Theoretische Informatik II

Searching for fast matrix multiplication algorithms

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

written by

Till Späth
tillspaeth@web.de
171085

supervised by

Professor Joachim Giesen

October, 2019

Abstract

The subject of this master's thesis is an example of automated learning of algorithms. Bilinear algorithms with fewer than n^3 products for the multiplication of $n \times n$ matrices are searched for. We are using a combination of the backpropagation algorithm and projection algorithms, the Difference-Map Algorithm and, alternatively, another heuristic algorithm. Bilinear matrix multiplication algorithms can be represented in network form, which motivates the use of the backpropagation algorithm. The solutions found by training the network contain much more than n^3 products, the set of solutions needs to be constraint to integer solutions only containing $-1, 0$ or 1 for the weights of the network. Parameters for the algorithms are determined through tests. For better performance of the search, an initialization step is developed. For reducing also additions and subtractions in found matrix multiplication algorithms, another algorithm that exploits common expressions is developed. The combination of the backpropagation algorithm and the Difference-Map Algorithm is tested for the cases $n = 2$, $n = 3$ and $n = 5$. Several thousand solutions are found for the case $n = 3$, among these solutions, the one with the least number of additions and subtractions is selected and shortly discussed. The behaviour of the Difference-Map Algorithm applied to our problem is investigated. The found solutions can be downloaded from GitHub.

Zusammenfassung

Thema dieser Masterarbeit ist ein Beispiel für das automatische Lernen von Algorithmen. Wir suchen nach bilinearen Algorithmen mit weniger als n^3 elementaren Multiplikationen für die Multiplikation von $n \times n$ Matrizen. Wir verwenden eine Kombination aus dem Backpropagation-Algorithmus und Projektionsalgorithmen, dem Difference-Map-Algorithmus und alternativ einem heuristischen Algorithmus. Bilineare Matritzenmultiplikationsalgorithmen können in Netzwerkform dargestellt werden, dies motiviert die Verwendung des Backpropagation-Algorithmus. Die Lösungen, die durch Trainieren des Netzwerks gefunden werden, enthalten viel mehr als n^3 elementare Multiplikationen. Die Menge der Lösungen muss auf ganzzahlige Lösungen beschränkt werden, die nur -1, 0 und 1 als Gewichte des Netzwerks enthalten. Parameter für die Algorithmen werden durch Tests ermittelt. Um die Geschwindigkeit der Suche zu verbessern, wird ein Initialisierungsschritt entwickelt. Um auch die Anzahl der Additionen und Subtraktionen in den gefundenen Matrizenmultiplikations-Algorithmen zu reduzieren, wird ein weiterer Algorithmus entwickelt, der gemeinsame Ausdrücke ausnützt. Die Kombination aus Backpropagation-Algorithmus und Difference-Map-Algorithmus wird für die Fälle $n = 2$, $n = 3$ und $n = 5$ getestet. Mehrere Tausend Lösungen werden für den Fall $n = 3$ gefunden. Unter diesen Lösungen wird die mit der geringsten Anzahl an Additionen und Subtraktionen ausgewählt und kurz diskutiert. Das Verhalten des Difference-Map-Algorithmus, angewendet auf unser Problem, wird untersucht. Die gefundenen Lösungen sind auf GitHub verfügbar.

Contents

Introduction

While the complexity of the ‘standard’ matrix multiplication algorithm for $n \times n$ matrices is $O(n^3)$, algorithms with lower complexity are known. Two examples are:

- Strassen, 1969, $O(n^{2.8074})$ [1]. This algorithm can be used with matrices of realistic size.
- Coppersmith–Winograd, 1990, $O(n^{2.375477})$ [2]. This algorithm possibly only has an advantage for very large matrices (‘galactic algorithm’).

This masters thesis, in Computational & Data Science, is about the automatic search for algorithms that calculate the matrix-matrix product $C = A \cdot B$ for quadratic matrices. We want to automate the search for fast matrix multiplication algorithms in bilinear form. Bilinear form here means, that the algorithm contains products of entries of A with entries of B, but never products of two entries of A or two entries of B.

The fast matrix multiplication algorithms discussed in this document require less multiplications but more additions (subtractions), compared to the standard algorithm. Modern CPUs (with instruction pipelines) require similar numbers of cycles for additions and multiplications, so it is questionable if the number of multiplications alone is the right measure to decide if a matrix multiplication algorithm is ‘fast’. Anyway, the problem to find fast matrix multiplication algorithms is still interesting as an example of automatic learning of algorithms, which is still a less advanced field of machine learning.

To find algorithms with less than n^3 multiplications, we are especially interested in a subset of all correct bilinear algorithms, the subset that consists of matrices filled only with $\{-1, 0, 1\}$. This restriction is the main problem to solve. After reducing the number of products in our algorithms, we also reduce the number of additions and subtractions of our algorithms.

The search-algorithms developed here are mainly tested for the cases $n = 2$ and $n = 3$. A few solutions for $n = 5$ are computed to further confirm the functionality of our solution. The effort for the automated search is growing very quickly with increasing n. But like the original Strassen Algorithm, the algorithms found in our search can be applied recursively.

Many solutions for the case $n = 3$ are found during the development of our search algorithms, but only the one solution with the lowest number of operations is briefly discussed in the end.

I want to thank Professor Joachim Giesen for the time he took to supervise me and for inspiration, tips and constructive critisism from him, concerning the content as well as the presentation of my masters thesis.

1 Literature on the topic

Here, we give a brief overview on some literature on the topic of bilinear algorithms for matrix multiplication.

Gaussian elimination is not optimal, Strassen [1]: This is the paper that started the race for matrix multiplication algorithms with lower numbers of multiplications involved, compared to the standard algorithm. We take the original ‘Strassen Algorithm’ from here. The paper has only three pages, so maybe it is a good start to read this paper, to understand what we are searching for.

A network that learns Strassen multiplication, Elser [3]: We take the idea, to model bilinear algorithms as a network from this paper. The reader finds a solution for the non-integer problem based on ‘conservative learning’ and a Lagrangian function in this paper.

Fast matrix multiplication, Bläser [4]: Here the reader finds theoretical background on computation and on bilinear algorithms. For example, the ‘Ostrowski Measure’ is described here. The tensor perspective on the bilinear algorithms is found here. Also, various other approaches for fast matrix multiplication algorithms are discussed.

Lectures on the complexity of bilinear problems, Groote [5]: This book is based on a lecture. It covers, for example, the theory behind bilinear mappings, complexity, rank, and the multiplication of large matrices.

Searching with iterated maps, Elser [6]: We take the algorithm to find the integer solutions to our problem from here. The algorithm could be interesting for many other problems where we search for the intersection of two constraint sets.

A noncommutative algorithm for multiplying 3$\times$3 matrices using 23 multiplications, Laderman [7]: In this paper from 1976, an algorithm for the multiplication of 3x3 matrices is introduced. This algorithm was found without using computers and has the same form as our algorithms.

A new general-purpose method to multiply 3$\times$3 matrices using only 23 multiplications, Curtois [8]: The authors of this paper find a solution of the same kind we are looking for. They use SAT solvers to find their solution. Also, they discuss the subject of equivalency of solutions.

A non-commutative algorithm for multiplying 5×5 matrices using 99 multiplications, Sedoglavic [9]: The author presents an algorithm for multiplying 5×5 matrices using 99 multiplications. The author develops his algorithm starting from Makarov's algorithm that uses 100 multiplications. Makarov's algorithm is developed by using Strassen-like algorithms for smaller matrices in an 'divide and conquer' approach.

2 Matrix multiplication algorithms

In this chapter, we describe what we mean when we write about matrix multiplication algorithms. We develop the special form of the algorithms we are searching for. Also, we give some properties of the algorithms that could be useful if someone is looking for alternative search methods or for extensions to our method.

In our discussion of the matrix multiplication algorithms, the main focus is the number of operations, especially the number of multiplications, involved in the execution of the algorithms. We use the convention, that multiplications with $-1, 0$ and 1 are not counted as operations, additions and subtractions of 0 and negations are also not counted.

Equations alone are not sufficient to describe algorithms. Often, when seeing an equation, we have an intuition about how to execute the computations involved for the evaluation of the equation. Sometimes only one possible way of computation might exist. However, in general, an equation does not dictate the computation used for the evaluation. If the equation, as a statement, is correct, every (correct) computation of the left and the right side of the equation has to be accepted. That is why, we also give algorithms in pseudo code to tell, which computations are meant.

For this document, we only want to look at the general case with fully populated matrices. Of course, things can look very different for sparse matrices.

2.1 The standard matrix multiplication algorithm

We start the development of the algorithms we are looking for with the 'standard' matrix multiplication algorithm. The 'standard' matrix multiplication algorithm for square matrices has multiplicative complexity $O(n^3)$. It is based on the equation:

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \cdot B_{k,j}, \qquad n \in \mathbb{N}, \quad A, B, C \in \mathbb{R}^{n \times n} \tag{2.1}$$

The right side of Eq. 2.1 is evaluated for every (i, j) separately, without using any synergies between the n^2 equations. So, for every (i, j), n multiplications are executed.

```
Data: $A, B$
Result: $C$
for $i$=1 .. $n$ do
    for $j$=1 .. $n$ do
        $C_{i,j} := 0$;
        for $k$=1 .. $n$ do
            $C_{i,j} := C_{i,j} + A_{i,k} \cdot B_{k,j}$; // counted as product
        end
    end
end
```

Algorithm 1: *Standard matrix multiplication algorithm with n^3 products*

In general, this algorithm uses n^3 multiplications between the entries of A and the entries of B to compute C. If we have sparse matrices that (implicitly) tell us which multiplications not to execute, we can of course be faster than n^3, but then Alg. 1 alone is already not sufficient.

Please note, that Eq. 2.1, is bilinear (products of elements of A with elements of A do not occur, the same holds for B). For this reason, we call Alg. 1 a 'bilinear algorithm'.

2.2 Bilinear matrix multiplication algorithms

For the further usage of the matrices A, B and C in this document, it is more practical to flatten the $n \times n$ matrices into vectors of size $N = n^2$. And to use $a_{i \cdot n+j} := a_{i,j}$, $b_{i \cdot n+j} := b_{i,j}$, $c_{i \cdot n+j} := c_{i,j}$ for the flattened matrices. For the matrix A this looks like:

$$A = \begin{bmatrix} a_1 = a_{1,1} & a_2 = a_{1,2} & ... & a_n = a_{1,n} \\ a_{n+1} = a_{2,1} & a_{n+2} = a_{2,2} & ... & a_{2n} = a_{2,n} \\ ... & ... & ... & ... \\ a_{(n-1)n+1} = a_{n,1} & a_{(n-1)n+2} = a_{n,2} & ... & a_{n^2} = a_{n,n} \end{bmatrix}, \quad a = [a_1, ..., a_{n^2}]^T$$

To adapt the matrix multiplication Alg. 1 for the flattened matrices, we then need to know which element of a has to be multiplied with which element of b to compute c_k. The information about which products between elements of a and elements of b are needed to compute c_k is stored in matrices U_k and V_k (which are slices of tensors U and V). j is the index for the n products needed. We can also simply write $mat(c) = mat(a) \cdot mat(b)$, but that does not tell us much about how to compute the matrix product. We use the following equation to build the algorithm for the flattened matrices:

$$c_k = \sum_{j=1}^{n} (\sum_{i=1}^{N} U_{k,j,i} a_i)(\sum_{i=1}^{N} V_{k,j,i} b_i), \qquad U, V \in \{0,1\}^{N \times n \times N} \tag{2.2}$$

```
Data: $a, b$
Result: $c$
for $k$=1 .. $N$ do
    $c_k := 0$;
    for $j$=1 .. $n$ do
        $i_1 :=$ find the index of the one non-zero entry of $U_{k,j}$;
        $i_2 :=$ find the index of the one non-zero entry of $V_{k,j}$;
        $c_k := c_k + a_{i_1} \cdot b_{i_2}$; // counted as product
    end
end
```

Algorithm 2: *Matrix multiplication algorithm for the flattened matrices*

Of course, Alg. 2 is only a matrix-multiplication algorithm if U and V are set in the right way.

How, for example, can we set $U_{k=1}$ and $V_{k=1}$ to compute $c_{1,1} = c_1$ in the case $n = 2$? Let:

$$A = \begin{bmatrix} a_1 = a_{1,1} & a_2 = a_{1,2} \\ a_3 = a_{2,1} & a_4 = a_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} b_1 = b_{1,1} & b_2 = b_{1,2} \\ b_3 = b_{2,1} & b_4 = b_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} c_1 = c_{1,1} & c_2 = c_{1,2} \\ c_3 = c_{2,1} & c_4 = c_{2,2} \end{bmatrix}.$$

If we flatten the matrices, then we get:

$$a = [a_1, a_2, a_3, a_4]^T, \quad b = [b_1, b_2, b_3, b_4]^T, \quad c = [c_1, c_2, c_3, c_4]^T.$$

If we set:

$$U_{k=1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad V_{k=1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

then we have a pair of matrices $U_{k=1}$, $V_{k=1}$ to compute $c_{1,1} = c_1$.

For the standard Algorithm 1, to compute one c_k, n multiplications are needed. Every matrix U_k (V_k) will have exactly n entries of 1, all other entries are 0. If we understand j to be the row-index of U_k (V_k), the matrices have n rows for the n products. In every row exactly one position is 1.

So far, we have a way to write Eq. 2.1 in the more complicated way Eq. 2.2, and Alg. 1 in the more complicated way Alg. 2, given the right definitions of U and V.

We do not want to work with the three dimensional tensors $U_{k,j,i}, V_{k,j,i}$ but stay with matrices. We want to split up the information about which products to compute, and the information about where to use those products. So, next we want to use the following equation and the corresponding algorithm:

$$c_k = \sum_{p=1}^{n^3} \tilde{W}_{k,p}(\sum_{i=1}^{N} \tilde{U}_{p,i}a_i)(\sum_{i=1}^{N} \tilde{V}_{p,i}b_i), \qquad \tilde{U}, \tilde{V} \in \{0,1\}^{n^3 \times N}, \tilde{W} \in \{0,1\}^{N \times n^3} \tag{2.3}$$

```
Data: $a, b$
Result: $c$
for $k$=1 .. $N$ do
    $c_k := 0$;
    for $\{p \in [1..n^3] | \tilde{W}_{k,p} \neq 0\}$ do
        $i_1 :=$ find the index of the one non-zero entry of $U_p$;
        $i_2 :=$ find the index of the one non-zero entry of $V_p$;
        $c_k := c_k + a_{i_1} \cdot b_{i_2}$; // counted as product
    end
end
```

Algorithm 3: *Algorithm with matrices $\tilde{U}, \tilde{V}, \tilde{W}$, still n^3 products used*

In the standard algorithm we need n^3 products, hence for now we use $p \in [1, .., n^3]$. The c_k get assembled from the n^3 products, the information about which products to choose for c_k is stored in row-k of matrix $\tilde{W}$. The Information about how to produce the products is stored in the matrices $\tilde{U}, \tilde{V}$. For the calculation of c according to $mat(c) = mat(a) \cdot mat(b)$, we can use:

$$\tilde{U} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \tilde{V} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

to build all the products we need. Then we assemble the c_k with:

$$\tilde{W} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

So we have the same calculation as in Alg. 1, again.

We already see, that there is more than one possibility to correctly set the matrices $\tilde{U}, \tilde{V}, \tilde{W}$. Also we can already suspect, that we do not need n^3 products, if we set $\tilde{U}, \tilde{V}, \tilde{W}$ in a more sensible way. Until now we have only been using the products of single elements of a with single elements of b, now we also want to use the products of the sums. So far, we only use every product once for the calculation of c, now we want to reuse the products. Also, until now we were only using 0 and 1 as entries for the design matrices $\tilde{U}, \tilde{V}$ and $\tilde{W}$. To find better algorithms, we now also want to allow -1 as entry for the design matrices. Without this decision we will probably not be able to find more efficient algorithms. By allowing the use of -1, we are introducing the cheap negation operation and the subtraction, that is comparable in effort to the addition.

So now we want to look at algorithms of the form:

```
**Data:** $a, b$
**Result:** $c$
**for** $p$=1..$r$ **do**
    $q_1 := 0$;
    **for** $\{i \in [1..N] | \tilde{U}_{p,i} \neq 0\}$ **do**
        $q_1 := q_1 + \tilde{U}_{p,i} \cdot b_i$; // not counted as product
    **end**
    $q_2 := 0$;
    **for** $\{i \in [1..N] | \tilde{V}_{p,i} \neq 0\}$ **do**
        $q_2 := q_2 + \tilde{V}_{p,i} \cdot b_i$; // not counted as product
    **end**
    $m_p := q_1 \cdot q_2$; // counted as product
**end**
**for** $k$=1..$N$ **do**
    $c_k := 0$;
    **for** $\{p \in [1..r] | \tilde{W}_{k,p} \neq 0\}$ **do**
        $c_k := c_k + \tilde{W}_{k,p} \cdot m_p$;
    **end**
**end**
```

Algorithm 4: *The matrix multiplication algorithm we will use in the document, from now on. It uses r products.*

that correspond to the equation:

$$c_k = \sum_{p=1}^{r} \tilde{W}_{k,p} (\sum_{i=1}^{N} \tilde{U}_{p,i} a_i)(\sum_{i=1}^{N} \tilde{V}_{p,i} b_i), \quad \tilde{U}, \tilde{V} \in \{-1, 0, 1\}^{r \times N}, \quad \tilde{W}, \in \{-1, 0, 1\}^{N \times r}. \tag{2.4}$$

with $r < n^3$.

In the Strassen-Algorithm [1] for 2×2 matrix multiplication, first seven products m_j

of sums of elements of a and sums of elements of b are computed.

$$\begin{aligned}
m_1 &:= (a_1 + a_4) \cdot (b_1 + b_4) \\
m_2 &:= (a_3 + a_4) \cdot b_1 \\
m_3 &:= a_1 \cdot (b_2 - b_4) \\
m_4 &:= a_4 \cdot (b_3 - b_1) \\
m_5 &:= (a_1 + a_2) \cdot b_4 \\
m_6 &:= (a_3 - a_1) \cdot (b_1 + b_2) \\
m_7 &:= (a_2 - a_4) \cdot (b_3 + b_4)
\end{aligned}$$

The c_k are computed as sums of the m_j.

$$\begin{aligned}
c_1 &= m_1 + m_4 - m_5 + m_7 \\
c_2 &= m_3 + m_5 \\
c_3 &= m_2 + m_4 \\
c_4 &= m_1 - m_2 + m_3 + m_6
\end{aligned}$$

Now we set the matrices $\tilde{U}$, $\tilde{V}$ and $\tilde{W}$, to bring the Strassen Algorithm in form of Algorithm 4. Only $r = 7$ instead of $r = 2^3 = 8$ products are necessary. We set:

$$\tilde{U} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad \tilde{V} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix},$$

to have the products for the calculation of c. We then assemble c from the products by setting:

$$\tilde{W} = \begin{bmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix},$$

This way we compute c like in the Strassen-Algorithm.

The $\tilde{U}_p$ and $\tilde{V}_p$ in Eq. 2.4 are vectors of length N and the inner sums in Eq. 2.4 are actually inner products of those vectors with a and b. We can write:

$$a_p^* := \sum_{i=1}^{N} \tilde{U}_{p,i} a_i = \langle \tilde{U}_p, a \rangle$$

$$b_p^* := \sum_{i=1}^{N} \tilde{V}_{p,i} b_i = \langle \tilde{V}_p, b \rangle$$

If we combine the a_p^* in one vector and understand the $\tilde{U}_p$ as the rows of a matrix, we can write the inner summations as:

$$a^* = \tilde{U} \cdot a$$

$$b^* = \tilde{V} \cdot b$$

The products of the inner sums of Eq. 2.4 can then be written as a vector of the r products ($\odot$ is the element-wise product):

$$c^* := a^* \odot b^* \tag{2.5}$$

Each c_k is an inner product of row-k of matrix $\tilde{W}_{k,p}$ and c^*. So for the vector c we get:

$$c = \tilde{W} c^*$$

Now we can write down our equation for the matrix product as:

$$c = \tilde{W}(\tilde{U}a \odot \tilde{V}b) \tag{2.6}$$

From now on, we will call Eq. 2.6 an 'algorithm' at times, even though, it is actually an equation. But we imply the computation of the right side of the equation according to Alg. 4. Representation 2.6 is taken from Elser [3], where the algorithm is modeled as a network.

Eq. 2.6 is bilinear in a and b, since (shown here for a, the same can be done with b):

$$\tilde{W}(\tilde{U}(\alpha \cdot a) \odot \tilde{V}b) = \alpha \cdot \tilde{W}(\tilde{U}a \odot \tilde{V}b)$$

$$\tilde{W}(\tilde{U}(a_1 + a_2) \odot \tilde{V}b) = \tilde{W}(\tilde{U}a_1 \odot \tilde{V}b) + \tilde{W}(\tilde{U}a_2 \odot \tilde{V}b)$$

2.3 Properties of the bilinear matrix multiplication algorithms

We want to look at some properties of the algorithms and the design matrices. Even though we are not going to exploit all those properties, they might become useful if someone wants to find other solutions to the search problem.

The minimal number of element-wise products in Alg. 2.6 is called the 'bilinear complexity'. As 'effort' of relevance we take, for now, the number of multiplications $x \cdot y$ with $x \notin \{-1, 0, 1\}$ and $y \notin \{-1, 0, 1\}$. For Alg. 2.6 that means: effort $= r$ as long as $\tilde{W}, \tilde{U}, \tilde{V}$ only have entries from $\{-1, 0, 1\}$, which we assume so far. It also means: effort = 'bilinear complexity', if we assume the minimal r . Later we will also have a look at additions and subtractions.

Strassen [10] shows that, given the 'Ostrowski Measure', divisions do not help to design a better bilinear algorithm (with further conditions on the field).

Other, non-bilinear, algorithms are possible. Bläser [4] shows, again given the 'Ostrowski Measure', that the bilinear complexity is at it's worst twice the multiplicative complexity, the length of the optimal quadratic computation.

However, the use of the 'Ostrowski Measure' is questionable for the case, that not all entries of $\tilde{W}, \tilde{U}, \tilde{V}$ are from $\{-1, 0, 1\}$. The 'Ostrowski Measure' only counts multiplications between (sums of) elements of a and (sums of) elements of b (additions, subtractions and especially multiplications with scalars that do not belong to a or b are not counted). For Algorithm 2.6, only the r products in $a^* \odot b^*$ would be counted even for $\tilde{W}, \tilde{U}, \tilde{V}$ with entries from $\mathbb{R} \backslash \{-1, 0, 1\}$. So, if we took the 'Ostrowski Measure' to judge our algorithms, we could avoid the search for the integer solutions completely.

One important point regarding the matrices $\tilde{U}, \tilde{V}, \tilde{W}$ is, that once $\tilde{U}, \tilde{V}$ are found, $\tilde{W}$ is also determined (only one $\tilde{W}$ is possible for given $\tilde{U}, \tilde{V}$). This can also be seen through the following:

We want to show, that if we choose r linearly independent vectors $c_1, .., c_r$, we can create an equation to determine exactly one $\tilde{W}$, if the matrices $\tilde{U}, \tilde{V}$ are given. After choosing the r linearly independent c_i, we can easily choose $a_1, ..., a_r$ and $b_1, ..., b_r$, such that $mat(c_i) = mat(a_i) \cdot mat(b_i)$.

With the choosen c_i and the matching a_i, b_i, we can build the following equation from Eq. 2.6:

$$[c_1, .., c_r] = \tilde{W}(\tilde{U}[a_1, .., a_r] \odot \tilde{V}[b_1, .., b_r]). \tag{2.7}$$

Since all the c_i are linearly independent, the $c_i^* = Ua_i \odot Vb_i$ are also linearly independent. Otherwise, if we have a linearly dependent $c_d^* = \sum_{i \neq d} \alpha_i c_i^*,\ i, d \in [1, .., r]$, we can multiply with $\tilde{W}$ to get $c_d = \tilde{W} c_d^* = \sum_{i \neq d} \alpha_i W c_i^* = \sum_{i \neq d} \alpha_i c_i$. Which shows, that we must have all c_i^* independent because we choose independent c_i. By combining Eq. 2.7 and Eq. 2.5, we get:

$$[c_1, .., c_r] = \tilde{W}[c_1^*, .., c_r^*]. \tag{2.8}$$

$\tilde{W}$ contains $N \cdot r$ unknown values and Eq. 2.8 contains the same number of linearly independent (scalar) equations. That shows, that $\tilde{W}$ is determined by $\tilde{U}$ and $\tilde{V}$.

The rank of U and V is exactly the full rank N.
We compute $a^* = Ua$. If the rank of U was smaller than N, we would find at least one vector z in the kernel of U (other than the zero vector). For every triple (a,b,c) with $mat(c) = mat(a) \cdot mat(b)$ we could find a second triple (a+z,b,c) with $mat(c) = mat(a+z) \cdot mat(b)$, which is wrong in the general case. Hence, the rank of U is N.

For the calculation of $\tilde{W}c^*$ we need at least $r - N$ additions/subtractions.
In all of the r columns of $\tilde{W}$ at least one position has to be non-zero, otherwise there would be one product that we could eliminate, we could reduce r. This means we have minimum r non zero entries in $\tilde{W}$. But we only have N rows in $\tilde{W}$. So $r - N$ of the non-zeros have to be in rows that have at least one non-zero already. Every time we add one of the $r - N$ non-zeros to $\tilde{W}$, we have to count that as one addition or subtraction, because it is at least the second non-zero in the row.
We could think about reducing the number of additions/subtractions below $r - N$ by using common expressions. But then we have to consider, that all of the $r-N$ non zeros are in different columns, so even if we use the non-zeros in common expressions, we have to add/subtract any of the $r-N$ non-zeros at least once to build the common-expressions.
Example:

$$\tilde{W} = \begin{bmatrix} \square & 0 & 0 & 0 & \blacksquare & 0 & 0 \\ 0 & \square & 0 & 0 & 0 & 0 & \blacksquare \\ 0 & 0 & \square & 0 & 0 & \blacksquare & 0 \\ 0 & 0 & 0 & \square & 0 & 0 & 0 \end{bmatrix}$$

The $7 - 4 = 3$ $\blacksquare$-position all cause one operation that cannot be reduced through the use of common expressions.

2.4 The bilinear matrix multiplication algorithm as a network

The idea to use the network shown here as representation of Algorithm 2.6 is from Elser [3]. The network structure motivates using the backpropagation algorithm to find settings for the design matrices of the algorithms later.

The matrices W^a, W^b, W^c contain the weights of this network (multiplications with scalars in the arrows). W^a contains the weights between the a_k and the left sum-inputs

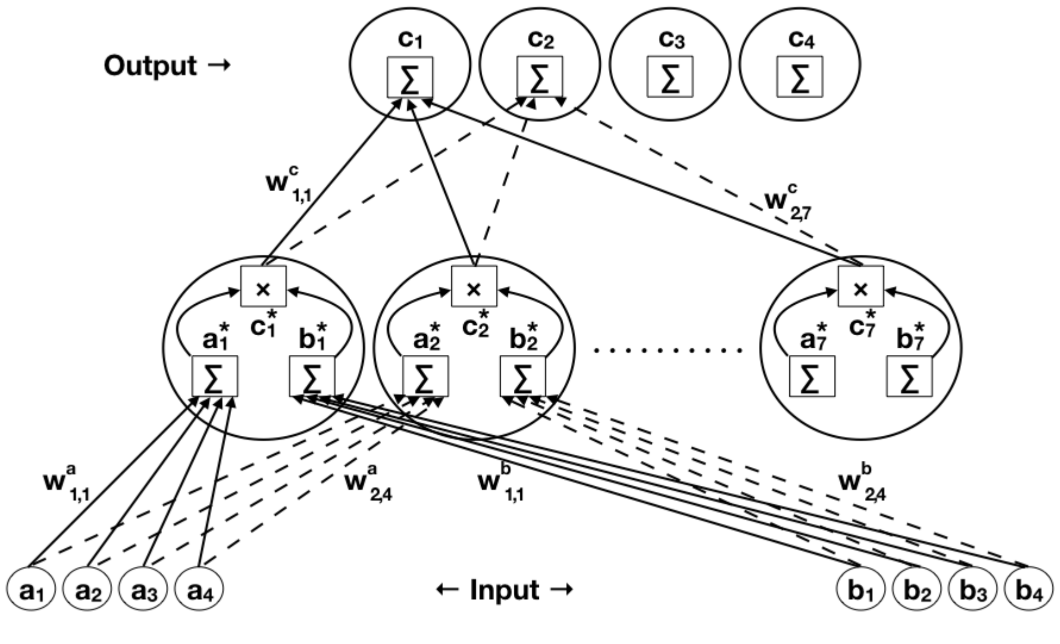

Figure 2.1: *Bilinear algorithm in network form ($n = 2$)*

of the middle-layer nodes. W^b contains the weights between the b_k and the right sum-inputs of the middle-layer nodes. The weights for the connections between the middle layer and the c_k are stored in W^c. So the network computes, Elser [3]:

$$c = W^c(W^a a \odot W^b b), \qquad W^a, W^b \in \mathbb{R}^{r \times N}, \quad W^c \in \mathbb{R}^{N \times r}$$

$$c = W^c c^*, \quad c^* := a^* \odot b^*, \quad a^* := W^a a, \quad b^* := W^b b \tag{2.9}$$

The symbols a^*, b^*, c^*, c are used if W^a, W^b, W^c are correct for the matrix multiplication algorithm. If the design matrices are not yet correct, $\tilde{a}^*, \tilde{b}^*, \tilde{c}^*, \tilde{c}$ are used. We do not use $\tilde{U}, \tilde{V}, \tilde{W}$ here, since the matrix entries are from $\mathbb{R}$, and not restricted to $\{-1, 0, 1\}$. Other than that, Eq. 2.9 is the same as Eq. 2.6

3 Backpropagation for the matrix multiplication network

The matrices W^a, W^b, W^c for Algorithm 2.9 have to be found somehow. Elser [3] uses an algorithm based on a Lagrange equation to solve this problem. Various other solutions are possible. Here the backpropagation algorithm is chosen because it is easy to implement and shows good results. The simple nature of the nodes of the network make it easy to adapt the backpropagation algorithm to our problem.

3.1 The backpropagation algorithm

To train the network 2.9 with the backpropagation algorithm, random inputs a, b are fed to the network in a 'forward step'. The output $\tilde{c}$ of the network is compared to the correct c (according to $mat(c) = mat(a) \cdot mat(b)$). The deviation between $\tilde{c}$ and c is then propagated back through the network to calculate corrections for the weights W^a, W^b, W^c. Remember Eq. 2.9: $c = W^c c^*,\ c^* = a^* \odot b^*,\ \ a^* = W^a a,\ \ b^* = W^b b.$

First, we draw a, b from an uniform distribution on the interval [-1,1]. a, b are scaled, so $|a|_2 = |b|_2 = 1$, this way all random inputs a, b should have comparable impact on the training result. Then we calculate the 'forward step' values for $\tilde{a}^*, \tilde{b}^*, \tilde{c}^*, \tilde{c}$ with Eq. 2.9 from the random input a, b, and also the correct c.

We calculate the correction Δ^c for W^c from the output-deviation e^c. Therefore we need the derivative $\frac{\partial e_i^c}{\partial W_{i,j}^c}$, the 'influence' of $W_{i,j}^c$ on the deviation e_i^c.

$$e_i^c := \frac{1}{2}(c_i - \tilde{c}_i)^2$$

$$\frac{\partial e_i^c}{\partial \tilde{c}_i} = \tilde{c}_i - c_i$$

$$\frac{\partial \tilde{c}_i}{\partial W_{i,j}^c} = \tilde{c}_j^*$$

$$\frac{\partial e_i^c}{\partial W_{i,j}^c} = \frac{\partial e_i^c}{\partial \tilde{c}_i} \cdot \frac{\partial \tilde{c}_i}{\partial W_{i,j}^c} = (\tilde{c}_i - c_i)\tilde{c}_j^*$$

$\frac{\partial e_i^c}{\partial W_{i,j}^c}$ is then fed back (negative feedback) with the learning rate η^c to W^c to make the deviation e^c go to zero. ($\otimes$ is the outer product.)

$$\Delta^c_{i,j} := -\eta^c \frac{\partial e^c_i}{\partial W^c_{i,j}}, \quad \eta^c \in \mathbb{R}_+$$

$$\Delta^c_{i,j} = -\eta^c (\tilde{c}_i - c_i)\tilde{c}^*_j$$

$$\Delta^c = -\eta^c (\tilde{c} - c) \otimes \tilde{c}^{*T} \tag{3.1}$$

Now, we calculate the correction Δ^a for W^a from the deviation e^{c*}. We want to calculate $\frac{\partial e^{c*}_i}{\partial W^a_{i,j}}$ first.

$$e^{c*} := \frac{1}{2}(c^*_i - \tilde{c}^*_i)^2$$

$$\frac{\partial e^{c*}_i}{\partial \tilde{c}^*_i} = \tilde{c}^*_i - c^*_i$$

$$\frac{\partial \tilde{c}^*_i}{\partial W^a_{i,j}} = a_j \tilde{b}^*_i$$

$$\frac{\partial e^{c*}_i}{\partial W^a_{i,j}} = \frac{\partial e^{c*}_i}{\partial \tilde{c}^*_i} \cdot \frac{\partial \tilde{c}^*_i}{\partial W^a_{i,j}} = (\tilde{c}^*_i - c^*_i) a_j \tilde{b}^*_i$$

Next, we can calculate Δ^a for the correction of W^a. We use η^{c*} as the learning rate.

$$\Delta^a_{i,j} := -\eta^{c*}(\tilde{c}^*_i - c^*_i) a_j \tilde{b}^*_i$$

$$\Delta^a = -\eta^{c*}((\tilde{c}^* - c^*) \odot \tilde{b}^*) \otimes a$$

We do not have the $\tilde{c}^* - c^*$, yet. If we consider $\tilde{c}^* - c^*$ to be the error in the middle layer output, this error will lead to the error $\tilde{c} - c = W^c(\tilde{c}^* - c^*)$ in the network output, if W^c is correct already (which is assumed for the next step). This means, that the error $\tilde{c}^* - c^*$ is amplified according to W^c to become the error $\tilde{c} - c$. It makes sense to distribute the error $\tilde{c} - c$ back to the error $\tilde{c}^* - c^*$ proportional to this amplification. This way, the correction of c^* stays small, because the values in c^* that have more influence on c (through W^c) also receive a stronger correction (through $\Delta^a a$).

$$\tilde{c}^* - c^* := W^{cT}(\tilde{c} - c)$$

And finally, with the same calculations for Δ^b:

$$\Delta^a = -\eta^{c*}((W^{cT}(\tilde{c} - c)) \odot \tilde{b}^*) \otimes a \tag{3.2}$$

$$\Delta^b = -\eta^{c*}((W^{cT}(\tilde{c} - c)) \odot \tilde{a}^*) \otimes b \tag{3.3}$$

The last step is to update the design matrices:

$$W^{a,updated} := W^a + \Delta^a \tag{3.4}$$

$$W^{b,updated} := W^b + \Delta^b \tag{3.5}$$

$$W^{c,updated} := W^c + \Delta^c \tag{3.6}$$

For further usage in the search for integral solutions for W^a, W^b, W^c, the backpropagation algorithm needs to be extended so entries of the design matrices can be marked to be alterable or not alterable in the iterations. To do this, the masks $M^a, M^b \in \{0,1\}^{r\times N}, M^c \in \{0,1\}^{N\times r}$ are used. An entry of 1 means 'alterable', an entry of 0 means 'not alterable'. Before the update of W^a, W^b, W^c, the matrices $\Delta^a, \Delta^b, \Delta^c$ are multiplied with the masks M^a, M^b, M^c. So Eq. 3.4, 3.5, 3.6 become:

$$W^{a,updated} := W^a + \Delta^a \odot M^a \tag{3.7}$$

$$W^{b,updated} := W^b + \Delta^b \odot M^b \tag{3.8}$$

$$W^{c,updated} := W^c + \Delta^c \odot M^c \tag{3.9}$$

We only need to change Eq. 3.4, 3.5, 3.6 to Eq. 3.7, 3.8, 3.9 in the basic backpropagation algorithm to get the masked-backpropagation algorithm.

The backpropagation algorithm for the training of the weigths for Algorithm 2.9 (with or without masks) is then:

```
Data: $W^a, W^b, W^c$
Result: updated $W^a, W^b, W^c$
repeat
    draw a,b from uniform distribution on interval [-1,1];
    scale a,b to $|a|_2 = |b|_2 = 1$;
    calculate $c$ from 2.1;
    calculate $\tilde{a}^*, \tilde{b}^*, \tilde{c}^*, \tilde{c}$ from 2.9;
    calculate $\Delta^a, \Delta^b, \Delta^c$ from 3.1,3.2,3.3;
    update $W^a, W^b, W^c$ with 3.4,3.5,3.6 (unmasked) or 3.7,3.8,3.9 (masked);
until $|\tilde{c} - c| <$ threshold for enough cycles;
```

Algorithm 5: *Backpropagation algorithm, adapted to our problem*

The algorithm does not converge for all starting values of W^a, W^b, W^c at all times. However, if the starting values are not too far from a solution, it will converge most times. If it does not converge after a certain number of cycles, it can be stopped and then restarted with new starting values, or even with the same starting values (since a, b are random values).

Often, one big and one small step down in the deviation can be found in successful runs of the algorithm. Figure 3.1 shows two runs of the backpropagation algorithm (without masking), where this behaviour can be found. On the y-axis $|\tilde{c} - c|_2$ is displayed. We find this observation interesting, but we are not sure enough about the causes of this behavior to give an explanation.

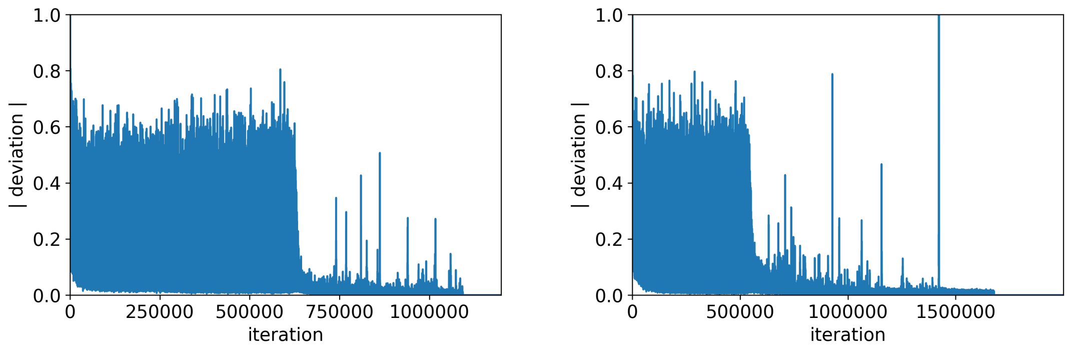

Figure 3.1: *Examples for the convergence of the backpropagation algorithm with one large and one small step down ($n = 3$)*

3.2 Parameters

To investigate the learning rates η^c and η^{c*}, we measure the average number of iterations needed to come to a correct solution, starting with a random initialization for W^a, W^b and W^c. We always stop at the latest after 25000 iterations in the case $n = 2$, and after 20000000 iterations in the case $n = 3$, even without a solution. The result can be seen in Figure 3.2, the red dots mark the minimum found.

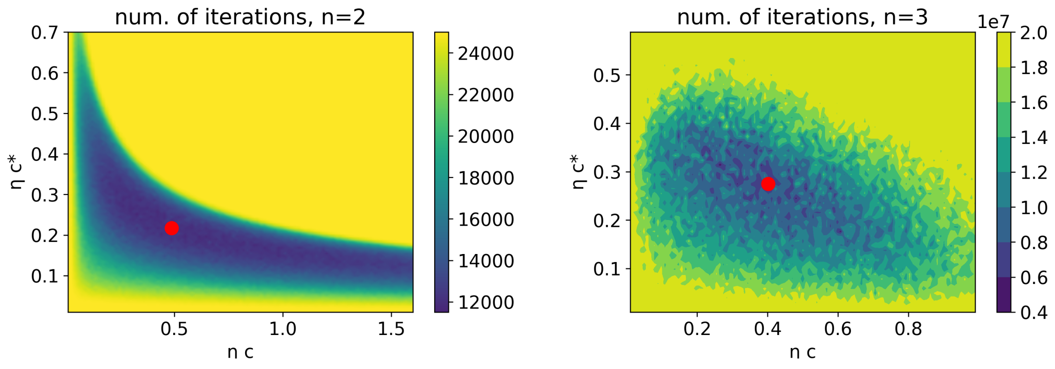

Figure 3.2: *Number of iterations and learning rates for the backpropagation algorithm, for the cases $n = 2$ and $n = 3$. The red dots mark the minimums found.*

The learning rates with the lowest average number of iterations are: $\eta^{c*} = 0.22$ and $\eta^c = 0.49$ for the case $n = 2$, $\eta^{c*} = 0.27$ and $\eta^c = 0.40$ for the case $n = 3$.

Later, we will embed the backpropagation algorithm into a meta algorithm. Doing this, we will experience, that the combination of meta algorithm and backpropagation algorithm only delivers solutions if we choose lower learning rates for the backpropagation algorithm. This has to do with Eq. 4.11, which will be explained later.

4 Projection algorithm for finding integer solutions

We are looking for algorithms with less than n^3 products, comparable to the Strassen Algorithm with seven multiplications instead of eight (for $n = 2$). However, the solutions we find with the backpropagation algorithm have much more than n^3 multiplications, because of the entries from $\mathbb{R}$ in the design matrices. We want to avoid all multiplications except for the r element-wise multiplications in $a^* \odot b^*$, if we assume the minimal r, we cannot avoid those element-wise multiplications. We want to find $W^a, W^b \in \{-1, 0, 1\}^{r \times N}$ and $W^c \in \{-1, 0, 1\}^{N \times r}$, so we do not have multiplications with the weights in the design matrices (remember: we do not count multiplications with $-1, 0, 1$).

We will stay with W^a, W^b and W^c instead of U, V and W, because the design matrices will not always be restricted to $-1, 0, 1$ during the run of the algorithm.

The Difference-Map algorithm is searching the intersection of two constraint sets, called A and B here. The convergence of the algorithm is more or less only an experimental observation, for the case that the two constraint sets are not convex. Nevertheless, other difficult problems were successfully solved with this approach, see Elser [6].

Our adaption of the Difference-Map Algorithm uses the main steps:

- backpropagation to find a solution in $\mathbb{R}$
- rounding to the next nearest value in $\{-1, 0, 1\}$,

$$round(x) := \operatorname*{argmin}_{y \in \{-1,0,1\}} |x - y| \tag{4.1}$$

The algorithm does not always find a solution after an acceptable number of cycles. However, if restarted after a maximum number of cycles, it reliably produces solutions.

4.1 The Difference-Map Algorithm

The basis for the Difference-Map Algorithm is taken from Elser [6]. It is based on projections on constraint sets that describe the problem. We can describe our problem as the search for design matrices that fulfill two conditions: They describe a matrix multiplication algorithm and they only contain entries from $\{-1, 0, 1\}$. If we find design matrices that fulfill both conditions, we have a solution.

So, here the two constraint sets are:

$$A := \{(W^a, W^b, W^c) : W^a, W^b \in \{-1, 0, 1\}^{r\times N}, W^c \in \{-1, 0, 1\}^{N\times r}\} \tag{4.2}$$

$$\begin{aligned} B := \{(W^a, W^b, W^c) : W^c(W^a a \odot W^b b) = vec(mat(a) \cdot mat(b))\ \forall a, b \in \mathbb{R}^N, \\ W^a, W^b \in \mathbb{R}^{r\times N}, W^c \in \mathbb{R}^{N\times r}\} \end{aligned} \tag{4.3}$$

A is the set of (W^a, W^b, W^c) that only contain entries from $\{-1, 0, 1\}$. B is the set of (W^a, W^b, W^c) that correctly describes the matrix-multiplication. If we find a triple $(W^a, W^b, W^c) \in A \cap B$, we have found a solution to our problem.

In the original algorithm Elser [6], we need two projections for the two constraint sets, that project to the nearest element in the constraint set. Here we choose the projections:

$$P_A(W^a, W^b, W^c) := round(W^a, W^b, W^c) \tag{4.4}$$

$$\begin{aligned} P_B(W^a, W^b, W^c) := \underset{(\tilde{W}^a, \tilde{W}^b, \tilde{W}^c)}{\text{argmin}}\ &\{|(\tilde{W}^a, \tilde{W}^b, \tilde{W}^c) - (W^a, W^b, W^c)|_* : \\ &\tilde{W}^a(\tilde{W}^b a \odot \tilde{W}^c b) = vec(mat(a) \cdot mat(b))\ \forall a, b \in \mathbb{R}^N, \\ &\tilde{W}^a, \tilde{W}^b \in \mathbb{R}^{r\times N}, \tilde{W}^c \in \mathbb{R}^{N\times r}\} \end{aligned} \tag{4.5}$$

Projection 4.4 is the element-wise rounding with Eq. 4.1 of all elements in the design matrices. Projection 4.5 is basically the task to find the correct design matrices $\tilde{W}^a, \tilde{W}^b, \tilde{W}^c$ closest to the not yet correct matrices W^a, W^b, W^c. $|(\tilde{W}^a, \tilde{W}^b, \tilde{W}^c) - (W^a, W^b, W^c)|_*$ could, for example, be the 2-norm for the flattened matrices (all combined to one vector).

From the projections we calculate $\Delta^W = (\Delta^a, \Delta^b, \Delta^c)$ for the update of $W = (W^a, W^b, W^c)$ (see the Difference-Map Algorithm in Elser [6]).

$$\Delta^W := \beta[P_A(f_B(W)) - P_B(f_A(W))] \tag{4.6}$$

$$f_A(W) := P_A(W) - \frac{P_A(W) - W}{\beta} \tag{4.7}$$

$$f_B(W) := P_B(W) + \frac{P_B(W) - W}{\beta} \tag{4.8}$$

W and Δ^W are triples of matrices. In Eq. 4.5-4.8 the operations $+, -$ are element-wise, $\cdot, /$ are multiplications (resp. divisions) of all elements with (resp. by) scalar. The choice of $\beta = 1$ seems to work well for the cases $n = 2$ and $n = 3$ (other values for β were also tried). $\beta = 1$ (similar for $\beta = -1$) has the advantage, that less projections per iteration are necessary. With $\beta = 1$ we get from Eq. 4.6-4.8:

$$\Delta^W := P_A(2P_B(W) - W) - P_B(W) \tag{4.9}$$

With Eq. 4.10 we update W as follows:

$$W^{updated} := W + \Delta^W. \tag{4.10}$$

Suppose, that we have already found a solution $W^{sol} \in A \cap B$, that means:

$$P_A(W^{sol}) = P_B(W^{sol}) = W^{sol}.$$

In Eq. 4.9 that leads to:

$$\begin{aligned} \Delta^W &= P_A(2P_B(W^{sol}) - W^{sol}) - P_B(W^{sol}) \\ &= P_A(2W^{sol} - W^{sol}) - W^{sol} \\ &= P_A(W^{sol}) - W^{sol} \\ &= W^{sol} - W^{sol} \\ &= 0. \end{aligned}$$

This shows at least, that once we have found a solution, we have $\Delta^W = 0$.

Between the updates, where do we need to look to eventually find a solution? We need to look at $P_B(W)$. By looking only at W we could miss a solution.

If $\Delta^W = 0$ we can have a look at the two cases $W \in A \cap B$ and $W \notin A \cap B$. In the first case, the iteration just ended in a stationary point that is a solution.

If we suppose $\Delta^W = 0$, $W \notin A \cap B$ we have: $P_B(W) = P_A(2P_B(W) - W)$. We know that $P_B(W) \in B$, and $P_A(2P_B(W) - W) \in A$. Because of the equality sign, also $P_B(W) \in A$. So, $P_B(W) \in A \cap B$ must be a solution. However, $W \notin A \cap B$ is not a solution in this case and $\Delta^W = 0$. That is, we iterated to a stationary point that is not a solution (W is not). That is not a problem if we always check $P_B(W)$ instead of W for validity for the case $\Delta^W = 0$ (we then automatically cover the case that W is a solution, because then $P_B(W) = W$).

We do not know how to implement P_B correctly with regard to the nearest point in B, so from now on we just use:

$$\begin{gathered} W_i := \textit{one run of backpropagation}(W), \quad i \in \{1, 2, .., t\} \\ P_B(W) := \underset{W_i}{\operatorname{argmin}} |W_i - W|_2 \end{gathered} \tag{4.11}$$

For a $t \in \mathbb{N}$. We hope, that the backpropagation algorithm will find solutions closer to the starting values with higher probability, and that the algorithm still works (which it does, solutions have been found). So far, $t = 3$ has worked well.

The $|\Delta^W|$ tends to stagnate in bands without further improvement for long times, as can be seen in Figure 4.1. This behavior is seen more frequently in the meta algorithm - the Difference-Map Algorithm - than in the underlying backpropagation algorithm. So for the Difference-Map Algorithm we want to take measures to improve the performance.

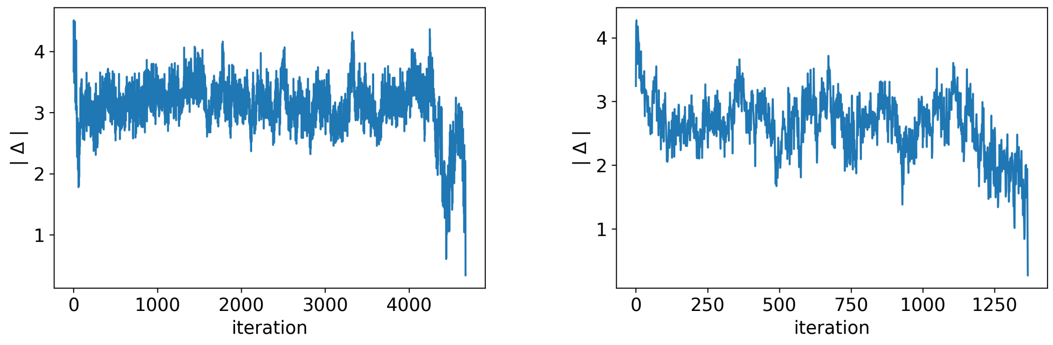

Figure 4.1: *Examples for the stagnation behaviour of the Difference-Map Algorithm ($n = 3$)*

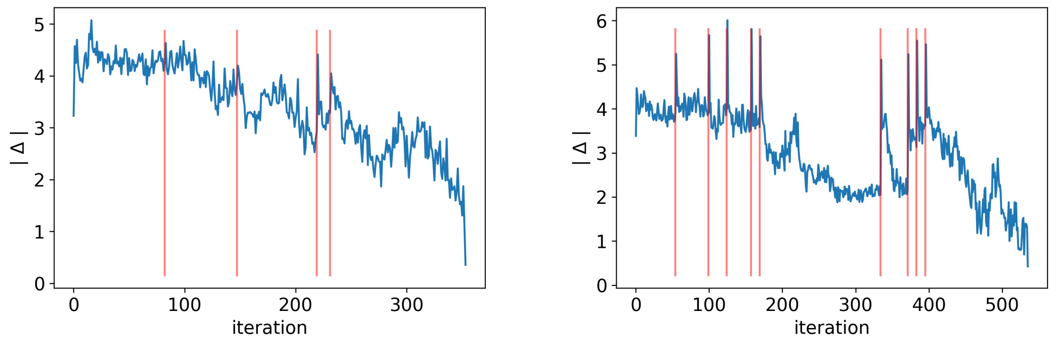

Figure 4.2: *Examples for the behaviour of the Difference-Map Algorithm with random values added ($n = 3$)*

The algorithm often spends most of the running time on those plateaus. That is why we need to identify this behavior and do something to shorten the running time. We check if Δ^W stays in some band for longer than a given number of iterations. If we find, that Δ^W stays between the minimum and the maximum of the last z iterations for more than q iterations, we try to escape by adding random values to W. We draw the random values from a uniform distribution on $[-1, 1]$ and multiply them with a 'jump factor'. Sometimes we need to add random values several times to get free. Through this measure, the running time is shortened significantly. Examples where we take this measure can be seen in Figure 4.2. The red lines mark the iterations where random values have been added. The total number of iterations is much lower.

Alg. 6 shows the algorithm in pseudo code.

```
Result: solution $W^{sol}$
repeat
    init $W$ with random values from uniform distribution on $[-1, 1]$;
    $bCount := 0$;
    $deltaHist :=$ empty list;
    repeat
        calculate $\Delta^W$ with 4.8;
        update $W$ with 4.9;
        if $\Delta^W$ is small then
            check if $P_B(W)$ is a correct solution;
            if $P_B(W)$ is correct then
                return $W^{sol} = P_B(W)$
            end
        end
        deltaHist.append($|\Delta^W|$);
        $bMax := max($last $z$ elements in $deltaHist)$;
        $bMin := min($last $z$ elements in $deltaHist)$;
        if $bMin < |\Delta^W| < bMax$ then
            $bCount := bCount + 1$;
        else
            $bCount := 0$;
        end
        if $bCount > q$ then
            draw $R$ from uniform distribution on $[-1, 1]$;
            $W := W + R \cdot jump\ factor$;
            $bCount := 0$;
        end
    until max. number of iterations reached;
until max. number of tries reached;
```

Algorithm 6: *Addaption of the Difference-Map Algorithm to our problem*

For the cases $n = 2$ and $n = 3$, the algorithm described above is capable of finding solutions. However, it is not very fast. Most initial values do not lead to a solution and the implementation is spending a lot of time on trying new starting values. As a rough idea, the implementation gives a new solution once per hour, run on four threads (four instances of the algorithm) on an Intel® Core^{TM} i7. However, with better initialization the performance is noticably improved.

One speciality of our implementation of the Difference-Map Algorithm is, that random numbers influence the run of the algorithm. The original algorithm, according to Eq. 4.2-4.10 does not contain random influences.

4.2 Improved initialisation for the Difference-Map Algorithm

To increase the performance of the projection algorithm, we want to improve the initialization. So far, the algorithm spends a lot of time trying different starting values, but most of the starting values do not lead to a solution. We improve the random starting values with a rounding step at the start of the algorithm.

We start with W from a uniform distribution, as before. We iterate $W = (W^a, W^b, W^c)$ to a solution in $\mathbb{R}$ with the backpropagation algorithm. Then we look at each weight to decide which one to round next based on doing minimal harm to the solution. We round the chosen weight, which introduces an error in our previously correct solution W. We fix the error with another step of backpropagation. For the backpropagation, we need to use the masked version, so once rounded, the values stay in $\{-1, 0, 1\}$. We repeat this procedure, until we have tried every entry of W. By this time, usually a good part of the weights have been successfully rounded. The improved W is already much closer to $A \cap B$. Experience shows, that now the probability of coming to a solution in $A \cap B$ has greatly increased.

We need a function $rank(i, j, m)$ to decide which of the weights to round next. For this function we choose the error the rounding would cause in the matrix multiplication if a and b were all set to 1 ($mat(\mathbf{1}) \cdot mat(\mathbf{1}) = n \cdot mat(\mathbf{1})$).

Here $\mathbf{1}$ is the vector of ones with length N. Basically, the difference between (W^a, W^b, W^c) and $(\hat{W}^a, \hat{W}^b, \hat{W}^c)$ is the one weight $\hat{W}^k_{i,j}$ that is rounded. All other weights are the same.

$$rank(i, j, m) := |\hat{W}^c(\hat{W}^a\mathbf{1} \odot \hat{W}^b\mathbf{1}) - n\mathbf{1}|_2 \tag{4.12}$$

$$\begin{gathered} m, k \in \{a, b, c\} \\ \hat{W}^k = W^k, \text{ if } m \neq k \\ \hat{W}^k_{r,s} = W^k_{r,s}, \text{ if } m = k \text{ and } (r, s) \neq (i, j) \\ \hat{W}^k_{i,j} = round(W^k_{i,j}), \text{ if } m = k \end{gathered}$$

The computation of the rank, see Eq. 4.12, can be done more efficiently than by calculating all the matrix multiplications for all the weights in W (we have to do this for every weight once, so it must be efficient).

Even if we think of W as a 'correct' solution, it is an iterated solution which still has some acceptable error. Rounding of a weight can affect this both ways, it can increase and decrease the extent of the error. That means, by rounding a weight we do not always impair the solution, it can also be improved. To judge the effect of the rounding correctly, we first calculate a base deviation,

$$e^{base} := W^c(W^a\mathbf{1} \odot W^b\mathbf{1}) - n\mathbf{1}, \tag{4.13}$$

and then we add the effect from the rounding to calculate the ranks:

$$rank(i,j,m=a) := |(round(W^a_{i,j}) - W^a_{i,j})\langle W^b_{i,:}, \mathbf{1}\rangle W^c_{:,i} + e^{base}|_2 \tag{4.14}$$

$$rank(i,j,m=b) := |(round(W^b_{i,j}) - W^b_{i,j})\langle W^a_{i,:}, \mathbf{1}\rangle W^c_{:,i} + e^{base}|_2 \tag{4.15}$$

$$rank(i,j,m=c) := |(round(W^c_{i,j}) - W^c_{i,j})\langle W^a_{j,:}, \mathbf{1}\rangle \langle W^b_{j,:}, \mathbf{1}\rangle e_i + e^{base}|_2 \tag{4.16}$$

e_i is the vector with length N with all zeros except for a 1 in position i. $W^a_{i,:}$ is row i of W^a, $W^a_{:,i}$ is column i of W^a.

We only have to compute e^{base} and the products $\langle W^a_{i,:}, \mathbf{1}\rangle W^c_{:,i}$, $\langle W^b_{i,:}, \mathbf{1}\rangle W^c_{:,i}$ and $\langle W^a_{j,:}, \mathbf{1}\rangle\langle W^b_{j,:}, \mathbf{1}\rangle$ once (until we change W), which is more efficient than the direct calculation of Eq. 4.12 for every weight.

The deduction of Eq. 4.13-4.16 from Eq. 4.12 can be done by splitting $\hat{W}$ up into W and $\hat{W} - W$. This way Eq. 4.12 splits up into e^{base} and the effect from the rounding. $\hat{W} - W$ only contains zeros and one element that is unequal to zero.

```
Data: $N, p$
Result: initialized $W$
draw $W$ from uniform distribution on $[-1, 1]$;
$W$:=backpropagation($W$);
set $M^a, M^b, T^a, T^b$ to $\mathbf{1}^{p\times N}$;
set $M^c, T^c$ to $\mathbf{1}^{N\times p}$;
repeat
    $i, j, m := \operatorname{argmin}_{i,j,m:T^m_{i,j}=1} rank(i,j,m)$;
    $T^m_{i,j} := 0$;
    $M^m_{i,j} := 0$;
    $W^* := W$;
    $W^{*,m}_{i,j} := round(W^{*,m}_{i,j})$;
    $W^*$, success:=backpropagation($W^*, M^a, M^b, M^c$);
    if success then
        $W := W^*$;
    else
        $M^m_{i,j} := 1$;
    end
until $T^a, T^b, T^c$ all 0;
```

Algorithm 7: *Initialisation step to get better starting values for Alg. 6*

Algorithm 7 now just replaces the random intialisation of W in Algorithm 6.

4.3 Parameters

For Projection 4.5 of the Difference-Map Algorithm, we only have the practical solution with Eq. 4.11. We want to find a triple of matrices $P_B(W)$ that is close to $W = (W^a, W^b, W^c)$. Given W, the projection distance $|W - \mathit{backpropagation}(W)|_2$ depends on the learning rates η^c and η^{c*} for the backpropagation algorithm, and on the random values used as input for the backpropagation algorithm. To investigate the dependency of the projection distance on the learning rates, we calculate the average projection distance for a range of learning rates, we set η^c and η^{c*} to the same value. Figure 4.3 shows the learning rates and average projection distance dependency for the cases $n = 2$ and $n = 3$. The learning rates identified for fast convergence of the backpropagation algorithm, see Figure 3.2, do not fit well with the Difference-Map Algorithm, they lead to high projection distances. Instead, we experienced learning rates $\eta^c = \eta^{c*} = 0.1$ to work well.

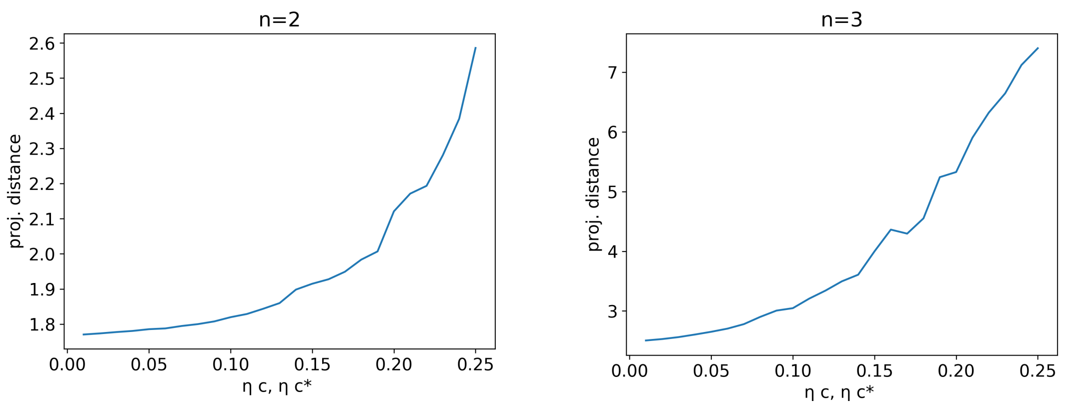

Figure 4.3: *Learning rates and average projection distance of the backpropagation algorithm, for the cases $n = 2$ and $n = 3$*

In the Difference-Map Algorithm, if we find that Δ^W stagnates for too long, we want to add random values to W^a, W^b and W^c. We draw the random values from a uniform distribution on $[-1, 1]$, then we multiply them with a jump factor before we add the result to the design matrices. Figure 4.4 shows two effects: the blue line is the average number of iterations needed to come from randomly initialized design matrices to a solution. We always stop after a maximum of 5000 iterations of the Difference-Map Algorithm, even without a valid solution. So the average values used here are biased towards lower numbers of iterations. The number of iterations decreases with a growing jump factor. But at the same time, the green curve shows, that with a growing jump factor the number of failed iterations is growing. Here, ‘failed iteration’ means, that the backpropagation algorithm (inside the Difference-Map Algorithm) is diverging, and so the run of the Difference-Map Algorithm is failing. The search for an optimal 'jump factor' was only done for the case $n = 3$. Since failed runs are expensive, because of the initialization step, we chose a jump factor of 0.25.

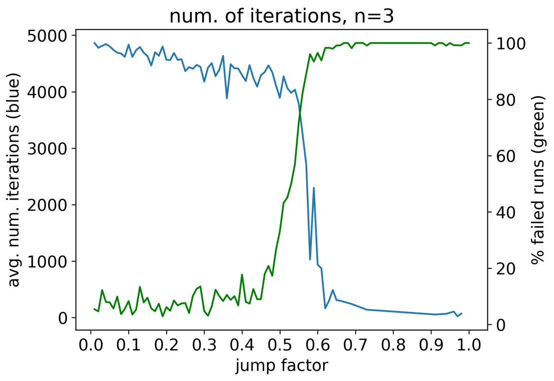

Figure 4.4: *Influence of the jump factor on the number of iterations and failed runs*

For the parameter z and q in Alg. 6, we experienced $z := 10$ and $q := 6$ to be a good setting.

5 Heuristic algorithm for finding the integer solutions

We develop a second algorithm that is easier to understand but also delivers solutions. It is based on the three steps:

- selection and rounding of one weight of W to the next nearest value in $\{-1, 0, 1\}$, and the blocking of the weights position for further iteration
- backpropagation to fix the error caused by the rounding
- reset of a random number of blocked weights to be available for the iteration again

The algorithm selects the weight of W that can be rounded to $\{-1, 0, 1\}$ with minimal error introduced to the solution. The weight is rounded and we try to eliminate the error with a run of the backpropagation algorithm. If we succeed in eliminating the error, we block the position in W so the weight stays fixed at the rounded value. We repeat the process by selecting and rounding the next weight.

If we do not succeed with the backpropagation run, for a long enough time, it means that rounding of the selected weight is not possible any more given the blocked positions in W. We made a mistake in the past with the selection of the weights to be rounded and blocked. We neither know the number of mistakes we made, nor the positions that should not have been blocked. To find a way out of this blind alley, we draw a random number to decide how many blocks to remove, then we draw random positions of blocked weights and unblock them.

We continue with the selection and rounding of the next weight. The algorithm finishes with a correct solution, when all weights are rounded and blocked.

We use the masks M^a, M^b, M^c (with the dimensions of W^a, W^b, W^c) to block or unblock positions to be alterable or not alterable in the backpropagation run, see Eq. 3.7-3.9.

For the selection of the next weight to be rounded we use Eq. 4.14-4.16 to rank the weights and then select the one with the minimal rank.

$$i, j, m := \underset{i,j,m: M^m_{i,j}=1}{\operatorname{argmin}} \; rank(i, j, m), \quad m \in \{a, b, c\} \tag{5.1}$$

The rounding is done with Eq. 4.1.

```
**Data:** $N, p$
**Result:** solution $W$
draw $W$ from uniform distribution on $[-1, 1]$;
$W$:=backpropagation($W$);
set $M^a, M^b$ to $\mathbf{1}^{p \times N}$;
set $M^c$ to $\mathbf{1}^{N \times p}$;
**repeat**
    find next position $i, j, m$ to round with Eq. 5.1;
    $W^m_{i,j} := round(W^m_{i,j})$;
    $M^m_{i,j} := 0$;
    $W, success := backpropagation(W, M)$;
    **if** *not success* **then**
        $M^m_{i,j} := 1$;
        $z := sum\ of\ zero - positions\ in\ M^a, M^b, M^c$;
        draw $r \in \mathbb{Z}$ from uniform distribution $[1, z]$;
        set $r$ zero-positions in $M^a, M^b, M^c$ to 1 (with same prob. for all pos.);
    **end**
**until** *solution found*;
```
Algorithm 8: *Heuristic search algorithm for finding integer solutions*

The algorithm sequentially builds up the solution and destroys parts of it once it runs into a blind alley, hopefully the blockage is removed by these random resets.

The algorithm is accelerated, if we run several instances. If one of the instances finds a solution that is better than all solutions found before, the other instances copy this solution and continue their work from there. After a short time the instances will diverge from the common solution because of the random resets. After some time one of the instances will find a new, best, solution to be copied by the other instances.

The algorithm proved to be able to find solutions for the cases of $n = 2$ and $n = 3$. The implementation is slower than the implementation of the difference map algorithm. One advantage of the heuristic search algorithm is, that it does not have parameters that need to be adjusted (it also is a meta algorithm, there are still parameters in backpropagation algorithm). For the difference map algorithm, the search for good parameters is very time consuming.

6 Algorithm for the reduction of additions and subtractions

Now we want to look at the number of additions/subtractions in our algorithms. We take a look at an example of a bilinear matrix multiplication algorithm for the case $n = 2$:

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right)$$

We want to look closer at the following expression:

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad \text{or} \quad b^* = W^b b$$

If we execute the calculation of b^* as the product of only one matrix with one vector, we need to execute the following operations:

row-1: No operation necessary, we only set $b_1^* \leftarrow b_1$.

row-2: No operation necessary, we only set $b_2^* \leftarrow b_3$.

row-3: 1 subtraction, we calculate $b_3^* \leftarrow b_4 - b_2$.

row-4: 1 addition and 2 subtractions, we calculate $b_4^* \leftarrow b_1 + b_2 - b_3 - b_4$.

row-5: 1 addition, we calculate $b_5^* \leftarrow b_1 + b_2$.

row-6: 1 addition and 1 subtraction, we calculate $b_6^* \leftarrow b_1 + b_2 - b_4$.

row-7: No operation necessary, we only set $b_7^* \leftarrow b_4$.

So, in total we need 7 additions/subtractions.

The multiplication $b^* = W^b b$ can be split up (how the split-up is done, is explained later):

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} (\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & -1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}) + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

If we want to calculate $y = M^1 x + M^2 x = (M^1 + M^2)x$, there are two cases where additions/subtractions occur:

- We find more than one non-zero in a row of a matrix, in this case there must be additions/subtractions when we build the inner product between the matrix-row and the vector.
- We are adding two vectors, in this case we have an addition for all the positions where not at least one of the two vectors is (guaranteed) zero.

In our example, by executing 1 addition and 1 subtraction we get:

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} (\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} b_2 - b_4 \\ b_1 + b_2 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}) + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

The remaining matrix-vector multiplications have the important property, that all rows of the matrices only contain one or zero non-zero values per row. So, except for negations, we do not need any operations for matrix-vector products to progress in the calculation:

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} (\begin{bmatrix} b_2 - b_4 \\ b_1 + b_2 \\ b_2 - b_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ 0 \\ 0 \end{bmatrix}) + \begin{bmatrix} b_1 \\ b_3 \\ 0 \\ -b_3 \\ 0 \\ 0 \\ b_4 \end{bmatrix}$$

Now we need 1 addition for the first elements of the two vectors in the brackets.

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b1 + b_2 - b_4 \\ b_1 + b_2 \\ b_2 - b_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_3 \\ 0 \\ -b_3 \\ 0 \\ 0 \\ b_4 \end{bmatrix}$$

After one matrix-vector product that does not involve any operation (except for one negation) we get:

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -(b_2 - b_4) \\ b_1 + b_2 - b_4 \\ b_1 + b_2 \\ b_1 + b_2 - b_4 \\ 0 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_3 \\ 0 \\ -b_3 \\ 0 \\ 0 \\ b_4 \end{bmatrix}$$

Only the addition of two vectors is left. Only 1 subtraction is necessary here:

$$\begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ b_4^* \\ b_5^* \\ b_6^* \\ b_7^* \end{bmatrix} = \begin{bmatrix} b_1 \\ b_3 \\ -(b_2 - b_4) \\ b_1 + b_2 - b_3 - b_4 \\ b_1 + b_2 \\ b_1 + b_2 - b_4 \\ b_4 \end{bmatrix}$$

So by calculating b^* this way, we only need 4 instead of 7 additions/subtractions (we do not count negations).

6.1 Reduction of operations through common expressions

For the task of calculating the product Mx of matrix M and vector x, how do we find a good decomposition for reducing the number of additions/subtractions?

A natural way is to look for a good set of common expressions in the sums (matrix row times vector). We can calculate the value of the common expression once, and then assemble the target sums from the common expressions. Because we cannot substitute everything with common expressions, we need to complete the missing parts with further operations. The operations necessary to calculate the common expressions can possibly again be reduced by finding common expressions in the common expressions. So the principle can be applied recursively until no further decomposition is found.

The decomposition we find, and the number of operations we eventually need, are dependent on the list of common expressions we use and by the order of the common expressions in the list.

These are conditions on the list of the common expressions used in our algorithm:

- ◦ Every common expression in the list has to be fully contained in at least two target sums (matrix row times vector).
- ◦ Every common expression has to contain at least one addition/subtraction.
- ◦ For every common expression in the list, the negated common expression cannot be in the list.

For now we assume a given list L^1 of k common expressions of M. If we have $M \in \{-1,0,1\}^{n\times m}$, we can store the list L^1 in form of a matrix $L^1 \in \{-1,0,1\}^{k\times m}$. Every row of L^1 represents one common expression of M.

We can now go through the list L^1 from the top and for every common expression l we find all the rows in M where the common expression is contained. We subtract l (or $-l$) from all the rows we found (even if we only found one, this is important because of the recursion). Now we have to store the information about where to insert the common expression later. We use the matrix $A^1 \in \{-1,0,1\}^{n\times k}$ to mark with 1 (or -1) in which places to use the common expression later, A^1 is initialized with zeroes. If row j of L^1 is used in row i of M, we need to set $A_{i,j} := 1$ if the common expression is not negated. If we have to negate the common expression, we set $A_{i,j} := -1$.

Now we can reassemble M by:

$$M^1 := A^1L^1,\ R^1 := M - M^1 \quad \Longrightarrow\ M = M^1 + R^1 = A^1L^1 + R^1 \tag{6.1}$$

Our calculation of Mx can now be written as:

$$Mx = A^1L^1x + R^1x$$

In the matrix A^1 we will possibly find all-zero columns, that means, that a common expression of L^1 is never used. If we find column j of A^1 to be all zero, we have to delete that column. We then also have to remove row j of L^1. In this way L^1 is shortened.

We hopefully already need less operations now. But, possibly, we can split up the shortened L^1 further by the same principle, using a list of common expressions of L^1, we call that list L^2.

$$M^2 := A^2L^2,\ R^2 := L^1 - M^2 \quad \Longrightarrow\ L^1 = M^2 + R^2 = A^2L^2 + R^2 \tag{6.2}$$

And then combine Eq. 6.1 and Eq. 6.2:

$$M^1 := A^1(A^2L^2 + R^2),\ R^1 := M - M^1 \quad \Longrightarrow\ M = A^1(A^2L^2 + R^2) + R^1 \tag{6.3}$$

Our calculation of Mx can now be written as:

$$Mx = A^1(A^2L^2x + R^2x) + R^1x$$

This is how we found the decomposition used in the example.

If we continue the recursion to depth r, we get the decomposition:

$$M = A^1(A^2(A^3...(A^rL^r + R^r) + ... + R^3) + R^2) + R^1$$
$$Mx = A^1(A^2(A^3...(A^rL^rx + R^rx) + ... + R^3x) + R^2x) + R^1x \tag{6.4}$$

We stop the recursion if no more common expressions can be found (according to the conditions mentioned above).

To reduce the number of additions/subtractions, the execution order is important. We need to calculate 'from the inside to the outside'. For the execution order, look at the example given above. Also, now that we split up the matrix vector products in Eq. 2.4, we can not exactly use Alg. 4 any more.

6.2 How to find the lists of common expressions

To create a list L of common expressions for the matrix M, we suggest the following algorithm.

We initialize the list L with all the rows of M that have at least two non-zeros.

Then we look at every expression l in the list L that has more than two non-zeros. For all the selected expressions we create all possible expressions that are contained in the expression l and have exactly one non-zero less than l, and at least two non-zeros. We append the new expressions to L. We continue until no new expressions can be created. Of course, we do not add those expressions to the List L that can be found in L already (or their negation).

The next step is to build a vector f of occurences for all the expressions. The vector f contains the number of occurence in M for all the expressions in L. Here we can already filter out all the expressions that have less than two occurences.

We also calculate a vector p that contains the number of operations saved, if we only substituted this single common expression. If the number of non-zeros in the expression is k and the number of occurences is f_*, we would save $p_* = (k-1)(f_*-1)$ operations.

Now we can sort L by descending order of p and have a good list of common expressions, but we do not claim that this is the optimum.

6.3 The operation reduction algorithm

We give the pseudo-code for the algorithm to reduce additions and subtractions. The pseudo-code does not contain the recursion. The list L we receive from Algorithm 10 can be used again in the same algorithm as input (instead of M) for the recursion.

```
Data: $M$
Result: $L$
$L := M$;
remove rows with less than two non-zeros from $L$;
remove duplicate rows from $L$;
$end := 0$;
repeat
    $start := end + 1$;
    $end := length(L)$;
    for $i = start..end$ do
        $nonZeros := list\ of\ indices\ where\ L_{i,:}\ is\ non-zero$;
        for $j = 1..length(nonZeros)$ do
            $newExpr := L_{i,:}$;
            $newExpr_j := 0$;
            $occ := number\ of\ occurences\ of\ newExpr\ in\ M$;
            $noZer := number\ of\ non-zeros\ in\ newExpr$;
            if $newExpr\ not\ yet\ in\ L\ \&\ noZer > 1\ \&\ occ > 1$ then
                add newExpr to $L$;
            end
        end
    end
until $no\ new\ expressions\ added$;
calculate $f$ and $p$;
sort $L$ by descending order of $p$;
```

Algorithm 9: *Creation of an expression list as input for Alg. 10*

After we created the list L with Alg. 9, we can now reduce the operations with Alg. 10:

```
Data: $M$
Result: $A, L, R$
$L :=$ create list of common expressions in $M$, use Alg. 9;
$A :=$ zero matrix;
forall $i$ in numb. of rows in $L$ do
    forall $ii$ in numb. of rows in $M$ do
        if $L_{i,:}$ is contained in $M_{ii,:}$ then
            $M_{ii,:} := M_{ii,:} - L_{i,:}$;
            $A_{ii,i} := 1$;
        else if $-L_{i,:}$ is contained in $M_{ii,:}$ then
            $M_{ii,:} := M_{ii,:} + L_{i,:}$;
            $A_{ii,i} := -1$;
        end
    end
    forall $j$ in number of columns of $A$ do
        if column $A_{:,j}$ is all zero then
            remove column $A_{:,j}$;
            remove row $L_{j,:}$;
        end
    end
end
$R := M - AL$;
```
Algorithm 10: *Reduction step for the reduction of additions and subtractions*

Experience shows, that with Alg. 10 the number of additions and subtractions can be reduced by roughly one third in the case $n = 3$.

7 Found solutions

7.1 For n=3

During the development and adjustment of the algorithms, several thousand correct solutions for the case $n = 3$ have been found. Only the solution with the lowest number of operations is shown here.

$$W^a = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
\mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
-1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & -1 & 0 & 1 & 1 & 0 & -1 \\
-1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\
-1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{0} & \mathbf{0}
\end{bmatrix}, \quad
W^b = \begin{bmatrix}
-1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{0} & \mathbf{0} \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
-1 & 1 & -1 & -1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{0} & \mathbf{0}
\end{bmatrix},$$

$$W^c = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 0 & 1 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

We have a solution with 23 multiplications, if we use W^a, W^b and W^c in Alg. 4 ($\tilde{U} := W^a$, $\tilde{V} := W^b$, $\tilde{W} := W^c$).

The design matrices shown here have some interesting properties: rows 13 and 22 (green), as rows 21 and 23 (blue) of W^a are identical. In W^b the rows 3 and 23 (magenta) are identical. It would be interesting, to search for solutions with many identical rows in the design matrices W^a and W^b, since this leads to reduced computation effort. Identical rows in W^c can not occur, unless we can reduce r further.

To reduce the number of additions and subtractions, we compute the decomposition of W^a, according to Eq. 6.4:

$$A^{a1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad A^{a2} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad L^{a2} = \begin{bmatrix} -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$R^{a1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad R^{a2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

According to Eq. 6.4: $W^a a = A^{a1}(A^{a2}L^{a2}a + R^{a2}a) + R^{a1}a$.

We do the same for the decomposition of W^b :

$$A^{b1} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad A^{b2} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad L^{b2} = \begin{bmatrix} -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix},$$

$$R^{b1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}, \quad R^{b2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix},$$

and the decomposition of W^c:

$$A^{c1} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}, \quad A^{c2} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad A^{c3} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

$$L^{c3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$R^{c1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$R^{c2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$R^{c3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}.$$

With this decomposition, we need 62 additions and subtractions and 23 multiplications for the computation of $mat(c) = mat(a) \cdot mat(b)$. If we use the standard Alg. 1, we need 18 additions and 27 multiplications.

So far, our strategy was to find many solutions and then pick the one with the lowest number of operations. If we search further, we might find a better solution. Also, a better strategy to minimize the total number of operations probably exists.

7.2 For n=5

We also use the Difference-Map Algorithm to search for matrix multiplication algorithms for the case $n = 5$. We do this, to show that the algorithm is not limited to the cases $n = 2$ and $n = 3$. However, the computation time needed for $n = 5$ is already very high. We found a solution with 117 multiplications ($n^3 = 5^3 = 125$). By searching longer we could probably find better algorithms. The solution for $n = 5$ can be downloaded from the GitHub repository `https://github.com/ubik80/searching-for-fast-MM-algorithms`.

8 Search behavior of the Difference-Map Algorithm

We want to have a look at the behavior of the Difference-Map Algorithm with regard to our search space. Our analysis is not exact, it is based on observations and some fantasy.

The implementation of the Difference-Map Algorithm often shows a steep descent during the last few iterations before we find the solution. This suggests, that once we are in some proximity of the solution, we do not have to search there for very long, and the difficulty is more to find some suitable area in the search space and to come close enough to a solution.
Figures 8.1 show four examples of runs where this behavior can be seen. For most parts of the runs, there is no decrease of $|\Delta^W|$ below a certain threshold, $|\Delta^W|$ is even increased at times. Once we are in the right area of the search space, $|\Delta^W|$ drops very quickly to zero within a few iterations.

So, the difficulty in finding a solution lies in getting inside a comparably small area around the solution. Figure 8.2 shows our rough idea about what the search space could look like. We start the iteration at the red dot. The green jumps are the iterations, the length of the jump is $|\Delta^W|$. We have to hit a relatively small area in the search space to come to a solution where $|\Delta^W| = 0$. This also gives an explanation, why finding a solution is relatively difficult and time consuming, a fair quantity of luck is involved in a successful search.

A reason why we are still finding solutions is, that there is not only one solution reachable from the same starting value. For the case $n = 2$ we start four runs from the same starting value and run into four different solutions. This is shown in Figure 8.3. The dimensions are reduced with 'T-distributed Stochastic Neighbor Embedding' (t-SNE) to come to a two dimensional visualization. Starting from the green dot, we see more or less clusters visited by the algorithm during the runs. The algorithm is searching inside the clusters before moving to the next cluster, often this is caused by the addition of random values, marked in magenta. Once we come close enough to a solution, only a few iterations with small $|\Delta^W|$ are necessary to find the solutions, marked by red dots.
We also find the distribution of the number of iterations for the found solutions interesting. In retrospect, if we knew Histogram 8.4 in advance, we could have used it to adjust the maximum number of iterations in Alg. 6 more sensibly.

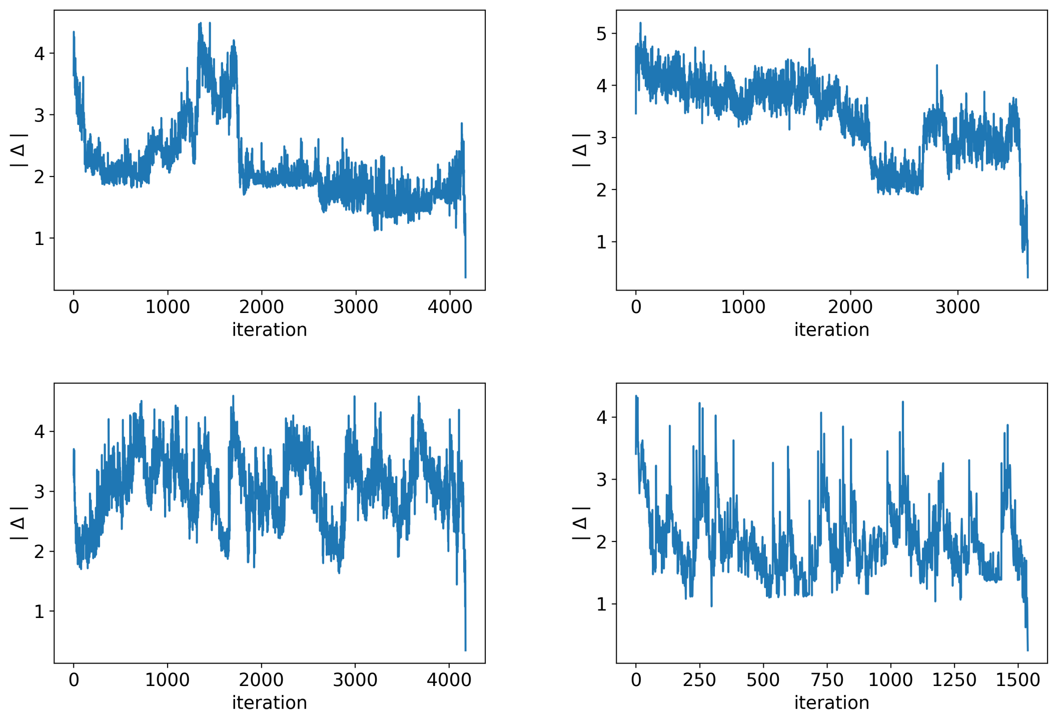

Figure 8.1: *Examples for a fast drop into the solution at the end of the run (n=3)*

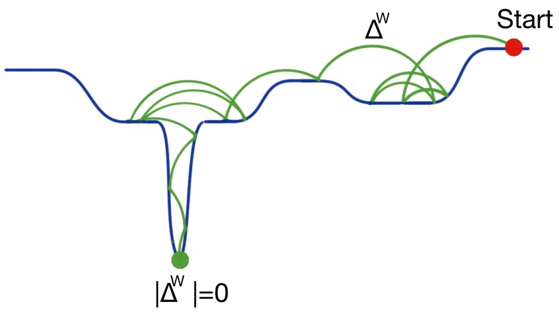

Figure 8.2: *comical drawing of our idea of the searched space*

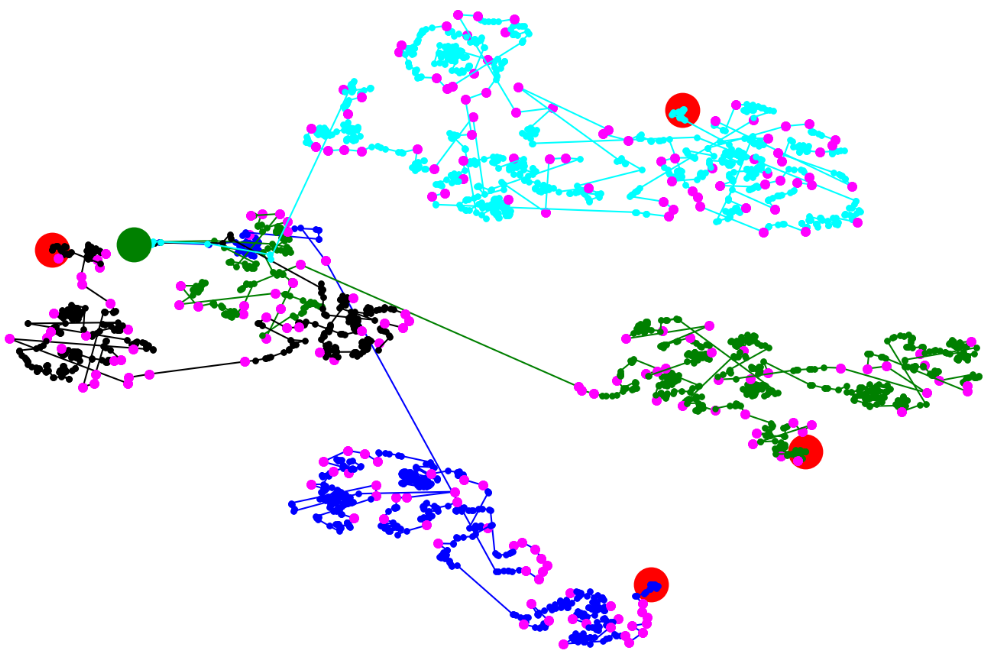
Figure 8.3: *Example of four search runs, dimensions reduced with TSNE (n=2)*

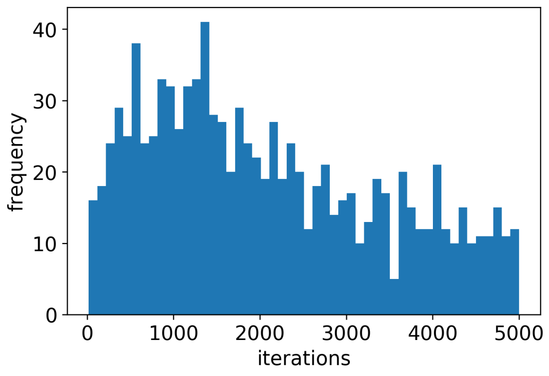

Figure 8.4: *Frequency of numbers of iterations for found solutions (n=3)*

9 Implementation details and availability

The implementation was mainly done in python (*python 3.7*). For some parts of the code, a just-in-time compiler was used (*numba*). The backpropagation algorithm, that consumes most of the running time, was implemented as a C++ extension (with *pybind11*). Even so, it is obvious that the backpropagation algorithm is the most time consuming part of the whole solution, that assumption was confirmed using a profiler (*line_profiler*). To shorten the running time, the solutions is implemented to run on multiple cores (*multiprocessing* package for python).

The code can be downloaded from the authors GitHub repository `https://github.com/ubik80/searching-for-fast-MM-algorithms`. Also, the found solutions can be downloaded from there.

10 Conclusions

This master's thesis shows, that 'artificial neural networks', combined with other algorithms, can find exact matrix multiplication algorithms by 'training'. The solutions found are not just the training-state of a neural network that produces an approximate solution for most inputs, but exact algorithms that can easily be understood and applied by human users.

The combination of the Difference-Map Algorithm, as a meta algorithm, and the backpropagation algorithm is capable of finding matrix multiplication algorithms with fewer than n^3 products.

For $n > 5$, measures have to be taken to significantly speed up the solution, either through optimization of the algorithm or its implementation, finding better sets of parameters, or the use of many processors in parallel.

Parallelisation can easily be achieved by just starting many instances of the algorithm in parallel (with different seeds for the random number generator), communication between the instances is not needed.

The long running time of our search algorithms makes it difficult to find good parameter sets through experiments. This is one of the flaws of our solution, it shows the value of algorithms with few or no parameters for computation intensive problems.

Large improvement potential probably lies in finding a better solution for the implementation of Eq. 4.5. Our choice here, Eq. 4.11 is responsible for most of the running time.

There is only a little theoretical knowledge about the Difference-Map Algorithm applied to the case of non-convex constraint sets. Further research on this topic could possibly lead to improvements in our solution as well.

The matrix multiplication algorithms found with the Difference-Map Algorithm can be processed further to also reduce their number of additions and subtractions, we describe a simple algorithm for this purpose.

For practical purposes, the algorithms we found are probably not relevant since the costs of multiplications and additions are not so different anymore in modern computers. Efficient use of main memory, cache, vector units, threads etc. are much better instruments to shorten the running time.

However, the adaption of our solution to other bilinear problems could be a direction for further research.

11 References

[1] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik. 13. 354-356. 10.1007/BF02165411*, 1969.

[2] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation, 9 (3): 251, doi:10.1016/S0747-7171(08)80013-2*, 1990.

[3] V. Elser, "A network that learns Strassen multiplication", *Journal of Machine Learning Research. 17*, 2016.

[4] M. Bläser, "Fast matrix multiplication", *Graduate Surveys, 5:1-60*, 2013.

[5] H. Groote, "Lectures on the complexity of bilinear problems", *Lecture Notes in Computer Science*, 1986.

[6] V. Elser, R. I., and T. P., "Searching with iterated maps", *PNAS January 9, 2007 104 (2) 418-423, doi.org/10.1073/pnas.0606359104*, 2007.

[7] J. Laderman, "A noncommutative algorithm for multiplying 3x3 matrices using 23 multiplications", *Bulletin of the American Mathematical Society*, vol. 82, Number 1, 1976.

[8] N. T. Courtois, G. V. Bard, and D. Hulme, "A new general-purpose method to multiply 3x3 matrices using only 23 multiplications", 2011. [Online]. Available: arXiv:1108.2830.

[9] A. Sedoglavic, "A non-commutative algorithm for multiplying 5x5 matrices using 99 multiplications", *CoRR*, vol. abs/1707.06860, 2017. arXiv: 1707.06860. [Online]. Available: http://arxiv.org/abs/1707.06860.

[10] V. Strassen, "Vermeidung von Divisionen.", *Journal für die reine und angewandte Mathematik*, vol. 264, pp. 184–202, 1973. [Online]. Available: http://eudml.org/doc/151394.

Erklärung der Urheberschaft

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung unzulässiger Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Seitens des Verfassers bestehen keine Einwände die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, 30.9.2019 T. Späth

Ort, Datum Unterschrift