NN2SQL: Let SQL Think for Neural Networks

Maximilian E. Schüle[1] Alfons Kemper[2] Thomas Neumann[3]

Results Reproduced BTW 2023

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 relational representation for training neural networks in SQL: We first describe building blocks for data transformation in SQL. Then, we compare an implementation for model training using array data types to the one using a relational representation in SQL-92 only. The evaluation proves the suitability of modern database systems for matrix algebra, although specialised array data types perform better than matrices in relational representation.

Keywords: SQL-92, Neural Networks, Automatic Differentiation

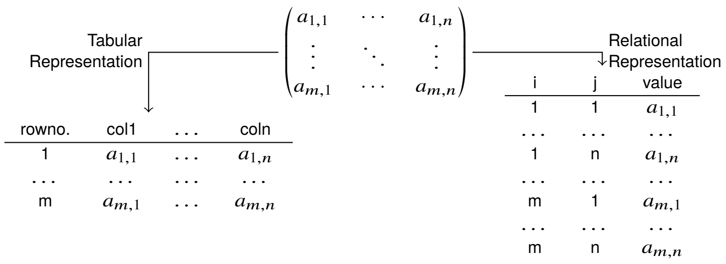
1 Introduction

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 [Bu22] to solve several tasks such as image classification or even replacing database system's components [He22; MD22]. These tasks rarely happen within database systems but in external tools [Re22; WP22] requiring the data to be extracted from database systems [Na22]. Thus, current research mostly focuses on eliminating the extraction process [Bu20; Ma15; Sc21a; SK22; WGR20] and developing systems that combine data management and machine learning [Ra18]. In contrast, in this paper, we argue that code generation allows database systems to perform well for machine learning when training neural networks [WH21] based on matrix algebra in SQL only [MAF21; OVZ22; Sa22; Sc19; Sc21d].

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 [Sc21c]. However, the use of an array as a nested data type interferes with the first normal form (referring to the definition of arrays as a non-atomic data type) and requires copying the data

[1] University of Bamberg, An der Weberei 5, 96047 Bamberg, maximilian.schuele@uni-bamberg.de
[2] TUM, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de
[3] TUM, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de

doi:10.18420/BTW2023-09



$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

rowno.	col1	...	coln
1	$a_{1,1}$	...	$a_{1,n}$
...	...	...	...
m	$a_{m,1}$	...	$a_{m,n}$

i	j	value
1	1	$a_{1,1}$
...	...	...
1	n	$a_{1,n}$
...	...	...
m	1	$a_{m,1}$
...	...	...
m	n	$a_{m,n}$

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

between operations. Instead, to process data in-place of CPU registers, we suggested an array backend for code-generating database systems [Sc21b], which stores matrices in a relational representation (cf. Figure 1). This representation stores arrays in normal form with the indices and the elements as table attributes [Sc22]. In a vision paper, Blacher et al. [Bl22] combined our both approaches to show that recursive CTEs (common table expressions) [DMG22] can deal with matrices in 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 relational representation allows database systems to efficiently process the computations along with neural networks. Therefore, this paper uses the relational representation of matrices to train neural networks. We first describe the mathematical background for reverse mode automatic differentiation that is needed to understand the individual matrix operations. We then discuss the intuitive implementation in Python and deduce an implementation in SQL using the relational representation. This includes building blocks for data transformation using one-hot-encoding, matrix/Hadamard product and recursive tables to imitate procedural loops. The evaluation compares the relational representation to the use of array data types within the Umbra database system. An implementation in Python provides the baseline, whose runtime is compared with regard to the batch size and the size of the hidden layer. We conclude with an outlook on optimising recursive tables for this context and on automatically generating the proposed queries.

2 Machine Learning in SQL

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

Algorithm 1 Automatic Differentiation (Matrices)

```
1: function DERIVE($Z, seed$)
2:     if $Z = X + Y$ then DERIVE($X$,$seed$); DERIVE($Y$,$seed$)
3:     else if $Z = X \circ Y$ then DERIVE($X$,$seed \circ Y$); DERIVE($Y$,$seed \circ X$)
4:     else if $Z = X \cdot Y$ then DERIVE($X$,$seed \cdot Y^T$); DERIVE($Y$,$seed^T \cdot X$)
5:     else if $Z = f(X)$ then DERIVE($X$,$seed \circ f'(X)$)
6:     else $\frac{\partial}{\partial Z} \leftarrow \frac{\partial}{\partial Z} + seed$
7:     end if
8: end function
```

matrices in-between. Thus, the limitation keeps the example short enough to present the implementations in SQL.

2.1 Theoretical Background

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 weights 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. $sig(x) \in [0, 1]$, Equation 1) 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 (Equation 3).

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

$$m_{w_{xh},w_{ho}}(x) = \underbrace{sig(\overbrace{sig(x \cdot w_{xh})}^{a_{xh}} \cdot w_{ho})}_{a_{ho}}, \tag{2}$$

$$l(x, y_{ones}) = (m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}. \tag{3}$$

After computing the loss, reverse mode automatic differentiation computes the derivatives per weight matrix in one pass. This mode derives a function $f(g(l))$ by decomposing and partially deriving its parts in top-down order: $\frac{\partial f(g(l))}{\partial l} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial l}$. Alg. 1 shows reverse mode automatic differentiation for matrices [Mu17]: The function `DERIVE` takes as input an arithmetic expression Z and a seed value $seed$ (the parent partial derivation). The algorithm follows pattern matching on the arithmetic expression Z to compute and further propagate the partial derivatives until arriving at a leaf node.

By step-wise applying the derivation rules, we obtain the expression tree shown in Figure 2. The derivative of mean squared error calculates the difference between propagated probabilities and the one-hot-encoded labels (Equation 4). This 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 (Equation 5, 7) or the matrix multiplication (Equation 6). Finally, the derivation of each weight matrix times the learning rate γ is subtracted from the weight matrix to form the updated weights (Equation 8, 9).

$$\begin{align} l_{ho} &= 2 \cdot (m_{w_{xh},w_{ho}}(x) - y_{ones}), \tag{4}\\ \delta_{ho} &= l_{ho} \circ sig'(a_{ho}) = l_{ho} \circ a_{ho} \circ (1 - a_{ho}), \tag{5}\\ l_{xh} &= \delta_{ho} \cdot w_{ho}^T, \tag{6}\\ \delta_{xh} &= l_{xh} \circ sig'(a_{xh}) = l_{xh} \circ a_{xh} \circ (1 - a_{xh}), \tag{7}\\ w'_{ho} &= w_{ho} - \gamma \cdot a_{xh}^T \cdot \delta_{ho}, \tag{8}\\ w'_{xh} &= w_{xh} - \gamma \cdot x^T \cdot \delta_{xh}. \tag{9} \end{align}$$

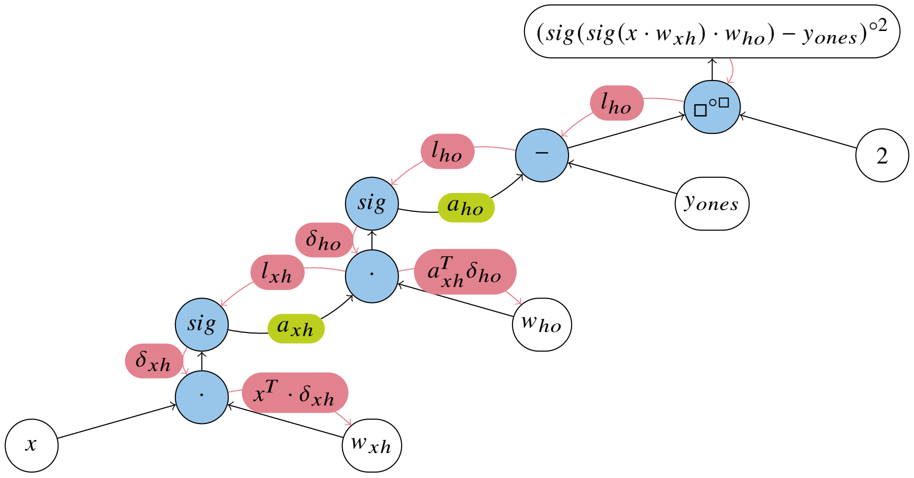


Fig. 2: Automatic differentiation for $(m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}$.

2.2 Implementation in Python and SQL-92

Having defined the equations for training a neural network, we can deduce a Python implementation (List. 1) that uses NumPy for data loading (line 3), transformation (lines 4-8) and generating randomised weights (lines 10-12). Afterwards, a procedural loop (line 14) performs gradient descent that updates the weights according to the derivation rules in each iteration (lines 15-24). So each variable represents one equation needed to backpropagate the loss.

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 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: $\gamma_{m.i,n.j,sum(m.v \cdot n.v)}(m \bowtie_{m.j=n.i} n)$. To transpose a matrix in relational representation, only the indices have to be renamed. The corresponding SQL building blocks are shown in List. 3 with their NumPy counterparts in List. 2.

```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))
y_oh[np.arange(y.size),y] = 1 # one-hot-encode: set one
# initialise weights
np.random.seed(1)
w_xh = 2*np.random.random((X[0].size,20)) - 1 # size: 4*20
w_ho = 2*np.random.random((20,3)) - 1         # size: 20*3
# train
for j in range(10):
    print("Iteration:␣" + str(j))
    a_xh = 1/(1+np.exp(-np.dot(X,w_xh)))      # sigmoid(x*w_xh)
    a_ho = 1/(1+np.exp(-np.dot(a_xh,w_ho)))   # sigmoid(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)
```

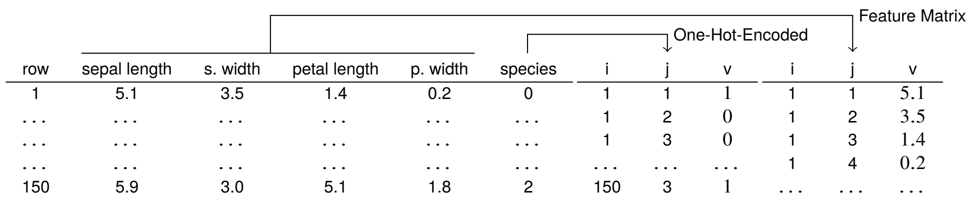
List. 1: Training a neural network with NumPy.

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

List. 2: Building blocks for matrices in NumPy.

```sql
-- create two matrices m and n
create table m (i int, j int, v float); create table n (i int, j int, v float);
insert into m ...
-- matrix multiplication
select m.i, n.j, SUM(m.v*n.v)) from m inner join n on m.j=n.i group by m.i, n.j
-- hadamard multiplication
select m.i, m.j, m.v*n.v from m inner join n on m.i=n.i and m.j=n.j
-- sigmoid function
select i, j,  1/(1+exp(-v)) from m;
-- transpose
select i as j, j as i, v from m;
```

List. 3: Building blocks for matrices in SQL-92.


Feature Matrix
One-Hot-Encoded

row	sepal length	s. width	petal length	p. width	species	i	j	v	i	j	v
1	5.1	3.5	1.4	0.2	0	1	1	1	1	1	5.1
...	...	...	...	...	...	1	2	0	1	2	3.5
...	...	...	...	...	...	1	3	0	1	3	1.4
...	...	...	...	...	...	...	...	...	1	4	0.2
150	5.9	3.0	5.1	1.8	2	150	3	1	...	...	...

Fig. 3: Transformation of the original data set into the relational representation.

To train the neural network in SQL, we first have to convert the data into the relational representation (List. 4). 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 5-8) 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 11) and a matrix shape—defined by all indices within the dimensions—out of `null` values (lines 12-14). Then, an outer join (lines 11/15) combines both tables and assigns zero to missing values (`coalesce`: line 10).

```
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 if not exists img (i int, j int, v float);
4  create table if not exists one_hot(i int, j int, v int);
5  insert into img (select id,1,sepal_length/10 from iris);
6  insert into img (select id,2,sepal_width/10 from iris);
7  insert into img (select id,3,petal_length/10 from iris);
8  insert into img (select id,4,petal_width/10 from iris);
9  insert into one_hot(
10    select n.i, n.j, coalesce(i.v,0), i.v
11    from (select id,species+1 as species,1 as v from iris) i right outer join
12         (select a.a as i, b.b as j from
13            (select generate_series as a from generate_series(1,select count(*) from iris)) a,
14            (select generate_series as b from generate_series(1,4)) b
15         ) n on n.i=i.id and n.j=i.species order by i,j);
```

List. 4: Data transformation: Feature matrix `img` and one-hot-encoded label `one_hot`.

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

```
create table if not exists w_xh (i int, j int, v float);
create table if not exists w_ho (i int, j int, v float);
insert into w_xh (select i.*,j.*,random()*2-1 from generate_series(1,4) i, generate_series(1,20) j);
insert into w_ho (select i.*,j.*,random()*2-1 from generate_series(1,20) i, generate_series(1,3) j);
```

List. 5: Create and initialise weights in SQL-92.

The feature matrix in relational representation forms the input for training the neural network within a recursive CTE (List. 6) 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 6-15), to backpropagate the loss (lines 16-29) and to compute the derivative per weight matrix (lines 30-37). The first CTE `w_now`—just referring to the original weights—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. Section 2.1). Finally, the weights were updated by subtracting their derivatives (lines 39-41).

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

List. 6: Training a neural network in SQL-92.

In order to predict the accuracy of the trained weights, an SQL query measures the number of correctly classified labels (List. 7). Evaluating the model (lines 3-9) 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 11) to compare whether the index of the highest probability matches the index of the one value (line 14). Although window functions were used for the ranking, they could be replaced by an anti-join using `not exists` to conform SQL-92.

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

List. 7: Prediction in SQL:2003 (with window functions).

3 Evaluation

System: Ubuntu 22.04 LTS, 20 Intel Xeon E5-2660 v2 CPU with hyper-threading, running at 2.20 GHz with 256 GB DDR4 RAM.

We compare the performance of the relational representation for matrices (*SQL-92*, List. 6) to their representation as an array data type [Sc21c] (*SQL + Arrays*). We apply both representations for use within neural networks in SQL and let the benchmarks[4] run in Umbra [NF20] and PostgreSQL (PSQL) 14.5 [SR86] as target engines. The implementation with NumPy (List. 1) serves as the baseline. We use two different data sets: Fisher's Iris flower data [Fi36] (four attributes, one label) and the MNIST data [CMS12] for image classification (ten categories, 784 pixels).

3.1 Scaling the Number of Input Tuples

Figure 4 shows the first benchmark on the Iris data set. As we are interested in the performance numbers and not in the model quality, we replicate the Iris flower data set for the first benchmark to enable a flexible input size. A neural network with one hidden

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

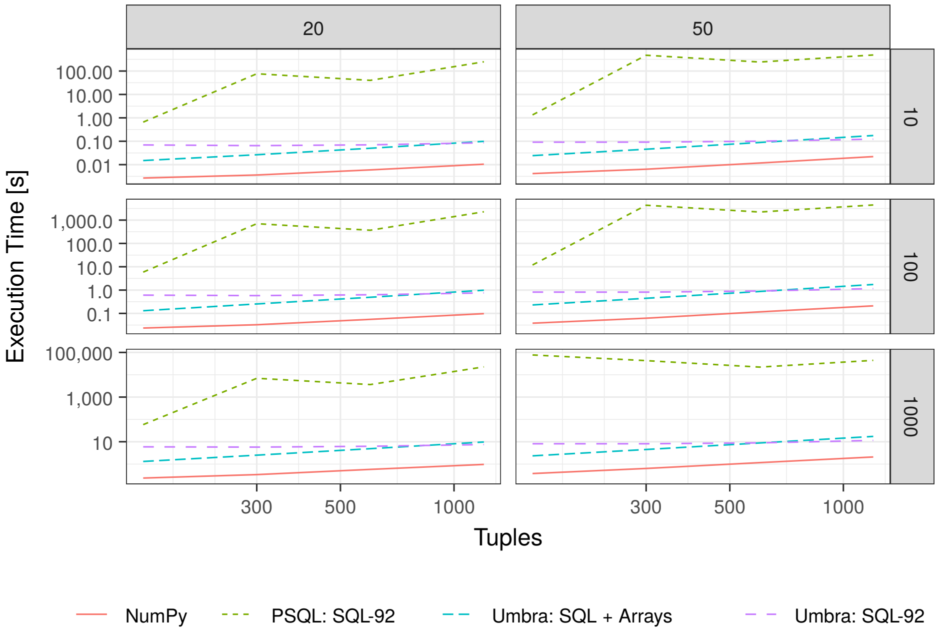


Fig. 4: Runtime for training a neural network with one hidden layer (size 20/50, 10/100/1000 iteration).

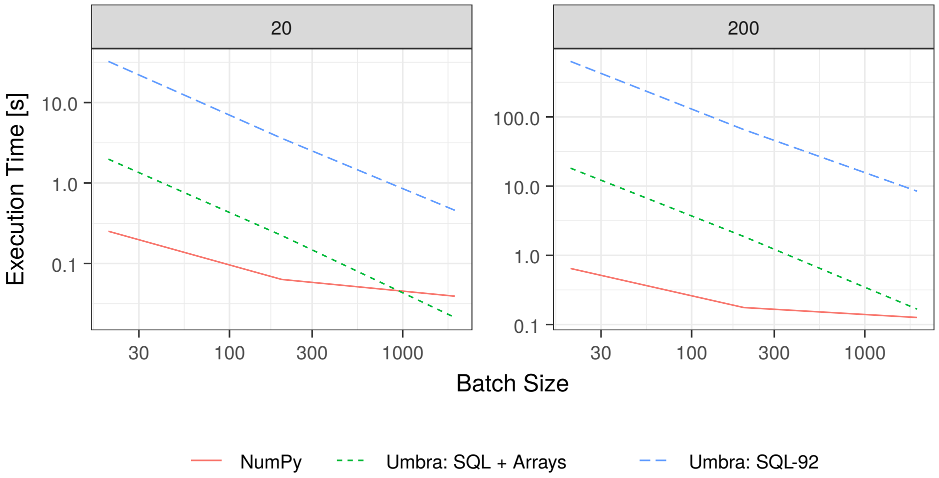


Fig. 5: Runtime for training one epoch with the MNIST data set with increasing batch size (one hidden layer size 20/200).

layer is trained to classify the flower category. We vary the size of the training data set, the number of iterations and the size of the hidden layer. Although the NumPy implementation outperforms both SQL variants, the performance increase of Umbra with its in-memory performance in comparison to PostgreSQL is visible. With only four attributes, the overhead for array operations dominates, so the relational representation performs better than the array data type for larger input data. Both SQL variants 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.

3.2 Image Classification

The second benchmark simulates image classification based on the MNIST data set using a neural network with one hidden layer. We measure the runtime for training one epoch depending on the batch size. As we can see in Figure 5, database systems perform better the bigger the batch size is. With a larger batch size, the runtime of the SQL implementations approximates the one of the baseline implementation. As the MNIST data set contains more attributes than the latter, the cost for aggregation into arrays is amortised and the SQL array data type outperforms the relational representation. To conclude, in-memory database systems are able to carry out matrix operations as required for neural networks. Nevertheless, use-case-specific optimisations are needed to support smaller batch sizes.

4 Conclusion

This 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 first 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. In the evaluation, in-memory enhanced database systems, i.e. Umbra, showed comparable performance to state-of-the-art libraries used in machine learning, i.e. NumPy in Python, when training with larger batch sizes only.

Future research is required on optimising recursive CTEs for this use case and on automatically generating the presented queries. As we are using recursion to imitate a procedural loop, the recursive CTE grows with each iteration. Therefore, the memory consumption increases per iteration, which restricts the number of iterations and the model size. To overcome the restrictions, database optimisers should either detect subsequent selections to eliminate intermediate results within the CTE or output intermediate results to free memory. Assuming these optimisations, one can use the presented queries to train more complex models with more weight variables.

References

[Bl22] Blacher, M.; Giesen, J.; Laue, S.; Klaus, J.; Leis, V.: Machine Learning, Linear Algebra, and More: Is SQL All You Need? In: CIDR. www.cidrdb.org, 2022.

[Bu20] Butterstein, D.; Martin, D.; Stolze, K.; Beier, F.; Zhong, J.; Wang, L.: Replication at the Speed of Change - a Fast, Scalable Replication Solution for Near Real-Time HTAP Processing. Proc. VLDB Endow. 13/12, pp. 3245–3257, 2020.

[Bu22] Budach, L.; Feuerpfeil, M.; Ihde, N.; Nathansen, A.; Noack, N. S.; Patzlaff, H.; Harmouch, H.; Naumann, F.: The Effects of Data Quality on ML-Model Performance. CoRR abs/2207.14529/, 2022.

[CMS12] Ciresan, D. C.; Meier, U.; Schmidhuber, J.: Multi-column deep neural networks for image classification. In: CVPR. IEEE Computer Society, pp. 3642–3649, 2012.

[DMG22] Dietrich, B.; Müller, T.; Grust, T.: Data provenance for recursive SQL queries. In: TaPP. ACM, 9:1–9:8, 2022.

[Fi36] Fisher, R. A.: The use of multiple measurements in taxonomic problems. Annals of eugenics 7/2, pp. 179–188, 1936.

[He22] Heinrich, R.; Luthra, M.; Kornmayer, H.; Binnig, C.: Zero-shot cost models for distributed stream processing. In: DEBS. ACM, pp. 85–90, 2022.

[Ma15] May, N.; Lehner, W.; P., S. H.; Maheshwari, N.; Müller, C.; Chowdhuri, S.; Goel, A. K.: SAP HANA - From Relational OLAP Database to Big Data Infrastructure. In: EDBT. OpenProceedings.org, pp. 581–592, 2015.

[MAF21] Miedema, D.; Aivaloglou, E.; Fletcher, G.: Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study. In: ICER. ACM, pp. 355–367, 2021.

[MD22] Maltry, M.; Dittrich, J.: A Critical Analysis of Recursive Model Indexes. Proc. VLDB Endow. 15/5, pp. 1079–1091, 2022.

[Mu17] Murray, I.: Machine Learning and Pattern Recognition (MLPR): Backpropagation of Derivatives, 2017, URL: `https://www.inf.ed.ac.uk/teaching/courses/mlpr/2017/notes/w5a_backprop.pdf`, visited on: 09/02/2021.

[Na22] Nath, R. P. D.; Romero, O.; Pedersen, T. B.; Hose, K.: High-level ETL for semantic data warehouses. Semantic Web 13/1, pp. 85–132, 2022.

[NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: CIDR. www.cidrdb.org, 2020.

[OVZ22] Olteanu, D.; Vortmeier, N.; Zivanovic, D.: Givens QR Decomposition over Relational Databases. In: SIGMOD Conference. ACM, pp. 1948–1961, 2022.

[Ra18] Raasveldt, M.; Holanda, P.; Mühleisen, H.; Manegold, S.: Deep Integration of Machine Learning Into Column Stores. In: EDBT. OpenProceedings.org, pp. 473–476, 2018.

[Re22] Renz-Wieland, A.; Gemulla, R.; Kaoudi, Z.; Markl, V.: NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. In: SIGMOD Conference. ACM, pp. 481–495, 2022.

[Sa22] Salimzadeh, S.; Gadiraju, U.; Hauff, C.; van Deursen, A.: Exploring the Feasibility of Crowd-Powered Decomposition of Complex User Questions in Text-to-SQL Tasks. In: HT. ACM, pp. 154–165, 2022.

[Sc19] Schüle, M. E.; Passing, L.; Kemper, A.; Neumann, T.: Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme. In: BTW. Vol. P-289. LNI, Gesellschaft für Informatik, Bonn, pp. 107–126, 2019.

[Sc21a] Schuhknecht, F. M.; Priesterroth, A.; Henneberg, J.; Salkhordeh, R.: AnyOLAP: Analytical Processing of Arbitrary Data-Intensive Applications without ETL. Proc. VLDB Endow. 14/12, pp. 2823–2826, 2021.

[Sc21b] Schüle, M. E.; Götz, T.; Kemper, A.; Neumann, T.: ArrayQL for Linear Algebra within Umbra. In: SSDBM. ACM, pp. 193–196, 2021.

[Sc21c] Schüle, M. E.; Lang, H.; Springer, M.; Kemper, A.; Neumann, T.; Günnemann, S.: In-Database Machine Learning with SQL on GPUs. In: SSDBM. ACM, pp. 25–36, 2021.

[Sc21d] Schüle, M. E.; Schmeißer, J.; Blum, T.; Kemper, A.; Neumann, T.: TardisDB: Extending SQL to Support Versioning. In: SIGMOD Conference. ACM, pp. 2775–2778, 2021.

[Sc22] Schüle, M. E.; Götz, T.; Kemper, A.; Neumann, T.: ArrayQL Integration into Code-Generating Database Systems. In: EDBT. OpenProceedings.org, 1:40–1:51, 2022.

[SK22] Störl, U.; Klettke, M.: Darwin: A Data Platform for Schema Evolution Management and Data Migration. In: EDBT/ICDT Workshops. Vol. 3135. CEUR Workshop Proceedings, CEUR-WS.org, 2022.

[SR86] Stonebraker, M.; Rowe, L. A.: The Design of Postgres. In: SIGMOD Conference. ACM Press, pp. 340–355, 1986.

[WGR20] Wingerath, W.; Gessert, F.; Ritter, N.: InvaliDB: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases (Extended). Proc. VLDB Endow. 13/12, pp. 3032–3045, 2020.

[WH21] Wiese, L.; Höltje, D.: NNCompare: a framework for dataset selection, data augmentation and comparison of different neural networks for medical image analysis. In: DEEM@SIGMOD. ACM, 6:1–6:7, 2021.

[WP22] Wenig, P.; Papenbrock, T.: DataGossip: A Data Exchange Extension for Distributed Machine Learning Algorithms. In: EDBT. OpenProceedings.org, 2:373–2:377, 2022.