector of probabilities has to be compared to the one-hot-encoded origin label of a test data set. So the index of the highest probability should match the position of the one in one-hot-encoding, for which we created an array function `highestposition` that returns the index of the maximum value (e.g., `highestposition([0,0.6,0.4])=1`). To calculate the accuracy, we count the number of correctly and incorrectly classifed labels (Fig. 15, line 2–5) and divide the number of correct labels through the number of all (line 6–7).

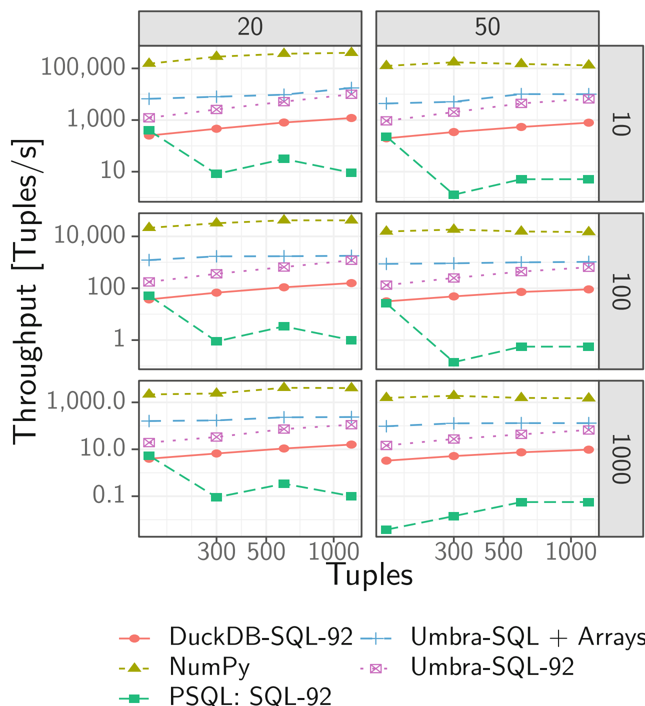


Fig. 16 Performance (Iris): Training a neural network with one hidden layer (size 20/50, 10/100/1000 iter.)

6 Evaluation

System: Ubuntu 22.04 LTS, Intel(R) Xeon(R) W-2295 CPU (18 cores @ 3.00GHz supporting hyper-threading), 128 GB DDR4 RAM.

We compare the performance of the coordinate relational representation for matrices (*SQL-92*, Sect. 4) to their representation as an array data type (*SQL + Arrays*, Sect. 5). We apply both representations for use within neural networks in SQL and let the benchmarks[1] run in Umbra [28], PostgreSQL (PSQL) 14.5 [43] and DuckDB 0.8.1 [32] as target engines. The implementation with NumPy (Fig. 4, Sect. 3) serves as the baseline. We use two different data sets: Fisher's Iris f ower data [15] (four attributes, one label) and the MNIST data [9] for image classifcatio n (ten categories, 784 pixels, excerpt of 6000 tuples).

6.1 Scaling the Number of Input Tuples

Fig. 16 shows the performance in terms of runtime and throughput using the Iris data set. As we are interested in the performance numbers and not in the model quality, we replicate the Iris f ower data set for the f rst benchmark to enable a f exible input size. A neural network with one hidden layer is trained to classify the f ower category. We vary the size of the training data set, the number of iterations and the size of the hidden layer.

6.1.1 Throughput

Although the NumPy implementation outperforms both SQL variants, the performance increase of Umbra with its in-memory performance in comparison to PostgreSQL is visible. The performance of DuckDB, as a vectorised engine without code-compilation, lies in-between: faster than PostgreSQL but slower than Umbra. Both SQL variants (SQL-92/SQL + Arrays) perform better with an increasing number of tuples per iteration. A small number of input tuples corresponds to a small batch size, leading to a small number of tuples used during one recursive step. This thwarts database systems as they excel in batched processing.

6.1.2 Memory Consumption

The downside of the coordinate relational representation is its high memory consumption for matrix multiplication. As it is not using any compression technique such as compressed sparse row, the matrices must be stored densely. Fig. 17a shows the memory consumption of Umbra for model training with inference on the whole Iris data set. We performed ten iterations on a neural network with one hidden layer of size 20 (resp. 50). Each iteration requires about

[1] https://gitlab.db.in.tum.de/MaxEmanuel/nn2sql.

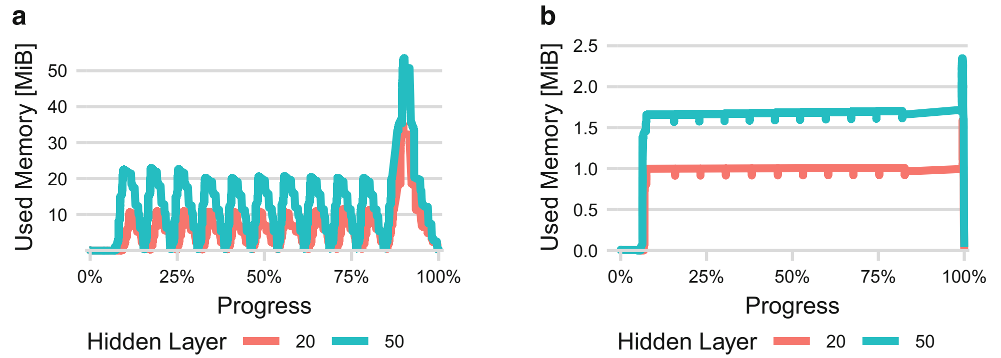


Fig. 17 Iris: Memory consumption (10 iter., batch size = 150 Tuples, hidden layer size 20/50). **a** Iris, SQL-92. **b** Iris, SQL + Arrays

Table 1 Matrix sizes: Iris data set and a neural network, one hidden layer (20)

Variable	#Entries	Size in B
$\lvert x\rvert$	$150 \cdot 4 = 600$	$600 \cdot 8$
$\lvert a_{xh}\rvert = \lvert l_{xh}\rvert = \lvert \delta_{xh}\rvert$	$150 \cdot 20 = 3000$	$3 \cdot 3000 \cdot 8$
$\lvert a_{ho}\rvert = \lvert l_{ho}\rvert = \lvert \delta_{ho}\rvert = \lvert y_{ones}\rvert$	$150 \cdot 3 = 450$	$4 \cdot 450 \cdot 8$
$\lvert w_{xh}\rvert$	$4 \cdot 20 = 80$	$80 \cdot 8$
$\lvert w_{ho}\rvert$	$20 \cdot 3 = 60$	$60 \cdot 8$
Sum		$11540 \cdot 8$

10 MiB (20 MiB) of main-memory, whereas its counterpart using arrays (cf. Fig. 17b) only requires 1 MiB (1.6 MiB).

Table 1 shows the sizes of the involved tables: The size of the weight matrices is independent of the data but depends on the size of the input/output vectors only (that are number of attributes, size of the hidden layer and number of categories). The other matrices contain one row per input tuple. For model inference, we need a total size of $(600 + 3000 + 450 + 450 + 80 + 20) \cdot 8B = 4640 \cdot 8B =$ 36.25 KiB. Training requires to materialise the matrices for backpropagation (90 KiB). As subsequent matrix operations on our array data type have not yet been optimised, we allocate a matrix for every single operation, which consumes additional memory. Storing the matrices in a coordinate relational representation consumes a threefold of the memory as the indices each consume $8B$.

6.2 Image Classification

The second benchmark simulates image classifcation based on the MNIST data set using a neural network with one hidden layer. To investigate how database systems perform for model inference, we split the measurements for image classifcation in two parts for which we benchmark model inference separately.

6.2.1 Throughput: Training

We measure the runtime for training one epoch depending on the batch size. As we can see in Fig. 18, database systems perform better the bigger the batch size is. With a higher batch size, the cost for aggregation into arrays is amortised and the SQL array data type outperforms the coordinate relational representation. PostgreSQL was not terminating after a time-out of one day, DuckDB—when working in-memory only—exceeded the size of main-memory for the neural network with the larger hidden layer. Only Umbra was capable of running all experiments. To conclude, in-memory database systems are able to carry out matrix operations as required for neural networks. Nevertheless, use-case-specif c optimisations are needed to support smaller batch sizes.

6.2.2 Throughput: Inference

As training on small batches is time-expensive in SQL-92, we argue that inference alone is worthful inside a database system to avoid data extraction. Fig. 19 shows the throughput just for model inference on the MNIST dataset in dependency on the size of the hidden layer. Unfortunately, the coordinate relational representation performs the worst, which underlies the need of a native integration for model inference. A native integration can be either an operator such as the ModelJoin [22] or a data structure with support for matrix algebra. The array data type in Umbra is such

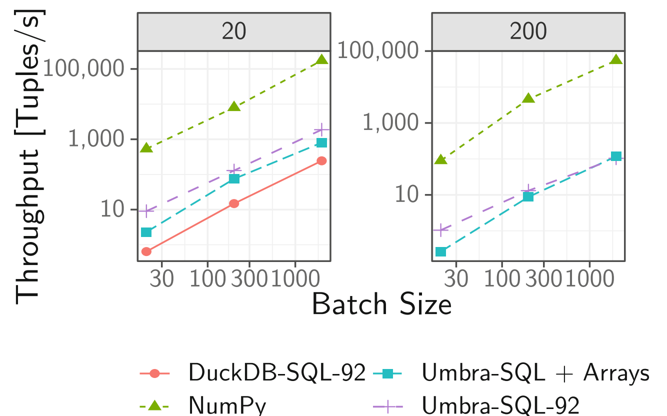


Fig. 18 Performance: Training one epoch with the MNIST data set with increasing batch size (one hidden layer size 20/200)

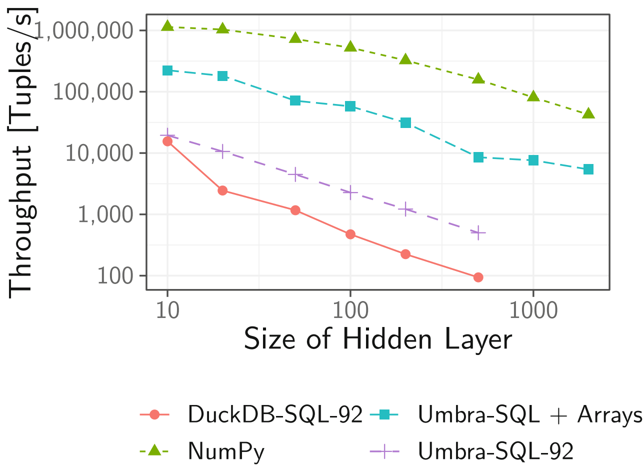


Fig. 19 Throughput for model inference (MNIST data set) depending on the size of the hidden layer

a data structure, although performing a factor of ten worse than the NumPy reference implementation. On the other side, a database integration for inference eliminates data extraction, thus eliminating additional costs. From a data engineer's perspective, one has to trade off between extraction and inference time. Further, it must be acknowledged that the array data type has not yet been optimised for subsequent operations. No optimisation leads to one separate call to the BLAS library for each operation, which decreases performance. In the future, we plan the query optimiser to detect and combine subsequent matrix operation on an array data type to be executed as a single library call.

6.2.3 Transformation

This section analyses the trade-off between avoiding data extraction from a database system and in-database preprocessing. Table 2 depicts the runtime for copying the data from CSV and its subsequent preprocessing including transformation into coordinate relational representation. Fig. 22 shows the transformation time dependent on the input size.

6.2.4 Memory Consumption: Training

As the MNIST data set contains more columns (784) and labels (10) than the Iris data set, also the size of the intermediate tables grow within each iteration. Fig. 20 shows the memory consumption within Umbra for training (without inference) in a coordinate relational representation depending on the batch size for four iterations (not a complete epoch as the number of iterations would not match otherwise). The higher the size of the hidden layer or the batch size, the higher the required memory for backpropagation gets. The memory consumption reaches 25 GiB per iteration, whereas—when supporting arrays—only a fraction of the memory of its relational counterpart is required (less than 5 MiB per iteration, cf. Fig. 21).

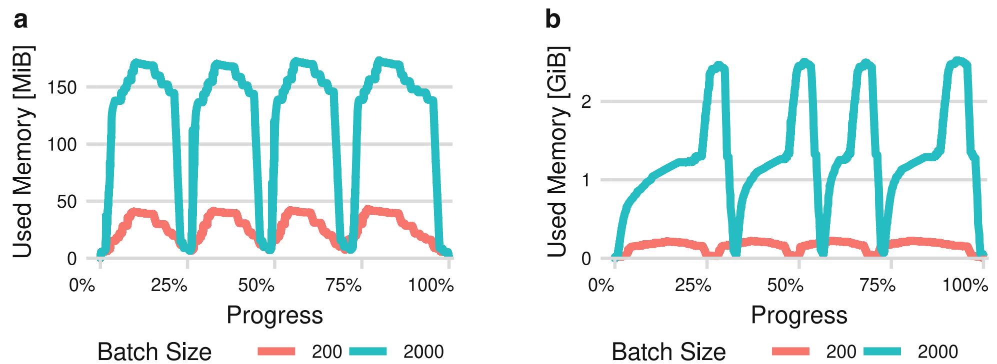


Fig. 20 Training (MNIST), SQL-92: Memory consumption (batch size = 200/2000 tuples). **a** Hidden layer size 20. **b** Hidden layer size 200

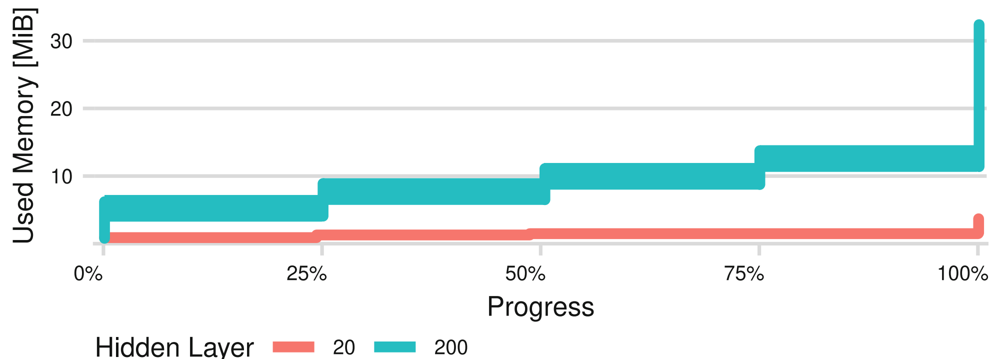


Fig. 21 Training (MNIST), SQL + Arrays: Memory consumption (batch size = 2000 tuples, hidden layer size 20/200)

Table 2 Runtime for copying and transforming 60k tuples (MNIST)

System	Copy [s]	Transform [s]
NumPy	1.50	1.61
DuckDB	1.93	3.41
Umbra	1.51	5.74

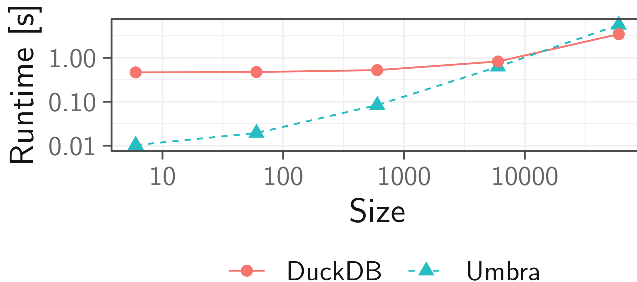


Fig. 22 Runtime for transformation from the original into the coordinate relational representation depending on the input size

6.2.5 Memory Consumption: Inference

For inference only the forward pass of backpropagation is needed, thus less memory is required per tuple. Fig. 23a shows the memory usage of the coordinate relational representation for inference: The two matrix multiplications that are involved for evaluating the model require each an aggregation in relational algebra and thus materialise the intermediate result. This limits model inference by the number of input tuples and by the number of hidden layers. To overcome these limitations, the input can be processed batchwise and stored to disk. Further, another physical implementation of the aggregation operator that uses sorting instead of hashing would allow for continuous output as required for data streams as input. An array data type reduces the memory consumption for model inference (cf. Fig. 23b) allowing for more tuples to be labelled.

7 Related Work

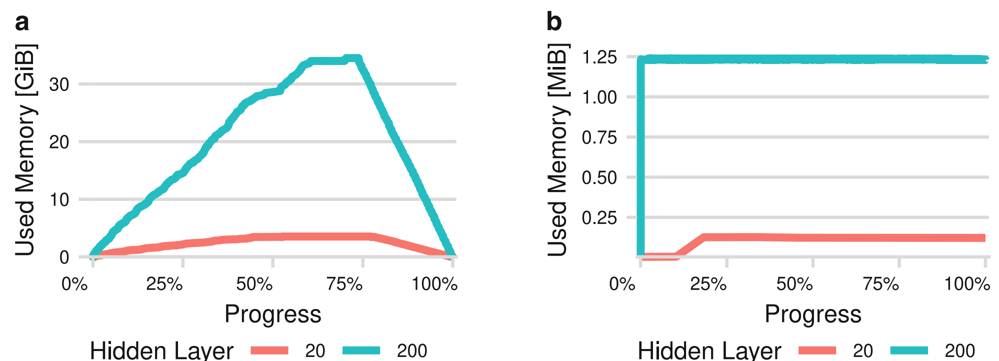
There are systems for machine learning, which enhance the application of models by integrating machine learning functionality into database systems, and machine learning for systems, which aims to replace components of systems with learned models. This paper focuses on the frst, as systems are fundamental for processing data and their enhancement is crucial for eff cient training models and their inference [26, 46, 52].

7.1 Database Systems for Machine Learning

For database systems completely taking over machine learning, the tasks can be either mapped to standardised SQL [42] as in Sect. 4 or database systems have to provide extensions [18, 45] as in Sect. 5. Kläbe et al. [22] focus on model inference, which means labelling unlabelled data based on a pre-trained model, for which they introduce the ModelJoin. They use database systems for labelling only using a relational graph representation. Paganelli et al. [29] also focus on model inference using case-when statements and one-hot-encoding to push prediction queries into the database system. Yang et al. [51] focus on user def ned functions for model inference and their optimisation.

Further studies propose to generate SQL queries for machine learning without modifying the underlying database system [11, 25]. Luo et al. [23] performed linear regression on the coordinate relational representation, but without the performance of a modern database system they required extensions to train neural networks [20]. Duta et al. [12] have proven how to translate procedural language extensions, i.e. PL/pgSQL, into recursive common table expressions (CTEs). Their approach led to a performance increase through the elimination of costly procedural constructs in favour of parallelisable recursive SQL queries. Based on this, Blacher et al. [4] use our coordinate relational representation for Einstein summation within code-generating database systems. They are focusing on the performance for matrix multiplication whereas we also measured the memory consumption. Other research focuses on database

Fig. 23 Inference (MNIST): Memory consumption (batch size 6000 Tuples, hidden layer 20/200). **a** SQL-92. **b** SQL + Arrays



systems being part of the data preprocessing pipeline and accelerating data engineering. Salazar et al. [34] combine the coordinate relational representation with indexes to accelerate training and prediction. Schleich et al. [35] let the database system pre-aggregate data to solve regression tasks in SQL. Yan et al. [50] accelerate prediction queries by detecting possible predicates to be executed within the database system beforehand.

7.2 Systems Optimisation for Linear Algebra

As the evaluation has shown that the memory consumption of matrices is limiting the amount of tuples to be processed, future extensions of database systems should incorporate research on matrix compression. Elgohary et al. [13] describe the trade-off between execution time and data size. Baunsgaard et al. [2] compress matrices depending on the linear algebra required by workload. Ferragina et al. [14] perform matrix-vector multiplication on a loss-less compressed matrix format based on compressed sparse row (CSR). Other research eliminates any relational representation by using matrices instead of tables [19].

7.3 Machine Learning Systems

Different approaches try to unify the requirements for supporting machine learning algorithms with the advantages of database systems. SystemDS [5] and SystemML [7] are machine learning systems empowered by database technology such as index structures and a declarative language that lets the user def ne customised algorithms. To support iterative algorithms on stored data within the database system, DB4ML [21] defnes an interface for iterative sub-transactions accessing stored tuples according to a data layout. The work of Asada et al. [1] pursues the counter approach: Instead of combining machine learning with database systems, they map database queries onto tensors. In this way, they reuse machine learning frameworks such as PyTorch with its extensions for hardware accelerators [16]. To specify the data layout for tensors instead of tuples, Schleich et al. [36] developed a declarative tensor query language. Other work try to avoid database systems and thus have to add functionality such as parallelisation for feature transformation [30]. In contrast to these works, we worked with database systems as the main component and with SQL as the principal query language. We argue that database systems are already optimised for modern hardware but their optimiser needs tuning for machine learning workloads.

8 Conclusion

The paper has discussed and benchmarked building blocks for training neural networks in SQL. In order to deduce the necessary SQL queries that represent matrix algebra for evaluating and training neural networks, we f rst discussed reverse mode automatic differentiation to reuse partial derivations. The partial derivations formed the foundation for nested CTEs. They were cached within a recursive CTE when deriving the weight matrices to compute the optimal weights. Instead of CTEs for each variable, we presented an extension of SQL arrays for matrix algebra and the required data transformation steps. In the evaluation, in-memory enhanced database systems, i.e. Umbra and DuckDB, showed better performance when processing data in larger batches, although they were still outperformed by NumPy in Python. Furthermore, the available memory limited the number of tuples that could be processed for model training and inference within in-memory systems. As the coordinate relational representation materialised the data for training and inference, the extended array data type addressed the limitation by using less memory. Although the set of matrix algebra extensions for array data types enabled the required operations, condensing subsequent calls would optimise memory usage further.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Asada Y, Fu V, Gandhi A, Gemawat A, Zhang L, Gupta V, Nosakhare E, Banda D, Sen R, Interlandi M (2022) Share the tensor tea: How databases can leverage the machine learning ecosystem. Proc Vldb Endow 15(12):3598–3601
2. Baunsgaard S, Boehm M (2023) AWARE: workload-aware, redundancy-exploiting linear algebra. Proc Acm Manag Data 1(1)
3. Blacher M, Giesen J, Laue S, Klaus J, Leis V (2022) Machine learning, linear algebra, and more: Is SQL all you need? In: CIDR (https://www.cidrdb.org)
4. Blacher M, Klaus J, Staudt C, Laue S, Leis V, Giesen J (2023) Eff ci ent and portable einstein summation in SQL. Proc Acm Manag Data 1(2)

5. Boehm M, Antonov I, Baunsgaard S, Dokter M, Ginthör R, Innerebner K, Klezin F, Lindstaedt SN, Phani A, Rath B, Reinwald B, Siddiqui S, Wrede SB (2020) Systemds: A declarative machine learning system for the end-to-end data science lifecycle. In: CIDR (https://www.cidrdb.org)
6. Boehm M, Interlandi M, Jermaine C (2023) Optimizing tensor computations: From applications to compilation and runtime techniques. In: SIGMOD Conference Companion. ACM, pp 53–59
7. Boehm M, Reinwald B, Hutchison D, Sen P, Evfm ievski AV, Pansare N (2018) On optimizing operator fusion plans for large-scale machine learning in systemml. Proc Vldb Endow 11(12): 1755–1768
8. Budach L, Feuerpfeil M, Ihde N, Nathansen A, Noack NS, Patzlaff H, Harmouch H, Naumann F (2022) The effects of data quality on ml-model performance. CoRR abs, vol 2207, p 14529
9. Ciresan DC, Meier U, Schmidhuber J (2012) Multi-column deep neural networks for image classifcat ion. In: CVPR. IEEE Computer Society, pp 3642–3649
10. Dietrich B, Müller T, Grust T (2022) Data provenance for recursive SQL queries. TaPP 9:1–9
11. Du L (2020) In-machine-learning database: Reimagining deep learning with old-school SQL. CoRR abs, vol 2004.05366
12. Duta C, Hirn D, Grust T (2020) Compiling PL/SQL away. In: CIDR (https://www.cidrdb.org)
13. Elgohary A, Boehm M, Haas PJ, Reiss FR, Reinwald B (2018) Compressed linear algebra for large-scale machine learning. VLDB J 27(5):719–744
14. Ferragina P, Manzini G, Gagie T, Köppl D, Navarro G, Striani M, Tosoni F (2022) Improving matrix-vector multiplication via lossless grammar-compressed matrices. Proc Vldb Endow 15(10):2175–2187
15. Fisher RA (1936) The use of multiple measurements in taxonomic problems. Ann Eugen 7(2):179–188
16. He D, Nakandala SC, Banda D, Sen R, Saur K, Park K, Curino C, Camacho-Rodríguez J, Karanasos K, Interlandi M (2022) Query processing on tensor computation runtimes. Proc Vldb Endow 15(11):2811–2825
17. Heinrich R, Luthra M, Kornmayer H, Binnig C (2022) Zero-shot cost models for distributed stream processing. In: DEBS. ACM, pp 85–90
18. Hellerstein JM, Ré C, Schoppmann F, Wang DZ, Fratkin E, Gorajek A, Ng KS, Welton C, Feng X, Li K, Kumar A (2012) The madlib analytics library or MAD skills. Sql Proc Vldb Endow 5(12):1700–1711
19. Huang Z, Chen S (2022) Density-optimized intersection-free mapping and matrix multiplication for join-project operations. Proc Vldb Endow 15(10):2244–2256
20. Jankov D, Luo S, Yuan B, Cai Z, Zou J, Jermaine C, Gao ZJ (2020) Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. Sigmod Rec 49(1):43–50
21. Jasny M, Ziegler T, Kraska T, Röhm U, Binnig C (2020) DB4ML – an in-memory database kernel with machine learning support. In: SIGMOD Conference. ACM, pp 159–173
22. Kläbe S, Hagedorn S, Sattler K (2023) Exploration of approaches for in-database ML. In: EDBT, pp 311–323 (OpenProceedings.org)
23. Luo S, Gao ZJ, Gubanov MN, Perez LL, Jermaine CM (2017) Scalable linear algebra on a relational database system. In: ICDE. IEEE Computer Society, pp 523–534
24. Maltry M, Dittrich J (2022) A critical analysis of recursive model indexes. Proc Vldb Endow 15(5):1079–1091
25. Marten D, Meyer H, Dietrich D, Heuer A (2019) Sparse and dense linear algebra for machine learning on parallel-rdbms using SQL. Open J Big Data 5(1):1–34
26. Mohr-Daurat H, Pirk H (2021) Homoiconicity for end-to-end machine learning with BOSS. Bicod ceur Workshop Proceedings 3163:46–49
27. Nath RPD, Romero O, Pedersen TB, Hose K (2022) High-level ETL for semantic data warehouses. SW 13(1):85–132
28. Neumann T, Freitag MJ (2020) Umbra: A disk-based system with in-memory performance. In: CIDR (https://www.cidrdb.org)
29. Paganelli M, Sottovia P, Park K, Interlandi M, Guerra F (2023) Pushing ML predictions into dbmss. IEEE Trans Knowl Data Eng 35(10):295–210
30. Phani A, Erlbacher L, Boehm M (2022) UPLIFT: parallelization strategies for feature transformations in machine learning workloads. Proc Vldb Endow 15(11):2929–2938
31. Raasveldt M, Holanda P, Mühleisen H, Manegold S (2018) Deep integration of machine learning into column stores. In: EDBT, pp 473–476 (OpenProceedings.org)
32. Raasveldt M, Mühleisen H (2019) Duckdb: an embeddable analytical database. In: SIGMOD Conference. ACM, pp 1981–1984
33. Renz-Wieland A, Gemulla R, Kaoudi Z, Markl V (2022) Nups: A parameter server for machine learning with non-uniform parameter access. In: SIGMOD Conference. ACM, pp 481–495
34. Salazar-Díaz R, Glavic B, Rabl T (2024) Inferdb: In-database machine learning inference using indexes. Proc Vldb Endow 17(8):1830–1842
35. Schleich M, Olteanu D, Khamis MA, Ngo HQ, Nguyen X (2019) A layered aggregate engine for analytics workloads. In: SIGMOD Conference. ACM, pp 1642–1659
36. Schleich M, Shaikhha A, Suciu D (2022) Optimizing tensor programs on f exible storage. CoRR abs, vol 2210.06267
37. Schüle ME (2023) Recursive SQL and gpu-support for in-database machine learning. Btw lni (331):931
38. Schüle ME, Götz T, Kemper A, Neumann T (2021) Arrayql for linear algebra within umbra. In: SSDBM. ACM, pp 193–196
39. Schüle ME, Götz T, Kemper A, Neumann T (2022) Arrayql integration into code-generating database systems. In: EDBT, vol 1, pp 40–41 (OpenProceedings.org)
40. Schüle ME, Kemper A, Neumann T (2023) NN2SQL: let SQL think for neural networks. Btw lni (331):183–194
41. Schüle ME, Lang H, Springer M, Kemper A, Neumann T, Günnemann S (2021) In-database machine learning with SQL on gpus. In: SSDBM. ACM, pp 25–36
42. Siksnys L, Pedersen TB, Nielsen TD, Frazzetto D (2021) Solvedb+: Sql-based prescriptive analytics. In: EDBT. OpenProceedings.org, pp 133–144
43. Stonebraker M, Rowe LA (1986) The design of postgres. In: SIGMOD Conference. ACM Press, pp 340–355
44. Störl U, Klettke M (2022) Darwin: A data platform for schema evolution management and data migration. In: EDBT/ICDT Workshops, *CEUR Workshop Proceedings*, vol 3135 (CEUR-WS.org)
45. Tang Y, Ding Z, Jankov D, Yuan B, Bourgeois D, Jermaine C (2023) Auto-differentiation of relational computations for very large scale machine learning. In: ICML, *Proceedings of Machine Learning Research*, vol 202. PMLR, pp 581–533
46. Wang Y, Wang DZ (2022) Extensible database simulator for fast prototyping in-database algorithms. In: CIKM. ACM, pp 5029–5033
47. Wenig P, Papenbrock T (2022) Datagossip: A data exchange extension for distributed machine learning algorithms. In: EDBT, pp 2:373–2:377 (OpenProceedings.org)
48. Wiese L, Höltje D (2021) Nncompare: a framework for dataset selection, data augmentation and comparison of different neural networks for medical image analysis. In: DEEM@SIGMOD. ACM, pp 6:1–6:7
49. Wingerath W, Gessert F, Ritter N (2020) Invalidb: Scalable push-based real-time queries on top of pull-based databases (extended). Proc Vldb Endow 13(12):3032–3045

50. Yan C, Lin Y, He Y (2023) Predicate pushdown for data science pipelines. Proc Acm Manag Data 1(2):136:1–136:28
51. Yang Z, Wang Z, Huang Y, Lu Y, Li C, Wang XS (2022) Optimizing machine learning inference queries with correlative proxy models. Proc Vldb Endow 15(10):2032–2044
52. Zheng Y, Tian Y, D'silva JV, Kemme B (2023) Dbmlsched: Scheduling in-database machine learning jobs. In: VLDB Workshops, *CEUR Workshop Proceedings*, vol 3462. CEUR-WS.org,

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional aff liations.